MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Real-time Communication in Web Browser

MASTER THESIS

Pavel Smolka

Brno, 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: doc. RNDr. Tomáš Pitner, Ph.D.

Acknowledgement

Thanks to everyone... TODO: Petr Kunc (help with OpenTok & RealTime communication). Thanks Káťa for cooking and being patient while I was typing the thesis regularly up to 3 AM

Abstract

TODO

Keywords

XMPP, real-time communication, RTC, Celebrio, web browser, HTTP, Comet, JavaScript, WebSockets, BOSH, Ember, Strophe, OpenTok, TODO

Contents

Chapter 1

Introduction

Millions, billions, trillions. That many and even more messages are exchanged every day between various people over the world. The Internet created brand new way to communicate and collaborate, even if you are located on the opposite parts of the world. Since the times of Alexander Graham Bell, the accessibility to the communication devices and their simplicity incredibly enhanced. Nowadays, almost 2.5 billion people over the world have access to the Internet and therefore they are able to use almost limitless communication possibilities it provides. [26]

However, the manner of Internet usage essentially changed during the first decade of 21st century. Using the Internet and using the web browser became almost synonyms. People use the web browser as the primary platform to do every single task on the Internet. Sometimes it's not even possible to use the other Internet services without visiting certain web page in the web browser and performing the authentication there.[1] Considering the mentioned fact, web browsers became also the basic platform for the communication tools. Even though the purpose of the world wide web and HTTP protocol was completely different at first (displaying single documents connected via hypertext links), it appeared that there is the need of common rich applications running withing web browser - rich Internet application sprung up. [41] So popular social networks are built on top of the web browser platform and they are used by more than billion people over the world. [16] And the main reason why the social networks are so popular is the real-time stream of news and messages from the other people. By the beginning of the year 2013, I would say that static web is dead - users prefer interactivity.

This thesis embraces the topic of real-time applications in web browsers, especially the text communication tools and the technologies being used to develop them. It also describes the problem of "inter-process" communication between various web pages which need to cooperate and exchange information as quickly as possible. Finally, the possibilities of multimedia content transfer (audio and video) and the current options of capturing multimedia directly from the web browser are described as well.

As mentioned above, the web browser became one of the most popular platforms. Cele-

1. Two examples of such behavior. Wi-fi network in the Student Agency coaches forces the user to visit the entry page in the web browser. The second example, very well known to the students of the Faculty of Informatics at Masaryk University, is the faculty wireless network called wlan_fi. Every user has to open the web browser and log in with her credentials. It's not possible just to open the terminal or e-mail client and start working online.

brio, simple software for the elderly simulating the operating system interface, is typical example of rich Internet application. [8] All the topics mentioned in the previous paragraph appeared to be very important in the system. When interviewing the elderly people in the Czech Republic, it appeared that almost 90 % of the elderly computer users use the real-time communication (RTC) applications, mostly Skype. [13] Interaction with their loved ones is the most desired benefit they expect from the computer. Therefore, creating real-time application, text messenger supporting video calling, became not only programming challenge but also a business goal.

TODO: rework this a bit. Mention all RTC tools, not only web based. Mention more history:

Considering the fact the people like real-time communication (not only direct messaging but also real-time cooperation, simultaneous document editing or playing multiplayer games) while using web browser brings us the question what the currently available solutions are. There are "big players" providing their own services as closed-source, without the possibility to use them. To name the most important, it's Google Talk web browser client and Facebook chat, using XMPP protocol. [22][18] Even though Facebook chat service is not pure XMPP server implementation (the message and presence engine is proprietary system of Facebook implemented mostly in C++ and Erlang), they provide the possibility to connect to the "world of Facebook Chat" via XMPP as proxy. [19] Combination of the facts that XMPP is open technology with open-sourced client and server implementations [68] and the big Internet companies also use it persuaded us to use it in our communication application too. XMPP itself and its usage in web applications is described in Chapter 3, "Extensible Messaging and Presence Protocol".

Since the web browser was designed to perform simple requests/response interaction, it is not a typical platform for building real-time application. Thus, there is a need for an extra layer enhancing or even completely replacing the common way HTTP communicates. Within the scope of this thesis, namely the WebSockets and HTTP long polling approaches are used. The two of them and basic information about several others are covered in Chapter 2, "Bidirectional communication between web browser and server".

There are many existing real-time chat-based applications over the Internet we could have used. However, none of them suited our needs perfectly. Celebrio has very specific graphical user interface (GUI) and there is a need to integrate both text-based chat and video calling. Just to mention, there is commercial chat module Cometchat[2] or even open project Jappix.[3] Video calling web browser applications are provided for example by TokBox Inc.[4] Nevertheless, following the rule that "If you have to customize 1/5 of a reusable component, its likely better to write it from scratch", [7] just very generic existing libraries (Strophe.js) and APIs (OpenTok) were used for implementing brand new application called *Celebrio Talker*. The general approaches when building web browser based chat application are mentioned in Chapter 4, "JavaScript XMPP client". Within the programming part

2. http://www.cometchat.com/
3. https://project.jappix.com/
4. http://www.tokbox.com/

of the thesis, the real-time text chat application, video calling application and simple "inter-process" communication tool for Celebrio has been implemented. Celebrio Talker application itself, its architecture and the specific procedures used to create it are described in Chapter 5, "Talker - IM client in web browser".

It has been said that Skype[5] is the most favorite communication tool among the target audience. If it had been implemented, the existing customer base could be used and converted to our messaging application. However, there is one big pitfall in this approach. Skype license strictly prohibits incorporating their software into mobile devices. [44] They support only prompting the official Skype client to be opened via Skype URI, which is insufficient for Celebrio since the messaging client has to be built in the system, with the corresponding user interface. [45]

Finally, there are several notes about "inter-process communication" between different applications running separately in various browser frames, tabs or even windows. XrefId[??] covers this topic and describes the issues we came across when implementing such functionality for Celebrio, where every application runs in separate iframe.

5. http://www.skype.com/

Chapter 2

Bidirectional communication between web browser and server

Suggestion TODO: split the chapter: describe HTTP more in depth, tell why HTTP is so important (not only because of browsers), describe the history of communication tools. (maybe in the intro)

Suggestion TODO: mention RTC history (unix talk, IRC, ICQ - Meebo)

The very essence of every instant messaging is the bidirectional stream where both sides can immediately *push* new data and the other side (or more other sides, respectively) is promptly notified without the need to perform any manual *pull* (update) action.[1] Such use case requires appropriate transport layer on top which the application can send the messages via any other protocol. When using HTTP, there is a TCP connection opened by the client (web browser) through which the data is sent. However, according to the HTTP protocol, the communication is strictly initiated by the client - HTTP is a request/response protocol. [39] When the client needs still up-to-date information, it must poll the server as frequently as possible. Such approach takes a lot of bandwidth and generates purposeless overhead on the server. So, when one wants to avoid that drawbacks and still make the web browser application to communicate in both directions, HTTP protocol must be hacked somehow or another communication channel has to be used. This chapter covers both - reshaping HTTP in Section 2.1 and brand new approach in Section 2.2, bypassing HTTP at all. Unfortunately, every approach brings also some disadvantages. Ultimately, overview of several higher-level solutions is stated in Section 2.5, most of which are based on HTTP requests or WebSockets.

2.1 Using HTTP requests

Historically the first approach, for a very long time the only one, is hacking HTTP to achieve RTC. The idea is very simple. Generally, the client sends an extra request and it is not awaiting the response immediately. Instead, the server keeps the request for some time and sends the data as it comes in the response. There are several techniques to achieve such behaviour, in general called *Comet*.

There is one common misunderstanding about long-lived HTTP requests. Since HTTP 1.1 (actually implemented even before, but not covered in the RFC specification), there

1. In this thesis, this behaviour is commonly referred as RTC. The "real-time part" relates mostly to the server part since the application running in the web browser can perform the AJAX request on background anytime and the server receives the request instantly.

is a possibility for the client to claim persistent TCP connection to the server declaring `Connection: Keep-Alive` in the request header. Actually, all connections are considered persistent unless declared otherwise. [39] Even though the default timeout after which the server closes the connection is only a few seconds, [3] the persistent TCP connection is very useful for delivering various resources (style sheets, scripts, images, etc.) to the client without the unnecessary overhead of creating multiple streams. However, every single transmission within the TCP connection has to be in form of separate HTTP request/response, always initiated by the client. On no account is the server allowed to push any data without respective request from the client. Therefore, such TCP connection is of no use for RTC.

The situation is depicted at Figure 2.1 and Figure 2.2. In the former case, a valid sequence of HTTP requests and responses is shown. Nevertheless, there is a delay between the moment the server gets (either generates or receives from third party) the data (2) and the following request (3). Yet it is possible to reduce the latency by shortening the polling time (the time between Response (1.1) and Request (3)), it is still trade-off between the delay and overhead caused by frequent empty request/response pairs.

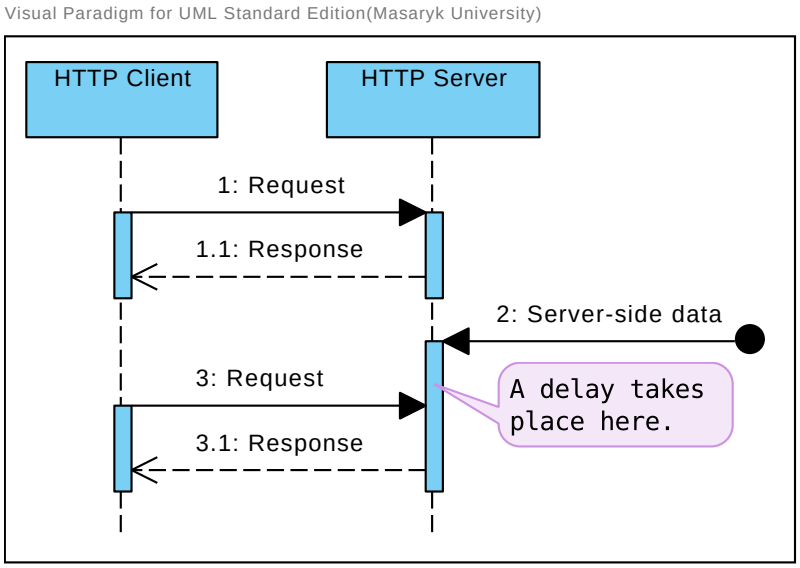

Figure 2.1: Correct HTTP polling with delay

On the other hand, Figure 2.2 depicts the forbidden situation of generating HTTP response without request immediately. When the server gets the data (2) it is not allowed to initiate the connection and send HTTP response without appropriate preceding request (2.1). Even though the delay, mentioned in the previous paragraph, can be minimized in this situation, HTTP servers cannot use such technique. To sum it up, response (2.1) is forbidden by HTTP protocol and this situation solution is not valid.

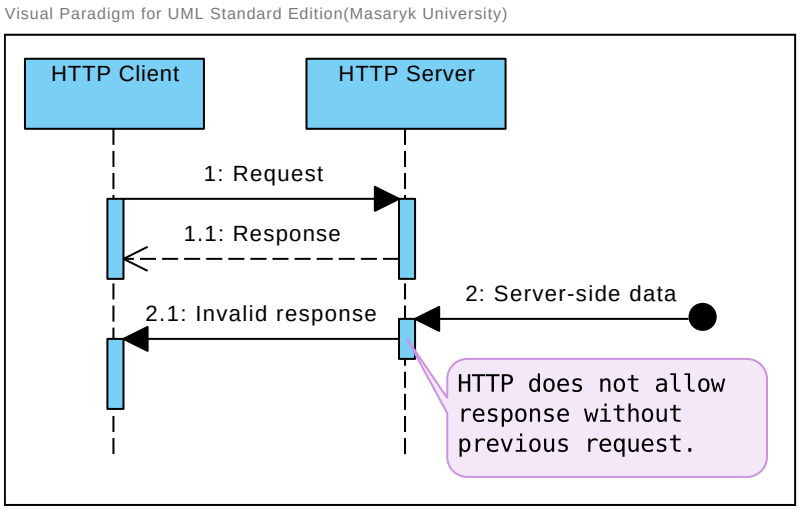

Figure 2.2: Forbidden HTTP response without respective request

2.1.1 HTTP long polling

The essence of HTTP long polling springs from the idea of prolonging the time span between two poll requests. In traditional "short polling", a client sends regular requests to the server and each request attempts to "pull" the available data. If no data is available, an empty response is sent. [38] That generates unnecessary overhead for both client and server.

On the contrary, long polling tries to reduce this load. After receiving the request, the server *does not* answer immediately and holds the connection opened. When the server receives (or even makes up by itself) new data, it carries out the response with the respective content, as depicted at Figure 2.3.

As soon the client obtains the response, it usually issues a new request immediately so the process can repeat endlessly. If no data appears on the server for certain amount of time, it usually responds with empty data field just to renew the connection.

One of the main drawbacks of long polling is header overhead. Every chunk of data in RTC applications is usually very short, for example some text message of minimal length. However, each update is served by full HTTP request/response with the header easily reaching 800 characters. [35] In case the payload is a message 20 characters long, the header constitutes 4000% overhead! This drawback has even bigger impact as the number of clients increases. Figure 2.4 shows the comparison of 1000 (A), 10000 (B) and 100000 (C) clients polling the server every second with the message 20 characters long, both using classic HTTP requests and WebSockets technology (mentioned in Section 2.2). [35] It is obvious that there is huge unnecessary network overhead when using HTTP polling instead of WebSockets.

Furthermore, in case the server just received the data and sent the response to the client, there is a "blind window" when the server cannot notify the client. Whole push system is

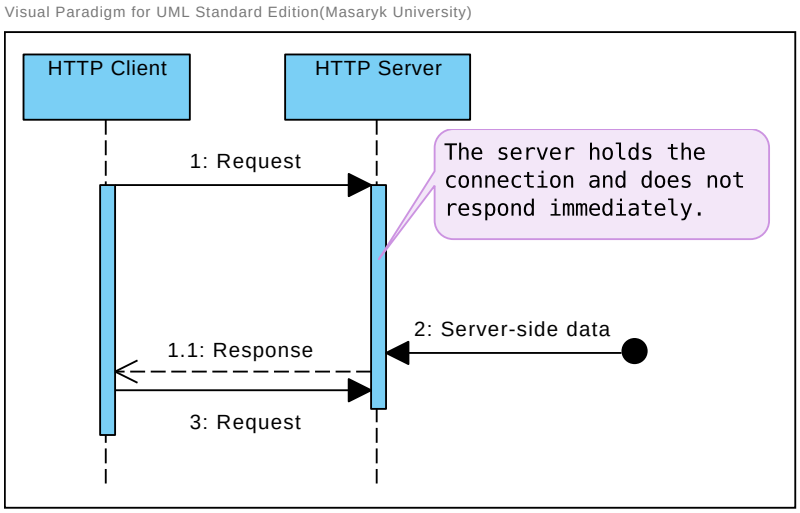

Figure 2.3: HTTP long polling

blocked until the response is received by the client, processed and new request is delivered back to the server. Considering also the possible packet loss and required retransmission in TCP protocol, the delay can be even longer than double bandwidth latency. [38]

2.1.2 HTTP streaming

HTTP streaming is slightly different technique than long polling, although they are confused one with the other very often. What is mutual for both of the approaches is the client initializing the communication by HTTP request. The server also sends the update as the part of the HTTP response. The main difference is that once the server initializes the response and sends the data, it does not terminate the response and keeps the HTTP connection opened. Meanwhile, the client listens to the response stream and reads the data pushed from the server. When any new data springs up on the server side, it is concatenated to the one existing response stream. [38] See the schema at Figure 2.5.

It is very important not to confuse HTTP streaming with the "persistent" HTTP requests. As said at the beginning of this chapter, declaring `Connection: Keep-Alive` does not allow the server to issue multiple responses to a single request. Such behaviour would be serious violation of HTTP protocol. Instead, the server can declare `Transfer-Encoding: chunked` status in the response header and send the response split into separate pieces, as show below (chunk of zero length stands for the end of the response): [38]

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

25
This is the data in the first chunk
```

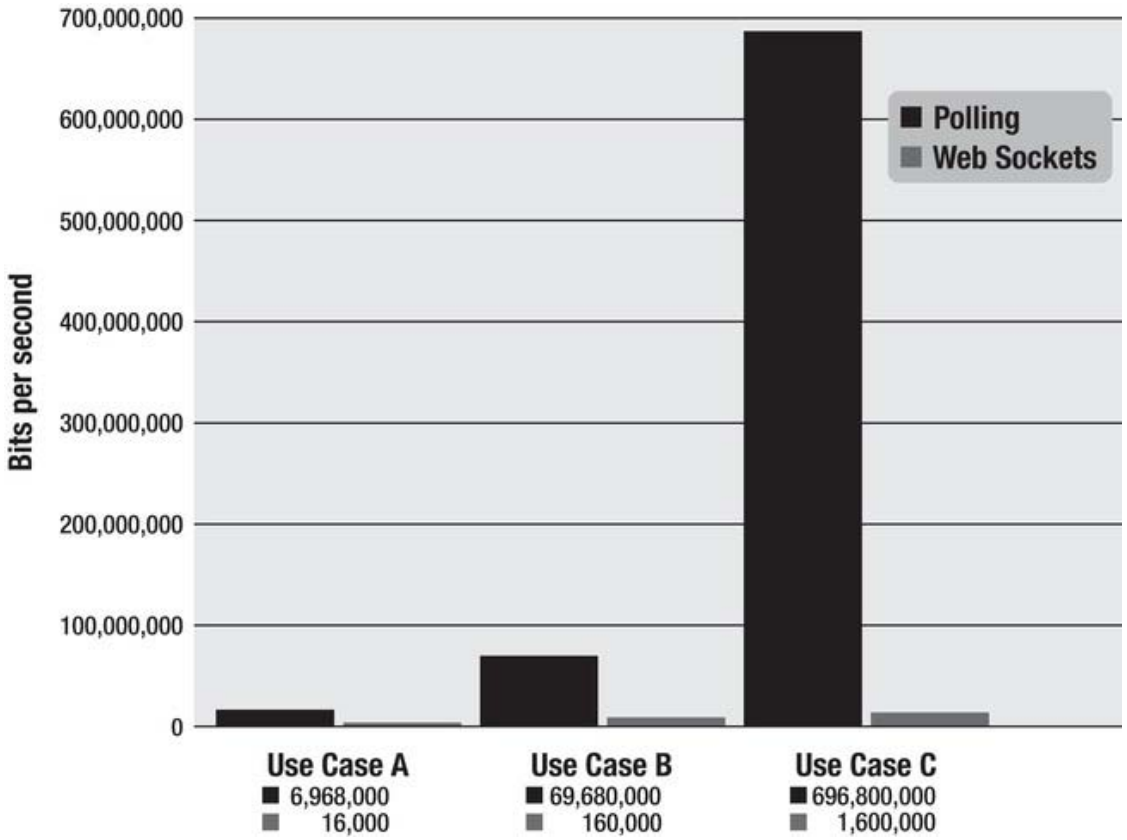

Figure 2.4: Comparison of network overhead (HTTP and WebSockets)

```
1C
and this is the second one

0
```

The main drawback of HTTP streaming can be generally called buffering. There is no requirement for both the client and any intermediary (proxies, gateways, etc.) to handle the incoming data until whole response is sent. Therefore, all parts of the response could be kept by the proxy and the messages (single HTTP response chunks) are not delivered to the client until whole response is sent. Similarly, when the response consists of JavaScript statements, the browser does not have to execute them before whole response is obtained (yet, most of the browsers execute it immediately). In such cases, HTTP streaming will not work. [38]

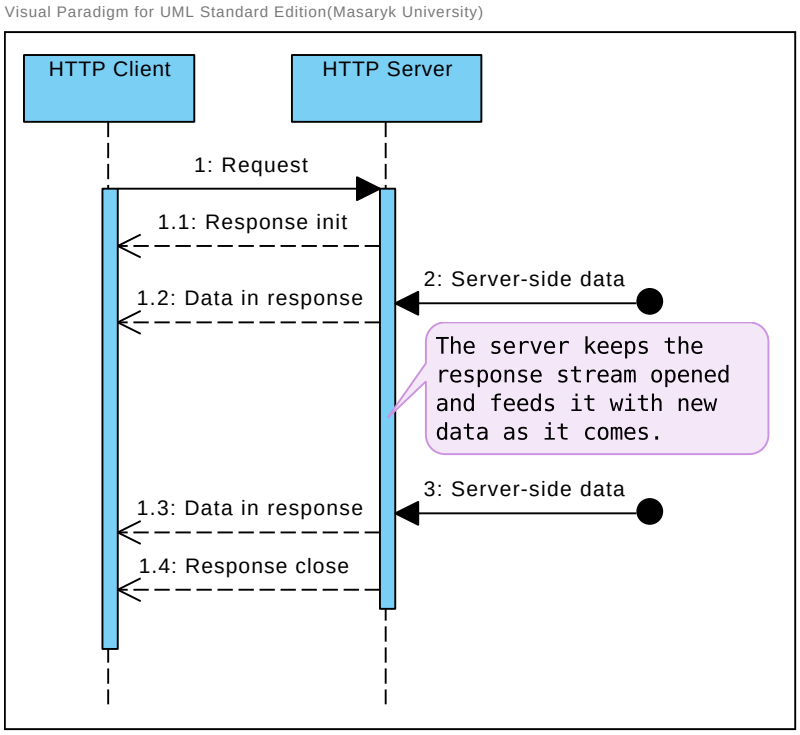

Figure 2.5: HTTP streaming

2.2 Permanent TCP streams with WebSockets

Although the World Wide Web with HTTP request/response schema has never been intended to serve as RTC platform, the contemporary applications require such functionality and the developers started to bend the protocol in the undesirable way. Most of the patterns described in Section 2.1.2 do their jobs and one can achieve sufficient two-way communication, but there are certain performance issues and drawbacks which make them difficult to use. At least, those techniques carry HTTP header overhead which is unnecessary for standard bidirectional streams. Therefore, brand new standard for creating full-duplex communication channels between the web browser and the server has been created. The technology is called *WebSockets* (sometimes shortened as WS) and it stands for a communication protocol layered over TCP along with the browser API for web developers. Anyway, not even WebSockets are allowed to access wider network, the connection possibilities are limited only to the dedicated WS servers (usually HTTP servers with additional module for WS support attached). [30]

Similarly as in HTTP, there is an unencrypted version of WebSockets working directly on top of TCP connection. The simplest way to recognize such connection is WebSocket URI beginning with `ws://`. It should not be used on account of two reasons. The first one, quite obvious, is security - the communication can be captured during the transmission. Transpar-

ent proxy servers are the second reason. If an unencrypted WebSocket connection is used, the browser is unaware of the transparent proxy and as a result, the WebSocket connection is most likely to fail. [63][12] As opposite, there is a secure way how to use WebSockets. WebSockets Secure (WSS) protocol is standard WS wrapped in TLS tunnel, similarly as HTTP can be transmitted over TLS layer. When using WSS, the URI begins with `wss://` and it uses port 443 by default. [12]

2.2.1 WebSocket Handshake

Alike any other multilateral protocol WebSockets need to perform a handshake before actual transmission can take place. During the handshake, the connection is established and both peers acknowledge the properties of the communication.

Since WebSockets emerged as HTTP supplement, the handshake is initialized by HTTP request[2] initialized by the client. The client sends the request as follows: [40]

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Let us have a look what each of the lines means. The first two lines are obvious, they represent typical HTTP GET request. Specifying `Host` is important for the server to be able to handle multiple virtual hosts on single IP address. The following two lines, `Upgrade: websocket` and `Connection: Upgrade`, are the most important. The client informs the server about the desire to use WS. The rest of the request stands for additional information for the server to be able to respond correctly - RFC 6455 describes them in detail.

The server should send HTTP response looking similar as this example: [40]

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Number 101 on the first line of the response stands for HTTP status code `Switching Protocols`, [39] which means the server supports WebSockets and the connection can be established. Any status code other than 101 indicates that the WebSocket handshake has not completed and that the semantics of HTTP still apply. [40]

2. According to RFC6455, the protocol is designed to work over HTTP ports 80 and 443 as well to support HTTP proxies. However, the design is not limited to HTTP and the future implementations can use simpler handshake over a dedicated port. [40]

`Sec-WebSocket-Key` is random secret issued by the client and added to the initial protocol-switching request. The server is supposed to concatenate the secret with Globally Unique Identifier (GUID) "258EAFA5-E914-47DA- 95CA-C5AB0DC85B11" and hash the result with SHA-1 algorithm. The results returned as `Sec-WebSocket-Accept` header field, base64-encoded. [40] However, there seems to be a security issue here. If the initial request is not sent over encrypted HTTP connection (HTTPS), it can be caught by a third party. Since the server does not authenticate in any way and the algorithm does not contain any server secret, the third party attacker could fake the response and pretend to be the server.

2.2.2 Frames and masking

All the data sent via WebSockets protocol is chunked into frames, working similarly as TCP frames. All the transmission features are handled by the WebSocket API and the transmission is transparent for the application layer above (for example JavaScript API) so that every message appears in the same state as it was sent. This means the message, single portion of WS communication, can be fragmented during the transmission.

There are several special features concerning WS frames, one of the interesting is masking. The payload data of every frame sent from the client is XORed by the masking key of 32-bit size. The purpose of masking is to prevent any third party from picking any part of the payload and reading it. The other goal might be distinguishing the server stream from the client stream instantly since client-to-server frames always *must* be masked and server-to-client frames *must not* be masked under any circumstances. In addition, WS peers have to use masking even if the communication is running on top of TLS layer so the "encryption" function is pointless. [40] The security function of masking is also questionable because the masking key is included in the frame header. The only reason is preventing from random cross-protocol attacks. [35]

2.2.3 WS JavaScript API

Since WebSocket technology is intended to be used particularly from the browser applications, there is need of an API web developers can use. The most widespread programming language of web browser client applications is JavaScript and so is the WebSocket API created for. The API consists of one relatively simple JavaScript interface called `WebSocket`,[3] [59] placed as the property of `window` object.[4] It wraps the WebSocket client functionality performed by the user agent (web browser). Using the API is very simple - the object, which handles all the WS functionality, is created by calling `WebSocket()` constructor: [61]

3. In the older versions of some browsers, the interface was called differently due to the immaturity of the technology. For instance, Firefox from version 6 to 10 supports WebSockets only as `MozWebSocket`. Interesting fact is that Firefox 4 and 5 provides `WebSocket` interface as it is, just implementing different WebSocket protocol version. Since Firefox 11.0, current (RFC 6455) WS protocol version is accessible via interface `WebSocket`. [33]
4. Properties of window are accessible in JavaScript directly. Simple test `window.WebSocket === WebSocket` returning `true` proves it.

```
var connection = new WebSocket(
        'ws://html5rocks.websocket.org/echo',
        ['soap', 'xmpp']
);
```

The first (mandatory) argument stands for the WebSocket URI the client attempts to connect to. It can either begin with ws:// prefix or wss://, depending whether TLS layer is used or not. The second parameter is optional - specific WS subprotocols can be demanded there. Since there are only a few subprotocols recorded by IANA registry, it is of a little use so far. [62]

`WebSocket` interface provides at least four event handlers, to each of them a custom callback can be attached. [59] Those are `onopen`, `onmessage`, `onerror` and `onclose`. The names are quite self-explanatory, they serve as the event listeners watching the incoming activity - anytime the websocket obtains a message, its status changes or an error occurs, the respective callback is fired. The callback registration can look as follows:

```
connection.onmessage = function (message) {
  console.log('We got a message: ' + message.data);
};
```

In addition, there is a property `readyState` (it would be `connection.readyState` in the previous example) keeping current WebSocket status all the time. The status can be retrieved by testing the property against one of the `WebSocket` property constants `CONNECTING`, `OPEN`, `CLOSING` or `CLOSED`.

Sending the data to the server is also quite straightforward. Either `DOMString`, `ArrayBuffer`, `ArrayBufferView` or `Blob` can be sent via `send()` method. See the examples below: [61]

```
// Sending String
connection.send('string message');

// Sending canvas ImageData as ArrayBuffe
var img = canvas_context.getImageData(0, 0, 400, 320);
var binary = new Uint8Array(img.data.length);
for (var i = 0; i < img.data.length; i++) {
  binary[i] = img.data[i];
}
connection.send(binary.buffer);

// Sending file as Blob
var file = document.querySelector('input[type="file"]').files[0];
connection.send(file);
```

To sum it up, using WebSockets became a very simple and elegant way to provide real-time communication channel between web browser and WS server. The main drawback of WS is lack of support not only in the older versions of web browsers but also in the mobile platform browsers. Currently less than 60 % of users can make use of WebSocket

full support. [60] Particularly, all versions of Internet Explorer below 10 (which means more than 98 % of IE users in November 2012) [24] does not implement WebSockets JavaScript API. There are two favourable aspects in favour of WebSockets. Firstly, more and more web browsers add JavaScript API to support WebSockets. Secondly, the ratio of clients who use old version of the web browser without WS support tends to diminish. Anyway, if the real-time functionality constitutes the application core functionality, there is a strong need to offer fallback technology that every browser supports - usually represented by HTTP long polling or streaming mechanism, described in Section 2.1.

2.2.4 WebSocket API wrappers

Yet WebSockets is powerful technology, there are still many browsers not supporting it. In that case, when real-time communication constitutes the core functionality, a fallback (i.e. the alternative technology used when the original is missing) must be defined to substitute WebSockets. It might be Flash or HTTP polling. It would be great not to have to define the fallback in every project again and again. Luckily, there are several API wrappers for this.

The basic use case is obvious. Using the wrapper instead of WS API itself guarantees the developer that a fallback is used when the application runs in the environment without WebSockets. Whole process of choosing the transport technology is transparent and not necessary to be specified. As examples, however not used within Talker application, projects Socket.IO[5] and SockJS should be mentioned.[6]

2.3 Server-sent events

Server-sent events (aka EventSource, from this point referred as SSE) should in fact not be listed here but in the next section. It is technology based on HTTP streaming described in Section 2.1.2 so it is not at the basic "zero" level. However, SSE are often compared to WebSockets so that the topic is introduced here. SSE have been standardized as part of HTML5 standard. [48] There is very brief summary of SSE API and usage in this section.

As any other web technology, SSE connection must be initialized by the client. There is a JavaScript API providing event handlers, very similar to WS API. The event stream is opened with the constructor, pointing to the resource at the server:

```
var eventSource = new EventSource("sse-example.php");
```

The script at the server, `sse-example.php` in our case, pushes the data to the opened HTTP response stream. What is important to get SSE work, the `Content-Type` header must be set to the value `text/event-stream`. The data has to be organized in form of "paragraphs" separated by blank line, where every paragraph stands for one message. Have a look at an example below:

5. http://socket.io/
6. https://github.com/sockjs/sockjs-client

```
data: This is one-line message

id: 123
event: myevent
data: Message of type "myevent" which consists of several lines
data: Another line of the event message
```

As shown above, every line in the message consists of key and value separated by colon, similar to JSON format. When we need to transfer multi-line message, we can repeat the key several times. [49] On the basis of `event` field (`myevent` in our example), the respective event handler is triggered in JavaScript API. In our case, it would be the following event listener, if it has been attached in JavaScript before, logging the event to the console:

```
eventSource.addEventListener("myevent", function(e) {
    // process the event
    console.log(e);
}, false);
```

The connection is closed either by the client by calling `close()` method on the `EventSource` object or by the server (when all data is sent). However, if server closes the connection, the client attempts to reconnect to the same resource. So, the server cannot close the connection permanently.

Sever-sent events are often compared to WebSockets, though they are much less known. The support of both in current web browsers is very similar. The only main difference is lack of support of SSE in Internet Explorer 10, which finally provides WebSocket API. Another difference is the fact that SSE, unlike WS, does not provide real bidirectional stream. Only the server can publish new messages through the opened HTTP connection, the client has to push the data to the server via other (standard) HTTP requests. Finally, there is quite a big limitation in the message format. While WebSockets provide real TCP connection any data can be transferred through, SSE is restricted to textual data in the form of key/value pairs. So, there is no compelling reason to use SSE when WebSockets exist. Anyway, it comes in handy to know about it when working with real-time applications in web browsers.

2.4 Media streaming with WebRTC technology

Up to now, none of the technologies mentioned was fully suited as a complete solution for web browser based media communication, such as video calling. WebSockets are the most advanced, however, it is only low level API providing TCP stream. On that account, web browser developers, with Chromium developers in the vanguard, invented WebRTC technology. Is is an API linking up the user media API (webcam, microphone) with the streaming API for sending the multimedia from the browser to the other node.[7]

While WebSockets serve as the interconnection between the client (web browser) and the dedicated server, which makes it suitable for "server-based" protocols such as XMPP,

7. Apart from video calling, WebRTC provides an API for sending the files from one peer to another.

WebRTC provides real peer-to-peer connection, directly between two web browsers. That makes WebRTC a perfect tool for implementing direct media communication, such as video calling.

2.4.1 Signaling

In fact, the server has to mediate the "meta data", such as initializing connection or negotiating the available media capabilities (such as codecs). This level of communication, exchanging the information about connection itself, is called *signaling*. WebRTC does not take care nor about the layer signaling data is transferred at, neither the signaling protocol itself. It can SIP, XMPP or any other, transferred via XMLHTTPRequest or WebSockets. What is important, signaling is not part of WebRTC API. The WebRTC connection itself then concerns only the peers, as shown at Figure 2.6. [56]

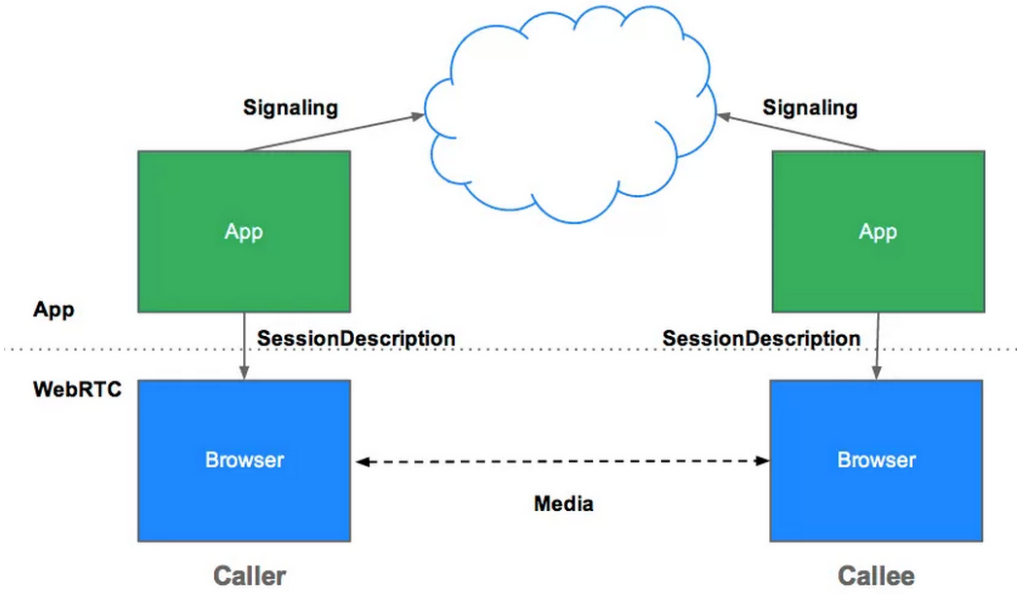

Figure 2.6: WebRTC communication schema

2.4.2 WebRTC API

WebRTC provides very high level API abstracting media device access, network connection and encoding/decoding the media streams from the programmer. Unfortunately, the WebRTC API is still in the phase of draft and has not been standardized yet. [55] It means that JavaScript objects are prefixed by vendor prefixes, so that there is `webkitRTCPeerConnection` instead of `RTCPeerConnection` in Chromium.

The core API object is JavaScript object RTCPeerConnection. Creating one may look like the following (with respective prefix):

```
var config =  {"iceServers": [{"url": "stun:stun.l.google.com:19302"}]};
var pc = new RTCPeerConnection(config); // webkitRTCPeerConnection
```

Then, we can send all available ICE[8] candidate to the other peer, via specified STUN server. ICE candidate is basically a possible transport address for media stream, later validated for peer-to-peer connectivity. [23] So, each possible connection address is sent via previously created `pc` object. The process is still in the phase of negotiation so sending is up to the signaling service:

```
pc.onicecandidate = function (event) {
    // use existing signaling channel to send the candidate
    signalingChannel.send(JSON.stringify({ "candidate": event.candidate }));
};
```

In case the other side publishes its video stream, we set up a hook which handles it and shows it in the remote video element, stored in `remoteView` JavaScript variable. [55] In other words, the incoming URL is assigned to the `video` element as a source (`src`) attribute. The `src` attribute determines the media source played within the element. [11]

```
pc.onaddstream = function (event) {
    remoteView.src = URL.createObjectURL(event.stream);
};
```

Sending the media stream from the local browser to the other peer is similar. First, we capture audio and video stream from the local multimedia devices. The result, multimedia stream, is passed as the argument to the function which adds it to the `RTCPeerConnection` object `pc`. Besides, it is a common habit to add the video of self to the page as well. It is handled by `selfView` variable, containing reference to another video element on the page. The example below represents the described situation:

```
navigator.getUserMedia({ "audio": true, "video": true }, function (stream) {
    selfView.src = URL.createObjectURL(stream);
    pc.addStream(stream);
});
```

Closing the connection is quite straightforward, by invoking `close()` method on `RTCPeerConnectio` object which has been instantiated before as `pc`.

8. ICE stands for Interactive Connectivity Establishment

2.4.3 WebRTC in various environments

WebRTC is truly new technology, not yet adopted by many web browsers. And for those which support it, the implementation may differ since the standard has not been fully defined and finished yet. Chromium browser (i.e. Chrome and Chromium) developers were the first who added WebRTC API to their products - in 2012. By the beginning of the year 2013, WebRTC API was added also to the newest Firefox builds so that Chrome and Firefox can "talk" to each other.[9] [53] Opera browser also takes part in this initiative, yet no existing official claim of support has been released. All of the mentioned browsers used the same backend implementation, hosted at <http://www.webrtc.org>. In theory, this backend implementation (written in C++) can be built into any application to support WebRTC, not only a web browser.

For the web browsers which does not support WebRTC yet, there are several attempts to add its functionality via browser plugins. It can be useful as a temporary improvement for the experienced users but one can never rely on the user having the plugin installed. The library providing WebRTC functionality for Safari, Opera, IE and older versions of Firefox is called webrtc4all. [58] However, it is still proprietary solution, adding one more prefix (`w4a`) to the world of WebRTC JavaScript APIs.

There are also other parties which implement WebRTC in their own way, keeping to the API more or less. One of such initiatives is Ericsson Browser (called just Browser), which claimed self to be the first browser to implement WebRTC. Ericsson Browser uses different backend implementation but it tries to be in accordance with the official API. [52]

Microsoft came up with another approach. Even though Internet Explorer does not support WebRTC, Microsoft invented their own API standard proposal. [54] It differs from the "official" API mainly in the extended possibilities to control more aspects of WebRTC communication, including low level "transport" layers. Since the none of the standards has been finished yet, it is possible (and probable) that the final API definition will result somwhere in between.

One of the logical reasons why Microsoft does want to intervene to the process of defining WebRTC API as much as possible is Skype. Skype has been bought by Microsoft some time ago and, as everything nowadays, is planned to be available in the web browser. Microsoft seems to bet on WebRTC technology. [57] As a pleasant side-effect of Skype working on top of WebRTC, it would be finally possible to play along with Skype with other technologies. Of course, the technology barrier is the smaller one compared to licences and legal regulations, but it is another story.

2.5 Other RTC solutions

Actually, there are several other, mostly higher-level, solutions for achieving bidirectional (and thus real-time) communication in web browser. Some of them use HTTP requests de-

9. Note that in current Firefox version (22, nightly build by the time this thesis part is created), the user has to enable `media.peerconnection.enabled` field in `about:config` to get WebRTC run.

scribed above (such as Bayeux), some of them are based on WebSockets (OpenTok) and several are built completely independent, installed as web browser plugins and thus behaving as separate runtime platforms (Adobe Flash, Google Talk). And, to be precise, some of the frameworks are built on top of others, for example OpenTok can use Flash. See the sections below to understand each of the technologies.

2.5.1 Adobe Flash

Among all, one of the most widespread technologies is Adobe Flash.[10] Apart from the possibility to establish bidirectional persistent TCP connection, Flash allows the developer to create almost any graphics, animations and user interface with nearly no limits.

Nevertheless, the disadvantages of Flash are significat. First, Flash is not a native part of any web browser. Until recently, it had to be installed manually as a plugin. Now, it is bundled and shipped with Chromium-based browsers (Chrome, Chromium), but it is still external plugin. [20] Another drawback of Flash is lack of support on mobile devices. Apple has been clear about it: iPhones and iPads have never supported Flash technology and it is not likely to change in the future. [27] Android devices supported Flash at first, but later Adobe quit Google Play. [1]

There are many other similar technologies such as Adobe Flash, but they all suffer the same pain. Since they are installed as proprietary plugins, the developer can never be sure the application will run in any environment. This concerns technologies like Microsoft Silverlight[11] or Adobe AIR[12] (even though AIR has been intended to be browser-independent platform).

So, Adobe Flash is often used as a fallback for older browsers, running on non-mobile devices, which does not WebRTC yet (described below). It has been used as fallback for the video calls in Talker application. However, creating a new application based on Flash (as a primary technology) in 2013 is not a good idea.

2.5.2 OpenTok video call library

TODO: OpenTok

2.5.3 Bayeux protocol

Bayeux is one of the higher level protocols designed specificaly for bidirectional communication between web browser and the server, primarily intended to work on top of HTTP. The idea of communication is almost the same as in case of HTTP long polling. For transfer of the messages from the server to client, the server holds the request and responds only when there is available message. Sending the data from the client to the server is straightforward, ordinary HTTP request is sufficient.

10. http://www.adobe.com/software/flash/about/
11. http://www.microsoft.com/silverlight/
12. http://www.adobe.com/products/air.html

Apart from HTTP requests, Bayeux defines the structure for the transmitted data, contained in the body of HTTP requests/responses. Each message has to be in form of JSON, containing structured data such as channel name, client ID and of course the transmitted data itself. [6] Bayeux provides interesting way to communicate, yet it has not been used in this thesis. The main reason for choosing XMPP is much better interoperability with other existing services and ability to easily communicate with the clients not running in web browser.

2.5.4 SignalR framework

SignalR is a framework taking care of both client (web browser) and server side of the application. SignalR is designed for .NET platform on the server side, so the web application should be powered by ASP.NET framework on the server side. It abstracts the user from finding out which technology the web browser the application runs into supports. Instead, SignalR provides API for sending the messages and handling the incoming ones. Internally, it uses WebSockets for establishing the connection. When the WebSocket API is not available (for example in IE9, which is fun fact considering that SignalR is Microsoft-platform-based), it tries Server Sent Events and then falls back on long polling technique. [43] To take is short, SignalR provides envelope for client-side WebSockets and HTTP long polling techniques, with ASP.NET API for handling the messages on the server.

2.5.5 Real-time communication tools from Google

TODO: Google Talk service (mention it runs with XMPP),
TODO: Google hangouts API,
TODO: Google App Channel (from app engine i guess)

Chapter 3

Extensible Messaging and Presence Protocol

Suggestion TODO: xmpp: to chce XMPP trochu více popsat a zasadit ho do kontextu HTTP komunikace a tu zase do celkového provozu na internetu mezi ostatní protokoly

Extensible Messaging and Presence Protocol (XMPP) technologies were invented by Jeremie Miller in 1998. [71] It is one of the most widespread technologies for instant messaging (IM),[1] i.e. exchanging the text or multimedia data between several endpoints. The "native" implementation of XMPP works right on top of TCP protocol: XMPP endpoint (called client as it represents the first actor in client-server architecture) opens long-lived TCP connection. Then, both the client and the server negotiate and open XML streams so there is one stream in each direction. [71] When the connection is established, both client and server can push any changes as XML elements to the stream and the other side obtains them immediately. Usual XMPP clients are standalone applications able to open TCP connection and listen to the stream opened by the server.

XMPP stands for communication protocol handling not only sending and receiving the messages, but also presence notification, contact list (roster) management and others. The architecture is distributed and decentralized. There is no central or top level XMPP server. Anyone can run XMPP server, very similarly as HTTP or FTP server. Identification and recognition on the network is also similar - XMPP relies on Domain Name System (DNS) so that every server is identified via string domain name with arbitrary subdomain level (e.g. xmpp.example.com or just example.org). [71]

The user name, called Jabber ID (or shortly as JID), has the same structure as e-mail address so the user name is followed by `@` and the server domain name. This rule also guarantees that every XMPP user is registered at the certain server. If there is any message or notification for the given user, her "home" server is looked up first, the message is transferred to that server first and then, the respective server (that the user belongs to) is responsible for delivering the message to the user or saving it until the user logs in. Therefore, two possible connection types take place in XMPP. Client-to-server communication is the first one, when the clients can talk only to their "home" server. Then, server-to-server communication is designed for delivering the messages to the users at different domains. When two servers are exchanging any data, direct connection to the target server is established. This approach is dissimilar to the way SMTP servers exchange e-mail messages. It helps to prevent address

1. Acutally, the IM client or even the techology itself is sometimes called "Instant Messenger". This term is registered as a trademark by AOL company. [2]

spoofing or spamming. [71]

XMPP has been chosen as the communication protocol for this thesis topic - Talker application. XMPP has been verified by big companies such as Google or Facebook. In addition, the openness of the protocol allows very easy connection to existing wide communication networks, using their server infrastructure, client software and existing user base.

3.1 XML Stanzas - XMPP building blocks

As mentioned in the introduction to this chapter, when XMPP connection is established, two streams are opened and both the client and the server can send any XML elements at any time. The meanings of various pieces of XML are described in this section.

There are three three basic XML elements that every XMPP communication consists of. Those are `<message/>`, `<presence/>` and `<iq/>` (which stands for Info/Query), altogether called `Stanza`s. [71] Each stanza element usually contains several attributes which specify the exact meaning of it. The actual content is usually placed in the element body. Example message stanza can look like this:

```
<message from="pavel.smolka@celebrio.cz/talker"
         to="tomas.pitner@celebrio.cz"
         type="chat">
    <body>Hello, how are you?</body>
</message>
```

The attributes `from` and `to` stand for the sender and recipient of the message. Actually, the value the sender sets to the `from` attribute (or whether she entirely leaves it out) does not matter. The "home" XMPP server the sender is registered at (in this example, it would be the one running at `celebrio.cz`) has to set the from attribute according to the real user name and domain name. This is one of the interesting defensive mechanisms distinguishing XMPP from other communication protocols as SMTP.

You might have noticed that `from` field does not contain only the XMPP address. There is a *resource* identifier after the domain name. Since it is possible to connect multiple times with the same user name, the resource makes a difference between the sessions of the same user. In addition, it is useful information for other peers the user might communicate with. It is usual to set the resource field according to the place the user logs from or the device she uses.

Message stanza receiving is not acknowledged so the sender has no information whether it has been delivered successfulIy or not. On the contrary, IQ stanza can be used in case the sender requires the answer - it is usually a *query*. The best example is obtaining the contact list - in XMPP terms called *Roster*:

```
<iq id="123456789" type="get">
  <query xmlns="jabber:iq:roster"/>
</iq>
```

Then, the server sends the result as another IQ stanza (notice that `id` attribute remains the same while the `type` attribute changed): [71]

```
<iq id="123456789" type="result">
  <query xmlns="jabber:iq:roster">
    <item jid="whiterabbit@wonderland.lit"/>
    <item jid="lory@wonderland.lit"/>
    <item jid="mouse@wonderland.lit"/>
    <item jid="sister@realworld.lit"/>
  </query>
</iq>
```

3.1.1 Subscription mechanism

The third letter in the abbreviation XMPP stands for *presence*, practically represented by sending `presence` stanzas. It is one of the important signs of real-time communication (not only in XMPP but overall) that the peers can see each other presence - whether the other side is online, alternatively whether it is available or busy. However this functionality is desired, it might slip to a huge privacy breach when anyone could see your presence status.

XMPP solves the privacy problem with the subscription mechanism. Each user has full control over the peers who can monitor her online status. If anyone else wants to track the presence status, the subscription request must be sent. When received, the user decides whether the permission will be granted or not. Unfortunatelly, the subscription request can be blocked by the respective "home" XMPP server of the user we try to reach. To be tangible: there are two widely used XMPP providers - `jappix.com` and `gmail.com`. If the user of the former sends the subscription to another user registered at the latter, it is not guaranteed it will be delivered (actually, it isn't, see Section 3.5 for details). It is one of the drawbacks of the opened protocol that one can never be sure that the other party co-operates.

3.2 XMPP over BOSH

Having described XMPP as communication protocol over TCP, it might be unclear how it is related to the thesis topic. XMPP is nice and mature technology and it would be nice to use it in web browser but it does not support communication over HTTP. Fortunately, XMPP offers many extensions (indeed, the first letter X stands for "extensible") providing additional functionality. In fact, those are XMPP extension *protocols* and so they are called XEPs.

This section briefly describes one of XEP exensions called BOSH (XEP-0124) designed for transferring XMPP over HTTP.[2] [65] The idea behind this extension is very simple: BOSH uses HTTP long polling technique (described in Section 2.1.1) to imitate bidirectional TCP

2. In fact, there are two more XEPs related to HTTP. First of them, XEP-0025: Jabber HTTP Polling, has been replaced by BOSH. It is obsolete and recommended not to be used any longer. [64] The other one is XEP-0206: XMPP Over BOSH. It is currently used standard but it constitutes just a supplement for BOSH. XEP-0206 describes mainly the session creation and authentication in BOSH. [67]

communication necessary for XMPP. We can imagine BOSH (it is a protocol itself) as a middle layer protocol or wrapper protocol between HTTP (only capable of sending requests from client to server) and XMPP (understanding only the XML stanzas). BOSH requests and responses are subset of all conceivable HTTP requests or responses (they include all HTTP features such as HTTP method in request or status code in the response). The constraint defined by BOSH protocol restricts the body part to have a specific structure.

Each BOSH request or response body is valid XML, which wraps up XMPP stanzas in special `<body />` element. For the purposes of the protocol itself, it is also possible to send just the `body` element with no child (XMPP) nodes - for example when starting the session or reporting an error. So, the XMPP part of the communication is separated from BOSH quite well: the former is represented by payload elements inside the `body`, the latter consists of `body` attributes. Have a look at example of BOSH request: [65]

```
POST /webclient HTTP/1.1
Host: httpcm.example.com
Accept-Encoding: gzip, deflate
Content-Type: text/xml; charset=utf-8
Content-Length: 188

<body rid='1249243562'
      sid='SomeSID'
      xmlns='http://jabber.org/protocol/httpbind'>
  <message to='tomp@example.com'
           xmlns='jabber:client'>
    <body>Good morning!</body>
  </message>
  <message to='pavel@example.com'
           xmlns='jabber:client'>
    <body>Hey, what&apos;s up?</body>
  </message>
</body>
```

As you can see, the request header is ordinary HTTP header. So much for the HTTP part. The request body consists of `body` element which represents BOSH layer, along with the element attributes (plus namespace). `sid` attribute represents the session ID, identifying the connection and not mutated during one session. The other one, `rid`, stands for request ID and it gets incremented with each request. Ultimately, the child nodes of the `body` represent two XMPP stanzas (both of message type). It is obvious that multiple XMPP stanzas can be transmitted via single BOSH request. TODO extract RID/SID/handshake stuff to separate sub-section here?

BOSH protocol is an important part of the Talker application implemented as a programming part of the thesis. Despite it bears the disadvantages of HTTP bidirectional communication, as described before, it is the only reliable technology nowadays. There are several mature client-side libraries using BOSH (such as Strophe.js we used) and it is also easy to install, configure and run BOSH extension at the server side. HTTP server usually hands over

the BOSH HTTP request to XMPP server with relevant module enabled, as described in [42]. However, the server side XMPP is not the topic of this thesis so it is not further discussed.

3.3 XMPP over WebSockets

Since there is possibility to transmit any data from the web browser application to the server via WebSockets, it could be handy to transfer XMPP stanzas using WS as well. Using WebSockets saves a lot of overhead and fixes some issues that can happen with BOSH (for example unreliability of HTTP). Generally it works, yet the programmer should be wary of several pitfalls that WebSockets bring. First, the server side must accept WebSockets connection. Usually, the XMPP servers does provide such functionality through addons or modules.[3] Provided that the WebSocket extension to the XMPP server is running on localhost, using WS to connect to the server is as simple as follows:

```
var ws = new WebSocket("ws://localhost:5280/xmpp-websocket/", "xmpp");
// XMPP handshake takes place here, omitting in the example
ws.send(
 "<message to='lasaris@example.com' xmlns='jabber:client'> \
    <body>Hello, lab!</body> \
  </message>"
);
```

Probably the most important difference compared to BOSH is that every WebSocket message (i.e. one chunk of incoming or outcoming communication - can be compared to BOSH request) can contain only one XMPP stanza. [70] It means that the peer cannot send more WS messages together, even if they are available by the time the WS message is sent.

The main drawback of using XMPP over WebSockets is still partial lack of support both in the web browsers (which includes WS support itself and the JavaScript XMPP libraries) and on the XMPP servers. Nevertheless, there is huge trend of implementing it at all sides.[4]

TODO create diagram here (XMPP server - plugin - incoming connection vs xmpp server - standard connection). Probably for both BOSH and WS.

---

3. Additional module for Prosody server has been used as well when implementing Talker application. The process of installation includes downloading the module, adding it to the path that Prosody searches for modules. Then, it must be enabled in the configuration file. Moreover, `luajit` and `liblua5.1-bitop0` packages had to be downloaded for the module to work correctly (assuming Debian/Ubuntu on the server side).

4. You might want to have a look at some of the current discussions concerning client-side (i.e. JavaScript) libraries:

   - <https://github.com/metajack/strophejs/issues/68>
   - <https://github.com/metajack/strophejs/pull/95>
   - <http://stackoverflow.com/questions/1850162/>

3.4 Jingle - XMPP media extension

Not only can XMPP send text messages but it also supports transferring various multimedia streams. Audio and video above all, yet it is also possible to send raw binary files. All the functionality related to those "advanced" transfers is being managed by XMPP extension called Jingle. According to the protocol extension, the two parties negotiate the data stream using standard IQ stanzas (described in Section 3.1). Then, the stream is established according to the prearranged entries. [66]

Although Jingle is relatively old and mature XMPP extension, it has not been used in web browsers for a long time. Web browsers had not supported multimedia transfers due to the troublesome bidirectional communication and difficult access to multimedia devices (microphone and webcam). Ergo Jingle has not been used in Talker application either - the main reason for not using Jingle in Talker application is lack of JavaScript libraries supporting it. There is just unofficial Strophe.js plugin for Jingle published by Michael Weibel, not very well maintained and relying on the WebRTC technology.[5] Therefore, Jingle is not described further in the thesis, it is mentioned here just as an option for the future.

Anyway, C/C++ Jingle libraries seem to be quite mature and ready to be used in the desktop client applications. Above all, there is library developed by Google called libjingle, supporting multi-user audio/video/file transfers. [32] Google seems to beleive that Jingle is the right protocol for developing multimedia client that can be used by everyone. They even sidelined original Google Talk VOIP protocol and switched to Jingle as their "primary signalling protocol for voice calls", in Gmail, iGoogle and Orkut. [28]

3.5 Interoperability problems

Google Talk has been the biggest XMPP provider for a long time. Smaller XMPP networks, including Celebrio, took advantage of the distributed environment and interoperability within various XMPP domains. It seems, however, that Google starts to prevent the users registered at other server from contacting Google users. More precisely, the subscription requests (described in Section 3.1.1) from other domains are not delivered to Google users. [21] It is very unexpected and unpleasant for all - smaller XMPP providers, their users and for the Google users too, after all. Google confirmed that throwing the subscription requests away is an attempt to reduce the amount of spam delivered to Google users. [46]

If Google closed its network for the people from other domains, it would be the second huge social network, along with Facebook, which does provide XMPP to its users but does not allow them to fully collaborate with everyone else. Unlike Facebook, Google users can still send the subscription requests and therefore initiate the connection (in the long-term meaning), but this option can be disabled in the future too.

Fortunatelly, severals weeks later, Google announced that the alleged spam problems are gone and XMPP network is opened again. [47] Hence, the users having their accounts

5. See <http://candy-chat.github.com/candy-webrtc/> and <https://github.com/mweibel/strophejs-plugins/tree/jingle> for the references.

registered at external XMPP providers can send invites (subscription requests) to Google users without problems.

Chapter 4

JavaScript XMPP client

In this chapter, the two previous topics are connected together. The possibilities of bidirectional communication, necessary for receiving messages real-time, were mentioned in Chapter 2, "Bidirectional communication between web browser and server". In simple terms, such techniques (HTTP long polling, WebSockets, ...) stand for the "transport" layer of the application, handling the low-level connection and transferring the messages from the server to client and vice versa. Above this layer, there is a "real communication protocol" definition, which would be XMPP in our case. It does not make any sense to divide the protocols according to the OSI Model (ISO/IEC 7498-1) since everything above HTTP (including HTTP itself) takes place at application layer (number 7). On Figure 4.1, the communication between layers is depicted for the case of an application initializing Jingle multimedia stream (XEP extension) over XMPP, using BOSH mechanism sending and receiving the messages via HTTP. All of the protocols have been described (or at least mentioned previously). XMLHTTPRequest is rather API than protocol, its usage should be, however, displayed as well.

It is nor simple neither easy to take care about such a broad range of protocols in the application, handling the communication correctly at all levels. Fortunately, there are several existing libraries serving the lower level protocols. Specifically, most of them would conceal HTTP requests as well as BOSH and simplifying the XMPP layer. One of the JavaScript libraries, used when implementing Talker application, called Strophe, is described in this chapter. There are also other libraries for creating XMPP client in web browser, for example JSJaC. However, JSJaC is not that popular (and thus verified) as Strophe.js and has not been used in Talker application. It is mentioned just as another option. It would be also possible to handle whole XMPP/BOSH/HTTP stack manually and parse XML with

All along the thesis, we assume the application runs in the web browser. After all, the thesis title contains it. Therefore, mostly JavaScript libraries and tools for building a client application within a web browser are be mentioned in this chapter. Yet, it is certainly possible to implement XMPP client at the server side or as an independent desktop appliacation. In the end of this chapter, some server-side implementations are also mentioned due to the security reasons. See Section 4.3 for details.

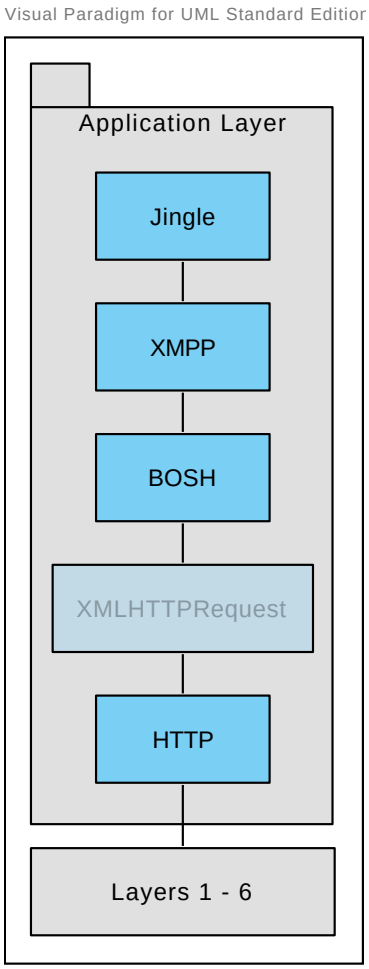

Figure 4.1: Complexity of the communication protocols at application layer

4.1 Strophe.js - JavaScript XMPP library

Strophe.js[1] is complex but simple to use JavaScript library for creating XMPP client in web browser application, initially created and further maintained by Jack Moffitt. Since XMPP is the protocol powering the Talker application, Strophe.js appeared to be a proper tool for creating it. In fact, Strophe handles whole "transport" layer for the application, as it was described in the introduction to this chapter.

Strophe.js is uses BOSH protocol (based on HTTP long polling) for sending messages and receiving the updates from the server. There is an unofficial project fork using WebSockets but it has not been accepted by the community yet. The main reason is probably unfinished standard for handling XMPP stanzas in WebSocket stream, currently available as a draft.

1. Strophe.js is sometimes refered only as "Strophe". However, Strophe is, strictly speaking, a name of the collection of libraries, from which only one is Strophe.js, the JavaScript tool described in this chapter.

[70]

In the following sections, the essential topics concerning Strophe.js are mentioned. It mostly comprises Strophe.js basic usage and philosophy, including using the plugins for getting additional functionality beyond the library itself.

4.1.1 New connection

Since XMPP client is intended to communicate with various nodes in the Internet, the connection represents the essence of the application. Strophe connects to the BOSH server through which it communicates with the rest of the XMPP world. Its URL is stated as BOSH_SERVICE variable in the following example. Setting up the connection is simple:

```
var BOSH_SERVICE = 'https://bind.jappix.com/',
    jid  = 'pavel@example.com',
    pass = 'mysecretpassword',
    onConnect = function(status) { /* implmementation */ };
var connection = new Strophe.Connection(BOSH_SERVICE);
connection.connect(jid, pass, onConnect);
```

Apart from the BOSH service URL, Jabber ID and password is passed to the connection method.

The last argument stands for the callback function which is triggered every time the connection status is changed. The status is passed as an argument to the function. It must take one of Strophe.Status enum values, for example `Strophe.Status.CONNECTED` when the connection is established. Example onConnect callback can look like this:

```
onConnect = function(status) {
  if (status === Strophe.Status.CONNECTED) {
    connected();
  } else {
    console.log('Connecting status: ' + status);
  }
};
```

When the connection is established, `connected` function is invoked. Usually, this is the place the event handlers are attached to the connection (passed as `this` in the context of onConnect function). Event handlers are described in Section 4.1.3.

4.1.2 Attaching to an existing connection

There is one problem with the process mentioned above. Generally, passing confidential data to JavaScript hard-coded in the rendered HTML files (i.e. within `<script/>` tag) constitutes a security issue. As you may have noticed, the previous example incorporates a password in the JavaScript snippet, which is exactly this kind of problem. Firstly, the page can be stored in plain text as a browser cache. Secondly, the web page is available for anyone

within the browser history tool so the password can be easily tracked down. Last but not least, unless TLS is used, the web page traverses Internet unencrypted. [5]

Luckily, Strophe.js comes with the possibility to attach to the existing connection. It basically means the initial BOSH handshake takes place at the server side. A connection is established and the server-side based XMPP client comes up with the session identifier and current request number (SID and RID, mentioned in Section 3.2). Afterwards, RID and SID are sent to the client (either within the initial HTML template or as a separate JSON response). RID and SID represent tokens which are secure enough to identify the client. However, unlike password, in case of being disclosed to the attacker, only a single session is compromised. [69]

Attaching to an existing connection in the JavaScript application using Strophe.js is quite simple, similar to initializing a new connection: [69]

```
var connection = new Strophe.Connection(BOSH_SERVICE);
connection.attach(jid, sid, rid, onConnect);
```

4.1.3 Event handling

Both JavaScript and XMPP are typical fields of using asynchronous processing of incoming events. The idea is simple. Generally, we specify the actions we are interested in and attach the event handlers to each of them. Event handler, function or method, is invoked as soon as the specified action occurs. The event handlers, defered functions, are called callbacks. Not only Strophe but all other asynchronous libraries (event-based or request/response) in JavaScript are based on callbacks, for example popular jQuery AJAX functions.[2]

When the idea of callback is clear, applying them to Strophe.js event handling is simple, see the example below. The first parameter is the callback itself (its name or anonymous function implemented on site). The "areas of concern" are specified as the remaining five parameters. The example callback here triggers every time the chat message comes, i.e. `message` stanza with attribute `type='chat'`. The parameters specified as null means that any value of that place is processed by the handler. So, the message sender, which can be specified by the next-to-last argument, can be anyone:

```
connection.addHandler(onChatMessage, null, 'message', 'chat', null,  null);
```

The callback function receives the incoming stanza object in Strophe-specific pre-parsed format. Then, the object can be queried (with jQuery or plain JavaScript selectors), such as at the example below:

```
onChatMessage = function(msg) {
  var messageBody = msg.getElementsByTagName('body');
  console.log("Message text: " + messageBody);
  return true;
}
```

2. http://api.jquery.com/jQuery.ajax/

The last line in the function is common pattern in Strophe.js the developer must be aware of. Each handler must return `true` (or any other value which evaluates to `true`) to stay bound to the event. If `false` is returned, the handler is unattached and it is not triggered anymore. [69] So `false` return values are proper for disposable callbacks, used only once.

4.1.4 Stanza builders

Whole XMPP protocol is represented by XML elements called stanzas, transferred between a client and a server. There are various approaches to create XML document structure in JavaScript. Using jQuery is the most common way to do so. jQuery object, created by jQuery `$()` function, represents XML structure tree which can be extended using methods like `append()`, `wrap()` and `after()`. [29]

Strophe.js comes up with completely independent XML creator, however inspired by jQuery. [69] The functionality is to be found within `Strophe.Builder` object. It allows the library user to create XML elements such as follows: [50]

```
$iq({to: 'tomas', from: 'pavel', type: 'get', id: '1'})
    .c('query', {xmlns: 'strophe:example'})
    .c('example')
    .toString()
```

The previous example code creates IQ stanza and the following snippet represents a result:

```
<iq to='tomas' from='pavel' type='get' id='1'>
  <query xmlns='strophe:example'>
    <example/>
  </query>
</iq>
```

Bijection between the two pieces of code is obvious. The only catch could be not yet defined `$iq()` function. Similarly as jQuery, Strophe.js provides shorthand functions `$msg()`, `$pres()`, and `$iq()` for creating all three kinds of stanzas, along with `$build()` function for making any kind of element. Therefore, creating presence stanza and sending it using Strophe.js can look as simple as the following example:

```
this.send($pres().tree());
```

4.1.5 Logger

Strophe.js does not conceal information about its operation. It contains several logger methods, divided by log level, mostly used by the library itself (it however can be used by the user code as well). Logging functionality is inserted in the main Strophe object itself. Calling the logger from the library methods is thus simplier, however it can hardly be replaced by another logger (all at once) defined by the user. All logger methods point to the ultimate log() method, which just returns by default. The intention of the library authors is providing this method to be overridden by custom user code. So is done in the Talker application, as described in Chapter 5, "Talker - IM client in web browser".

4.2 Strophe plugins

XMPP has been designed to be extensible (even by the protocol name) by various XEP extensions. Strophe.js follows the same direction and provides a simple way for the plugins to be added. And fortunately, there is not just the possibility to create the plugin, there are plenty existing extensions available publicly at GitHub. In this section, general way how to use the pulgins is described as well as brief overview of several plugins used in Talker application.

Each Strophe.js plugin is contained in separate JavaScript file, included to the application which uses it. The common name convention for plugin file is `strophe.myplugin.js`. [69] Whole plugin functionality is packed to JavaScript object and passed to Strophe as follows:

```
Strophe.addConnectionPlugin('myplugin',
{
  // plugin methods and properties

  init: function(connection) {
    // init body
  },

  exampleMethod: function() { /* ... */ }
});
```

Among all plugin object properties and methods, `init()` is explicitly quoted. That's because `init()` method is run as the plugin setup automatically by Strophe, after the new connection object has been created (which is passed to the method as the only parameter). The plugin can save the connection object for later use. [69] When the plugin is initialized, any method (such as exampleMethod() from the example above) can be invoked using the plugin name property of the connection object, as follows:

```
connection.myplugin.exampleMethod();
```

In the Talker application, several Strophe plugins are used. Particularly, the application uses Roster plugin, which provides easier handling of the user contact list (XMPP Roster). The basic use cases comprise retrieving the roster from the server by calling `connection.roster.get(o` usually as soon as the Strophe connection is established. The parameter represents callback function invoked when the server responds and the contact list is fetched. Of course, the contact list is no static structure and it gets changed as the contacts log in and out. All incoming changes are captured by the plugin and processed by the handler method, if it has been set up. For example, the following line of code sets up `presenceListener` function as a callback which is triggered any time the contact list is changed. The structure of method parameters is exactly the same as in case of ordinary Strophe handlers.

```
connection.roster.registerCallback(presenceListener, null, 'presence', null, null,  nu
```

4.3 Server-side implementations

Connection attachment, as described in Section 4.1.2, is used in Talker application. Apart from the security advantages and the possibility of boosting performance by pre-creating the connections, there are also some drawbacks in this approach. The most important is a need of an XMPP client (or a connection manager) implemented in the server-side language, alternatively running Strophe itself in the server-side JavaScript environment.

There are several available solutions. For example, Python XMPP connection manager called Punjab[3] is well-suited for "pre-creating" BOSH connections. Speaking of PHP (as it is the language Celebrio is powered by), there is JAXL[4] library which has been used in this project. Anyway, it is not very steady tool. I have found two bugs in this library, not mentioning poor documentation. One minor bug has already been accepted by the library author to the official branch,[5] the other is still being opened by the time of finishing this thesis.[6]Although the latter might not be a real issue, it still prevents from successful BOSH connection establishment.

3. https://github.com/twonds/punjab
4. https://github.com/abhinavsingh/JAXL
5. https://github.com/abhinavsingh/JAXL/pull/30
6. https://github.com/abhinavsingh/JAXL/issues/32

Chapter 5

Talker - IM client in web browser

As people use computers and the Internet more and more, they do not avoid interaction with others. On the contrary, communication and sharing represents essential part of current web systems. The social networks provide mostly the way to communicate with other people and they are used by millions. The same rules are valid for Celebrio, smart and simple application imitating the interface of operating system. Therefore, application Talker has been designed and implemented to serve in Celebrio as real-time communicator or instant messenger client.

In this chapter, Talker application is described in depth. There are the requirements listed, both functional and non-functional. Further on, the used technologies are mentioned and compared to possible substitutions. Some of the technologies, are described in individual dedicated chapters, such as Strophe.js in Chapter 4, "JavaScript XMPP client" so that only the parts specific to Talker are mentioned. Ultimately, several sections are dedicated to the implementation and testing. Source codes of the application are packed as the thesis supplement, they are however unlikely to run without Celebrio which was the application built for.

5.1 Analysis

Firstly, this section describes the application analysis, i.e. what the requirements are. It includes both the non-functional requirements (such as the platform the application must run or the system it has to cooperate with) and the probable usage by the customers. After the requirements are clear, it must be decided how to implement the desired functionality. That is where design takes place. Application design springs from the analysis and it just further decomposes and clarifies the way it will be implemented. The design part is crucial because the sooner possible glitches are unveiled and fixed the lower the cost is. The last thing which has to be considered and decided is the "stack" of proper technologies to be used, with regard to the both non-functional requirements and the design complexity.

5.1.1 Value proposition

The first step when creating a successful application (or even a whole ecosystem of application such as Celebrio is) should start by considering the added customer (or more precisely user) value. Value proposition mainly contemplates the possible market, the users and the

benefits they would have from using our application, i.e. Talker. [51] Our target market is well-defined and quite easy to interview: the elderly people. More than a half of the elderly use (or are willing to use) the real-time communication tool, comprising both text chat and video calls. They prefer it to be interoperable so they can speak to the people not using directly the same software tool as well.

Talker offers the benefit of running within Celebrio, the "operating system" designed specifically for the elderly. The compatibility is limitless within XMPP network. Unlike many communication tools, it provides text chat and video calling bundled together so the user just uses it as a "dialogue" application, no matter which way she feels like communicating at the moment.

5.1.2 Use cases

The use case analysis springs from two information sources. The first one is based on author's own opinion and experience with online communication tools. The other part is supported by user interviews and market research. In other words, the potential customers said what their probable use cases would look like. However, there is no quantitative data analysis on this topic.

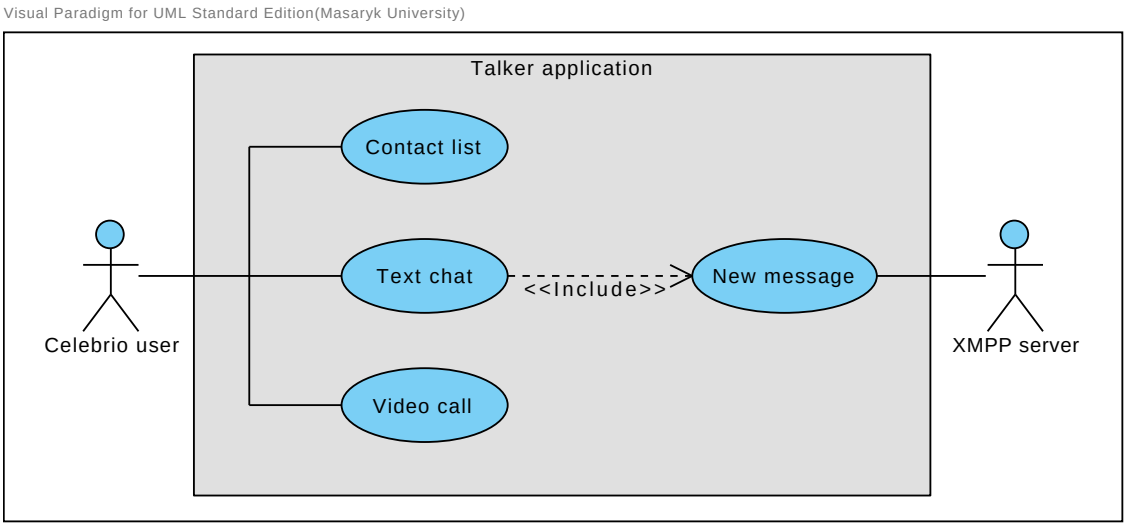

Figure 5.1: Talker use case diagram

As displayed at Figure 5.1, Talker use case analysis is quite simple. There are two actors operating the system, Celebrio user and remote XMPP server. The former uses all GUI application components, i.e. she can display the contact list with current contact present statuses, enter the text chat (which includes sending and receiving new messages) and set up a video call. A contact list, the first application screen, is further described in Section 5.7. When any of the contact is clicked at, communication view is displayed so the user can read the incoming messages, send a new message or start a video call. Several application screenshots are

pictured in Appendix A.

There is one more actor displayed at Figure 5.1. It is remote XMPP server, communicating with the application indirectly through established connection. When a new message appears at the server, it is pushed to the application and handled it as an incoming message. Every new message, received or sent, is appended to the "chat history", as usual in similar IM clients.

5.1.3 Choosing technologies

The use cases and value proposition imply some technologies and tools that are obvious to be used and also some constraints. At the lowest level, Talker is part of web application and it must run in the web browser environment. With respect to interactivity and real-time behaviour, it must be powered by *JavaScript*. There are other possibilities such as Adobe Flash or browser plugins, they however do not play well with various environments.

XMPP has been chosen as the communication protocol for sending the messages. It is opened, mature and interoperable technology, facilitating just plugging in the new part of world-wide network. XMPP is described in Chapter 3, "Extensible Messaging and Presence Protocol" and JavaScript XMPP framework in Chapter 4, "JavaScript XMPP client". Such nontrivial application, as Talker is, deserves fine client-side framework so the connections, states, contact lists and all messages are not only stored in DOM but also managed by JavaScript itself. In the last years, there are a lot of MVC[1] JavaScript frameworks. When implementing the application, I have tried Backbone, Angular, Knockout and Ember - all modern MVC JavaScript frameworks. After short period of testing, I chose Ember as the best one. Framework overview, various advantages and also some pitfalls are mentioned in Section 5.3.

The last important field to be covered are the video calls. On no account did I consider to implement it from scratch. Anyway, there are several handy tools and frameworks which could have been used. After all, I decided to use OpenTok, video call library described in Section 2.5.2, mostly for its wide compatibility. It even supports WebRTC in its newest version, which is considered to be the future of media applications on web.

5.2 Application architecture

As mentioned in the previous section, Talker is not trivial application which could be served by several inline scripts manipulating the DOM. It has to handle states, keep track of all simultaneous chats and dispatch all messages to appropriate contacts. Those reasons led to semi-modular architecture, pictured at Figure 5.2. Modules inside the grey package represent those I directly implemented, the outside ones are the used libraries and third party

1. MVC stands for Model - View - Controller, common three-tier architecture. Model is responsible for data layer, View stands for presentation layer. The function of controller varies from one implementation to another, but it is mostly responsible for dispatching the user actions and controlling application flow, for example redirecting.

modules, however necessary to run the application.

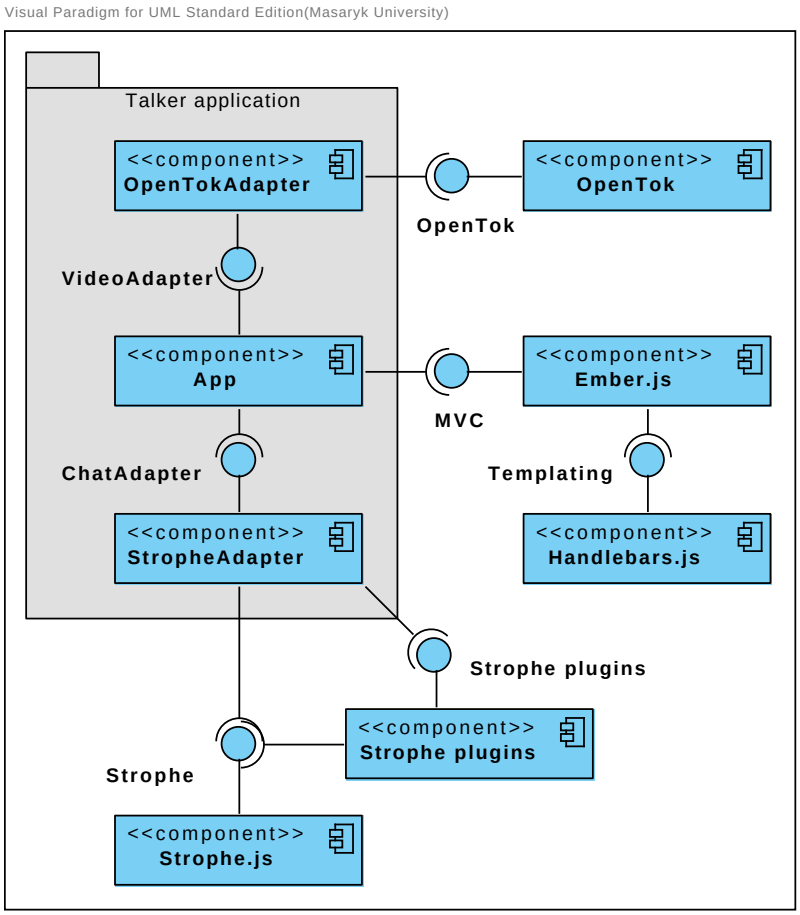

Figure 5.2: Application modules structure

The main application logic is placed inside `App` module. It is implemented as `Ember.Application` object and it serves as namespace for the rest of the application. Using namespace to wrap the whole application is good practice so the global namespace (i.e. window object) is not polluted. Within the namespace, other Ember-based classes are separated into four parts according to the MVC structure, i.e. models, views and controllers. The last part is router, responsible for handling the application state.

5.2.1 Models

Talker is supposed to handle data: keep the information about the contact list, update it as the contacts sign in and out, keep track of the message lists corresponding to individual contacts. All mentioned data manipulation is achieved with models. Model hierarchy is displayed at Figure 5.3. The main two entities, `Contact` and `Message`, are quite self-

explaining. It should be mentioned that messages are stored directly as a property list of each contact. Potential contact removal would erase its messages as well. Class[2] `Person` encapsulates the properties common for the contacts and the user operating the application. In other words, information about the current user, logged in to the application, is stored in an instance of that class.

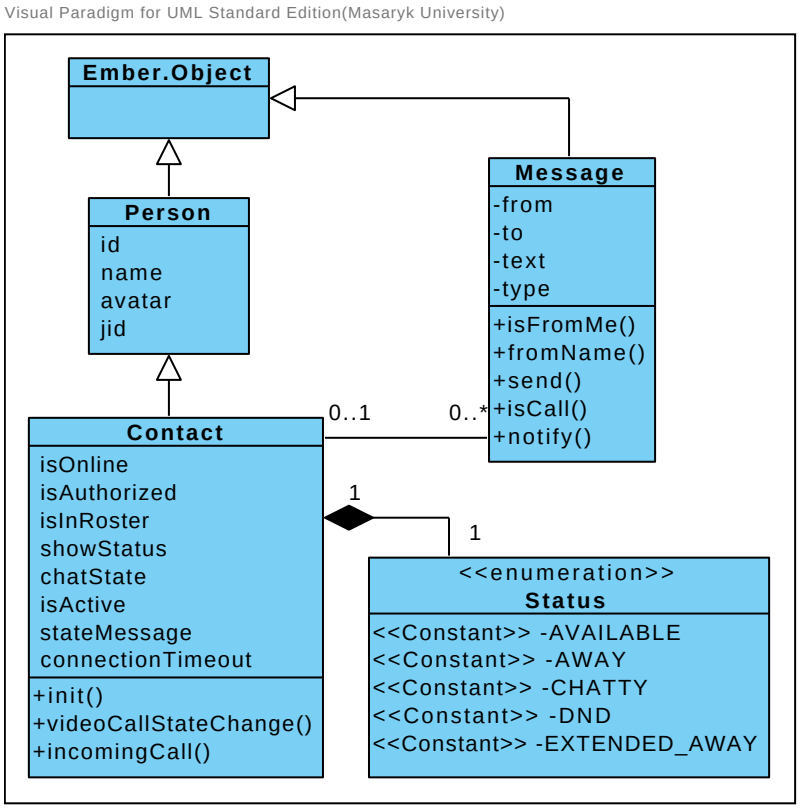

Figure 5.3: Structure of application models

5.2.2 Views and controllers

All Ember-related problematics is described in section Section 5.3, only the structure is mentioned here. As the user traverses the application, various controllers manage the states. The structure of controllers bijectively corresponds to the views hierarchy (i.e. almost every view has its respective controller and vice versa). Only the former is displayed, see Figure 5.4. The main view, contact list, is handled by `ContactsController` instance. When the user enters the conversation, ConversationController takes over the command. TextController and

2. When speaking about classes and displaying them at the class diagrams, one should be aware of the fact there are no real language-based classes in JavaScript. However, Ember provides a convenient way to imitate the class behaviour with `Ember.Object.extend` method, further described in the next section.

VideoController (with their respective views) are present within the conversation. Calling one or another depends on whether the user interacts with the text part (chat) or the video call.

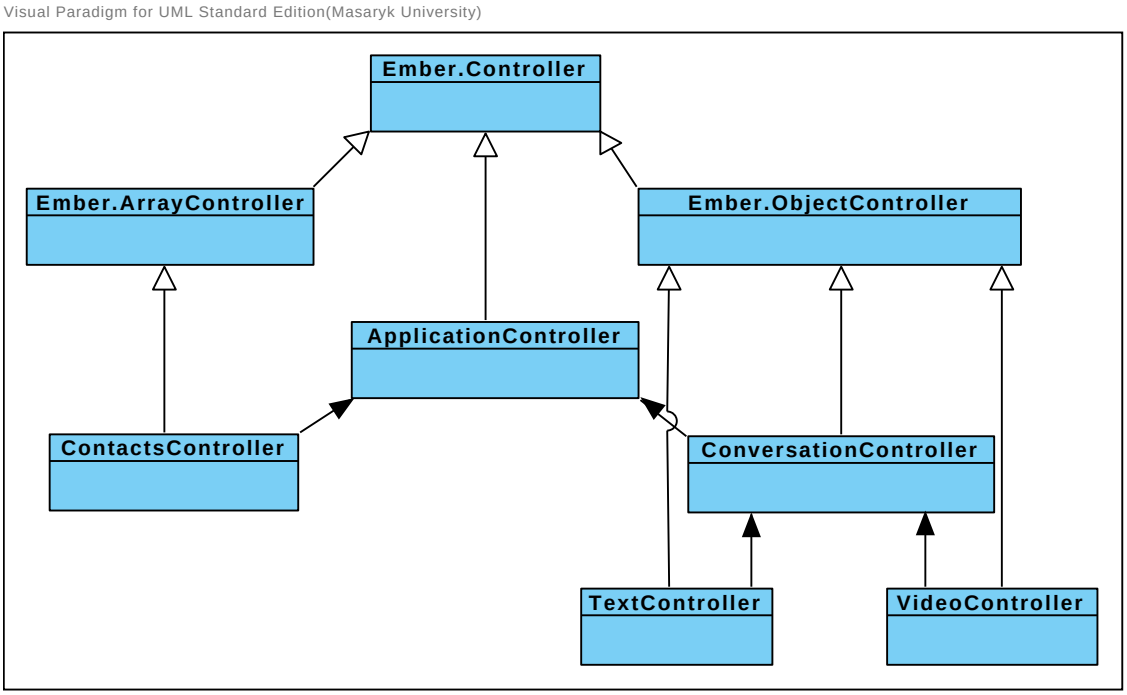

Figure 5.4: Application views/controllers structure

5.2.3 Adapters

There is a lot of application logic not related to the data itself, but comprising connection establishing, managing and triggering correct handlers when any new data comes from the server. As Jon Cairns fittingly notes, models should not contain the unnecessary application logic not directly related to them. [17] This pieces of code are divided into two "adapter" classes. `StropheAdapter` is responsible for creating, keeping up and reacting to the XMPP connection with BOSH server. In other words, it mostly handles the text messages. `OpenTokAdapter` uses OpenTok library to keep track of the video calls. Video adapter object is assigned to every contact because it keeps the state information (session) which is different for each contact. On the other hand, StropheAdapter is created and initialized just once for whole application since it does not contain any contact-specific data. The class structure is displayed at Figure 5.5.

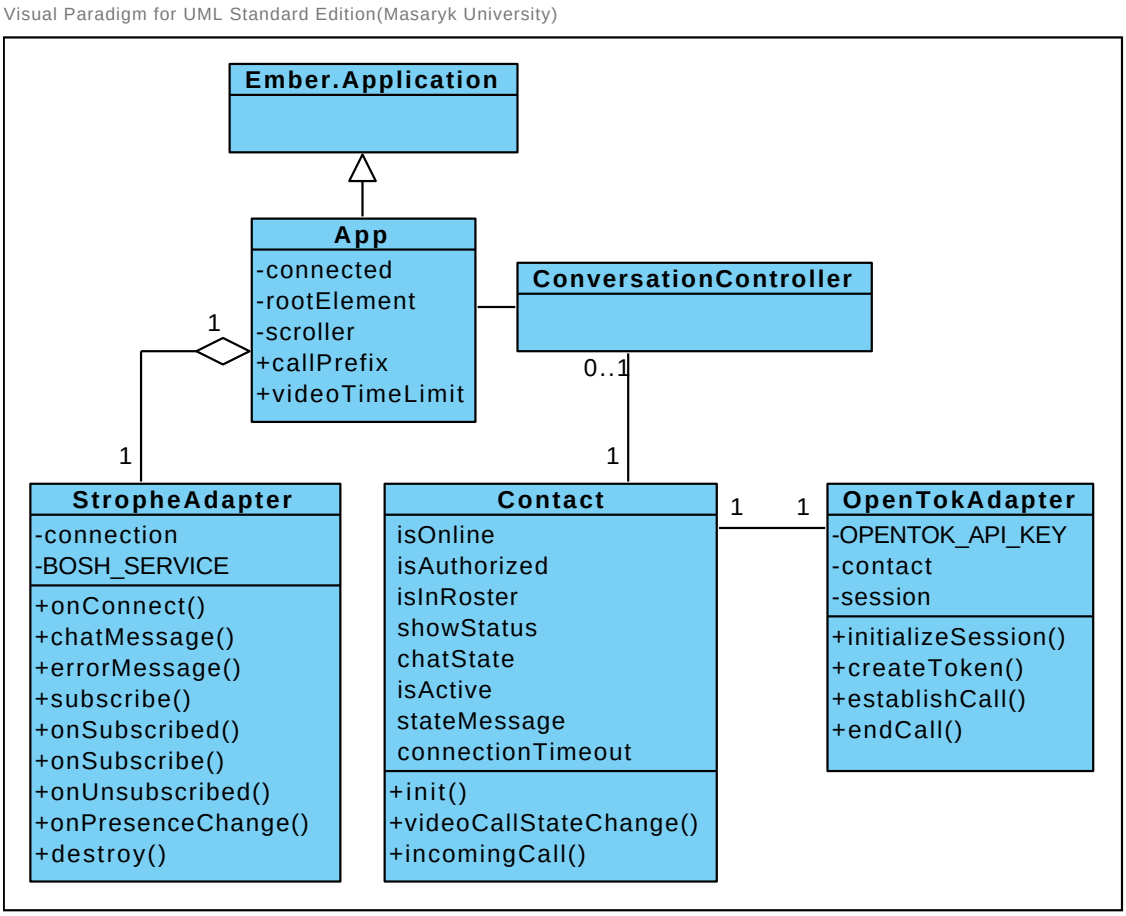

Figure 5.5: Connection of the application and adapters

5.3 Ember.js - JavaScript MVC framework

Talker is purely client-side-based application, implemented in JavaScript and running in web browser. As it is mentioned in the technology overview in Section 5.1.3, Talker implements MVC structure on the client side, in the form of Ember MVC objects. In this section, JavaScript MVC is analysed overally at first, then some features specific to Ember which were used in the implementation are described.

5.3.1 Client-side MVC

First of all, it should be clarified why it is necessary to do MVC in JavaScript. Most of the web technologies running on the server and powering web applications use some kind of MVC, accross all programming languages. It is Rails for Ruby; Zend, Nette or Symphony for PHP; Django for Python; Spring MVC or simple servlets with JSPs in Java; ASP.NET MVC for C#/.NET, etc. It appears that three-tier architecture helps the developers to organize the

code and the frameworks can do a lot of job automatically.

In the years, the approach to web application development changes a bit. As the clients performance increases, interactivity and quick response becomes the most desired benchmark. And here comes JavaScript, which can manage whole application at the client side, only synchronizing data with a server using AJAX requests. It seems this trend of *thick client* will be stronger and stronger in the next years and the server is going to be transformed only to the proxy in front of database system. [72][34]

It is obvious that many tasks previously a server was responsible for are moved to the client, to JavaScript. Above all, JavaScript application handles user actions (such as clicking a button or even typing to an input), rendering HTML snippets (templates) and storing the "temporary" data. Storing the data has become more tricky, because the JavaScript application works as a first level cache which processes all changes but synchronizes with the server only occasionally. Simple example: let there be an input for the user name, bound to the JavaScript model object representing a user. Whenever a new letter appears in the input, JavaScript updates its model object (still on the client side). No sooner has whole username been entered (either submitted or after a delay) when JavaScript synchronizes it with the server. Whole process schema is displayed at Figure 5.6.

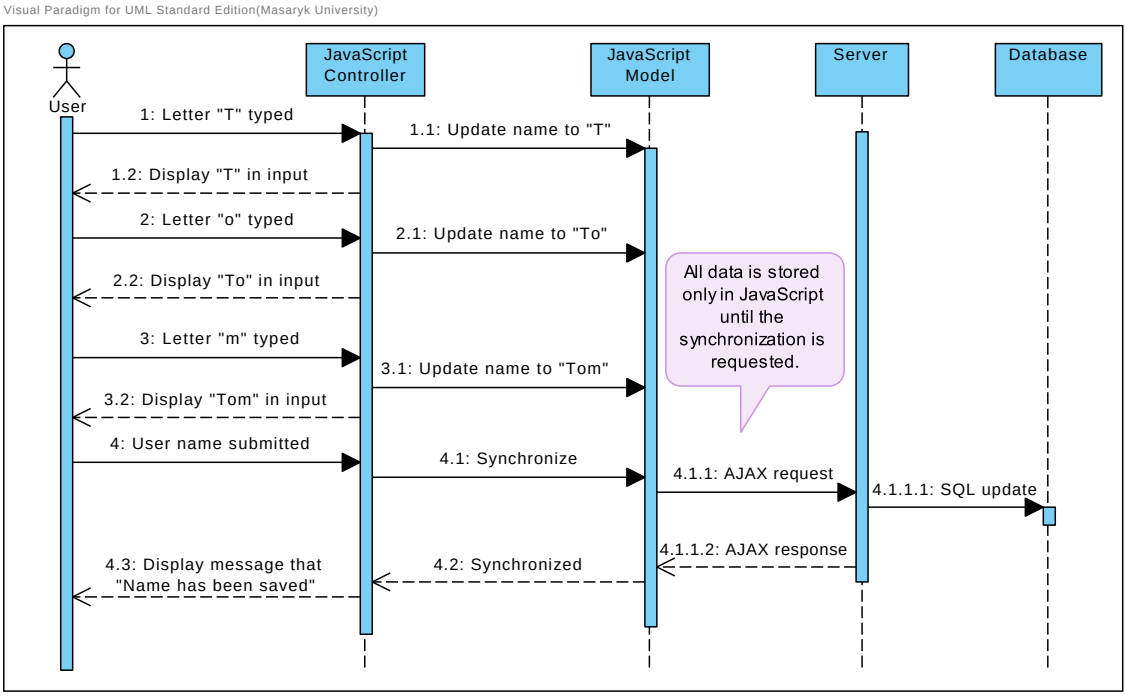

Figure 5.6: Example client-handled input, bound to JavaScript model

The main advantage of such approach is definitely the swiftness. Input changes numbered 1, 2 and 3 are processed immediately at the client. If the browser ran in the offline mode, submitting would store the data to the local storage eventually. Other very important

benefit, especially for the application developer, is separation of concerns between individual tiers. The result is much better organized and thus better maintainable than monolithic blend of code containing all aspects of an application at one place.

In modern client-side JavaScript MVC frameworks, JavaScript is also responsible for rendering HTML. This topic is set aside to separate section in Section 5.3.4.

5.3.2 Comparsion to other frameworks

Ember.js is not the only available JavaScript MVC framework. In fact, there are plenty of similar frameworks, each of them providing slightly different set of tools and asserting slightly different philosophy. I have chosen Ember for Talker for several reasons. The facts in this section may be a little subjective as they are mostly just opinions of mine, as author of this thesis and Talker application. There is actually not much comparison but more the features specific for Ember are described and explained.

First of all, Ember provides automatic bindings. There are automatic bindings between object properties, so that all computed properties are automatically updated when the "raw" properties, which they are based on, change their values. [15] The following snippet of code from Talker `Message` object represents such situation: `fromName` attribute represents the contact name and it is based on the contact JID, stored in `from` attribute:

```
App.Message = Ember.Object.extend({
    from : null,
    fromName : function() {
        /* implementation */
    }.property('from')
});
```

Bindings are really valuable when `fromName` property value has already been rendered in the template. There is no `render()` function (as it is in Backbone, for instance) to render the template; templates are rendered automatically and updated automatically as well, as the underlying data changes.

There is a lot of additional work Ember does. It follows the "convention over configuration" rule to the maximum extent, above all other MVC frameworks. For example, when one keeps to the official naming conventions, the framework creates the objects which are necessary and has not been defined, such as missing controller.

Ember.js uses templating engine Handlebars, which provides very clean and convenient way to create presentation layer of the application. In my opinion, using dedicated templating engine makes the code much more cleaner than using HTML attributes, as for example Knockout.js does. Templating is described in detail in Section 5.3.4.

To sum it up, Ember.js is modern and powerful tool doing a lot of job that programmer does not have to take care about. On the other hand, the learning curve of Ember is told to be one of the steepest and it is quite difficult to make the *second* step, after reading the basic tutorials and examples. Fortunatelly, there is quite active community working on screencasts[3],

3. <http://emberwatch.com/>

documentation and actively answering question on Twitter and Stackoverflow.

5.3.3 Controllers

Application controllers are responsible for storing the current state and mediating the communication between the views and models. [14] In Ember (and thus in Talker as well) controllers also represent collections of model objects.

The first function, processing the user actions from the view, is represented by `processMessage` method in `TextController`. The following snippet is shortened part from Talker source code:

```
App.TextController = Ember.ObjectController.extend({

    processMessage : function(messageText) {

        var message = App.Message.create({ ... });

        message.send();
        this.get('content.messages').pushObject(message);
    }

});
```

The method `processMessage` is called from the view object when the message input is submitted. The view just grabs the data from input and sends it to the controller, which creates new `Message` object, sends the message and finally updates the list of messages for the current contact, accessible via `this.get('content')`. Adding `.messages` to the "selector" points the getter directly to the array of messages.[4]

Another example of controller from Talker is not `ObjectController` as in previous example, but `ArrayController`. It represents the collection of contacts, displayed in the main application view:

```
App.contactsController = Ember.ArrayController.create({

    content:[],

    pair : function () {
        var content = this.get('content');
        var result = [];
        for (ii = 0; ii < content.length; ii += 2) {
            result.pushObject({
                "first" : content[ii],
                "second" : content[ii + 1] ? content[ii + 1] : null
            });
```

4. Acutally, the method contains a lot more code such as argument check and return values. They have been omitted in this example for clarity and shortness.

```
        }
        return result;
    }.property('content.@each')
});
```

The contacts are stored in the `content` attribute, which is Ember convention for naming the actual data set within the `ArrayController`. As it is necessary to render the contacts as pairs in the two-row-view, there is `pair` method returning the content reformated as the array of pairs. Notice that the function is property of `content.@each`, which means that any update, insert or delete of the content array forces to recalculate pairs as well.

5.3.4 Rendering HTML

Thick JavaScript client is responsible not only for manipulating the data but also for rendering it to the HTML page. Ember is shipped with Handlebars - a lightweight templating engine. The application is decomposed to the separate views; in case of Talker it is a contact list, the conversation, which consists of another two parts - text chat and video. The structure basically corresponds with the controller tree at Figure 5.4. Additionally, text chat contains theoretically unlimited number of view, each of them for a single message, either sent or received.

There are two important parts of the view: the view object itself and its template. To describe the philosophy of presentation layer, `VideoView` is further examined here, as an example:

```
App.VideoView = Ember.View.extend({
    templateName: 'video',

    buttonLabel : function() {
        return this.get('controller.content.isActive') ? m.stop : m.start;
    }.property("controller.content.isActive")
});
```

There are two properties in the view object, `templateName` and `buttonLabel`. The former just determines the name of template connected to this view. The latter is common method returning one of `stop` or `start` messages, according to the value of property `isActive`, located in the `controller.content` object, pointing to the contact the user is currently talking to. The other important part is the template itself, inserted directly in HTML page:[5]

```
<script type="text/x-handlebars" data-template-name="video">
```

5. Handlebars templates must be compiled by Ember to JavaScript code to be ready to be used. It is possible to let Ember compile the templates in the browser or precompile the templates on the server. However, there is very poor support for doing so in PHP. Therefore, templates are rendered in HTML page and compiled in the browser while developing the application. For production, it is recommended to precompile them using dedicated Node.js module: <http://handlebarsjs.com/precompilation.html>.

```
    <div class="content_margin" >
        <div id="video"><!-- video of me comes here --></div>
        <div id="stream"><!-- video of the contact comes here --></div>
        <div id="message">{{stateMessage}}</div>
    </div>

    <div class="button_wrapper_video">
        <button {{action toggleCall target="controller"}} type="submit">
            {{view.buttonLabel}}
        </button>
    </div>

</script>
```

The template is inserted in script element, declaring itself as a handlebars template. The only attribute `data-template-name` identifies the snippet and connects it to the view object, which specified the same value in the `templateName` attribute, as shown above. The template can contain any HTML, along with the variables and expressions, both enclosed with the double braces, sometimes called moustaches. In the example of video template, there is `stateMessage` variable queried, which is looked up in the object bound to the template. The object is provided by router (described in Section 5.3.5). In this case, it would be the current contact the user is talking to.

The second handlebars expression in the template is action. Actions are usually bound to the active elements, such as buttons. When the button is clicked, the action is triggered. Ember finds the controller responsible for the current state, which would be an instance of `VideoController` in this case, and calls the method `toggleCall`, specified in the template expression. Actions can be placed both in the controllers the views, according to the each of whose nature.

Handlebars templates can transform almost any data stored in JavaScript to HTML code so the rendered template corresponds exactly to the application state. It is possible to add classes according to the values of the object properties. In the following snippet, there is message template, rendered individually for each message in the conversation. The class is assigned to the wrapping div element according to `isFromMe` property. If it is true, `from_me` class attribute is added, `to_me` otherwise. Class `message` is added independently of the properties and it is always present so the block can be styled as message with custom CSS.

```
<script type="text/x-handlebars" data-template-name="message">
    <div {{bindAttr class="isFromMe:from_me:to_me :message"}}>
        <span class="from">{{fromName}}:</span>
        <span class="text">{{text}}</span>
    </div>
</script>
```

The last two things to be mentioned about templates are iterating and nesting. Have a look at the following example from the main contact list view (altered a bit):

```
<script type="text/x-handlebars" data-template-name="contact-list">
    {{#each App.contactsController.pair}}
        <div class="pair_wrapper">
            {{view App.ContactView contentBinding="first"}}
            {{view App.ContactView contentBinding="second" }}
        </div>
    {{/each}}
</script>
```

When the template is rendered, Ember iterates over the collection of contacts, accessed with `pair` method (it was mentioned in Section 5.3.3). Then, framework takes each contact from the pair and instantiates `ContactView` for it, passing it as a current object for the view and rendering the template exactly in the place the view expression is stated. Therefore, the list of contacts is rendered, grouped by two inside `pair_wrapper` blocks.

5.3.5 Routing

Despite HTTP is stateless protocol, it is common habit to build stateful web applications. The states are simulated by various web pages and the transitions are carried out clicking on the links. All changes are simultaneously displayed in URL bar.

The problem is that JavaScript application cannot change the web page, leave one and continue at other. JavaScript would lost its context and whole MVC structure would serve to no purpose. Therefore, most of the JavaScript applications using MVC frameworks run at single web page, no matter how many states they contain. In Ember.js, there is special object called *Router*, responsible for transitions between states, dispatching of events and rendering correct templates.[6]

Router contains several states, ordered in a tree structure, according to the flow of possible user actions. The routes can be nested. The first aspect of router is mapping the states to URLs. Every time the transition takes place, router changes the context (basically renders different template) and it changes URL as well. Since JavaScript can change only URL fragment router does exactly so. And vice versa, when the user types specific URL in the URL bar, the application lets the router to choose the matching state and set the context in a similar way.

```
App.Router = Ember.Router.extend({

    root: Ember.Route.extend({

        goToContactList : Ember.State.transitionTo('contacts'),
        goToConversation : Ember.State.transitionTo('conversation.index'),

        index : Ember.Route.extend({
```

6. Ember router API changed a lot since the application was implemented (it is less than a year, but still). Further in this section, the old version of API is described since it is the version used in Talker application. Updating Talker to the newest Ember.js version with new router is one of the possible future tasks.

```
            route:'/',
            redirectsTo:"contacts"
        }),

        contacts : Ember.Route.extend({
            route:'/contacts'
        }),

        conversation : Ember.Route.extend({
            route:'/conversation/:contact_id'
            index : Ember.Route.extend({
                route:'/'
            })
        })
    })
});
```

Handling the application links is the second responsibility of router. In the example above, there are two link destinations, named by `goTo*` labels. When any element is marked with Handlebars `{{action goTo*}}`, it is transformed into a link leading to the corresponding router state. And the router handles the event by simple transitioning to one of existing states.

Let's have a look at one of the routes in more detail. As mentioned before, router is responsible for creating unique URL for every application state. That usually means serializing the application state to URL when the transition is carried out via link. In the other case, when the URL is typed into URL bar directly, it has to deserialize it and establish correct application state.

```
conversation : Ember.Route.extend({

    route : '/conversation/:contact_id',

    connectOutlets : function (router, contact) {
        // binds current contact to the conversationController
        // so the text/video view can access it
        router.get('applicationController')
              .connectOutlet('conversation', contact);
    },
    deserialize : function (router, params) {
        return App.contactsController.find(function(item) {
            return item.id == params.contact_id;
        });
    },
    serialize : function (router, context) {
        return context ? { contact_id : context.get('id') } : {};
    }
})
```

There are two methods providing such functionality, `serialize` and `deserialize`. The former picks the unique identifier for current context (contact object) and returns it as `contact_id`. This parameter is used in the route `'/conversation/:contact_id'` as a dynamic segment, so that actual route would be `'/conversation/42'` for example. Deserializing works in exactly opposite way. Ember prepares the URL segment for the router so it is passed as object containing the map of dynamic segments, which would be `{ contact_id : '42' }` in our (example) case. The only job `deserialize` must do is to find the appropriate object (contact with ID 42) and return it as the current context object.

The last remarkable aspect of router is choosing the appropriate templates and connecting them together. Basically, `connectOutlets` method tells main application controller to "include" the controller responsible for current state, which would be `ConversationController` instance in the example above. This is importand when rendering the templates. First, main application template is rendered. Then, Ember looks into the template for `{{outlet}}` tag and inserts the conversation template exactly to the place it is found. The last thing, appropriate contact object is passed as the context for the conversation controller (and thus the template as well). Everything described in this paragraph is deduced from one line implementation of `connectOutlets` method in the example above.

5.4 Initializing the connection

When the user clicks at Talker icon in the Celebrio main menu, a request is sent to the server which renders HTML page and sends it along with all necessary assets, such as JavaScript files and style sheets. When the page is loaded, Talker starts running. First of all, the connection to remote BOSH server must be established so that the application can communicate with the rest of the world. There are two options to do it: creating a brand new connection or using the existing one, i.e. attaching to the connection as it is described in Section 4.1.2.

As JavaScript files are loaded, `App` object is defined by Ember firstly - whole MVC structure described in Section 5.3. Then, new `StropheAdapter` object is created and it attempts to establish the connection with BOSH server immediately. Strophe adapter supports both creating a new connection and letting server to create one. The initialization code basically checks whether the password has been provided to the client code. If yes, BOSH authentication takes place, as displayed at Figure 5.7. When the password is not provided, JavaScript client sends an authentication request to the server, which should establish the connection and return JID, SID and RID so the client can attach to the connection and use it for further communication. The workflow is depicted at Figure 5.8. In both cases, the authentication handshake follows the SASL authentication protocol defined by XEP-0206. [67]

It is recommended not to send a password to the client, for obvious security reasons. Therefore, Talker is set to use server-side connection establishment by default. On the other hand, PHP implementations of XMPP clients are not that matured and certain problems appeared, as mentioned in Section 4.3.

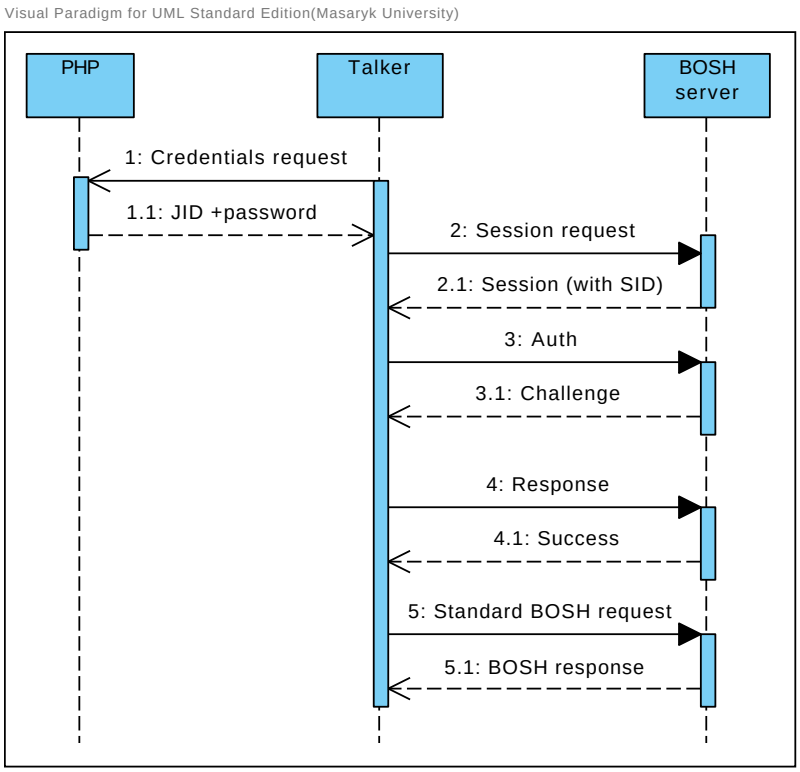

Figure 5.7: Establishing a new BOSH connection from JavaScript

5.5 Processing events

Talker works as a typical asynchronous application. The application mostly consists of many callbacks (event listeners) attached to different events and awaiting to be triggered within the event loop. [4] It is common pattern in JavaScript applications nowadays. [30] Talker does expect the events to come either from the user (traversing the application, sending a message, ...) or from the network (response from BOSH server, incoming message, ...). Basically, almost everything in Talker is intended to be either declarative (Ember object patterns) or event-based (Strophe events).

The first kind of events are those incoming from the network. As it was mentioned in Section 4.1.3, Strophe is set up with the event handlers attached to various event types. There is not even one synchronous call between the application and the server so that every response is handled with custom callback asynchronously. All callbacks are defined as methods of `StropheAdapter.prototype` object so that they are shared by all `StropheAdapter` instances. [31]

Incoming chat messages are handled by `chatMessage` method. The message is passed to the method from Strophe library in form of `Element` object. The method parses the message, constructs the Ember model `Message` object and tries to assign the message to a con-

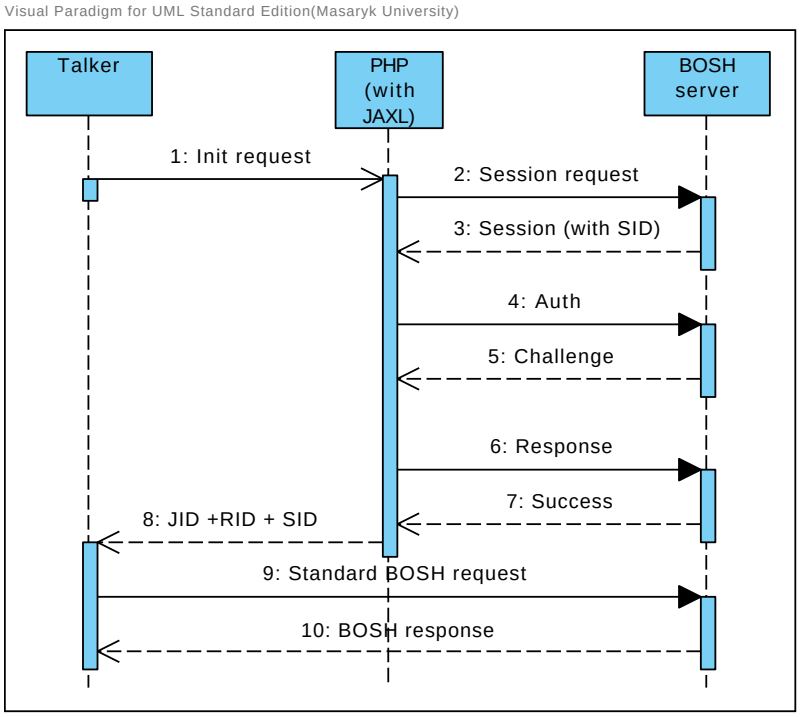

Figure 5.8: Establishing a new BOSH connection using server side XMPP client

tact (the recipient). When the recipient is know (so she is present in the the user contact list), new message is appended to the list of messages. Ember automatically updates the view so the message appears in the list immediately.

```
var message = App.Message.create({
    to : msg.getAttribute('to'),
    from : Strophe.getBareJidFromJid(msg.getAttribute('from')),
    type : msg.getAttribute('type')
});

var contact = App.contactsController.find(function(item) {
    return item.jid == message.from;
});

contact.get('messages').pushObject(message);
```

The user actions are handled mostly by Ember itself. For example sending a new message is managed by the controller (described in Section 5.3.3), which just calls adapter function which actually sends the prepared message object. It is one of the design drawbacks of the application, that the there is two-way awareness between the adapter and controllers (somethimes controller calls adapter, sometimes the adapter actively calls controller). This design was chosen for its simplicity and it appeared to be sufficient for the application of Talker's

size.

5.5.1 Notifications

Talker is only a part of bigger system - Celebrio. The user has to find out when a new message arrives. Therefore, simple mechanism of notifications has been developed and used accross the system, not only in Talker. When a new chat message arrives, the application triggers the built-in mechanism and the notification is propagated to the main Celebrio frame using `postMessage` API.[7] Since notifications are not specific only to Talker and they were not developed as a part of the application, they are not described further here.

5.6 Logger

Although JavaScript is very popular language, at least in web applications, there is not common way to log messages. There are two possible reasons. Firstly, JavaScript runs in the web browser and frequent AJAX requests would be necessary to log everything what happens. Secondly, web browser is kind of "hostile" environment, so the system developer cannot rely on the message truthfulness at all. Anyway, it is useful to have some information logged, at least in the development mode, locally to the browser console.[8]

The main problem with common frequently used `console.log` method is the impossibility of hiding certain kind of messages. In Talker, there are seven different kinds of logs: those related to messages, video, subscriptions, notifications etc. It is impossible to have the output uncluttered without constant commenting and uncommenting `console.log()` calls.

In order to solve this trouble, logging library Log4js has been used in the application. It allows to create various loggers (with various appenders). Each logger has its own log level set so it outputs only the messages tagged by that level, or higher. Actually, the logger usage is very similar to well known log4j, including formating layouts. There is AJAX appender available in the library so it is possible to send the messages to the server to be stored. This might be useful to log error messages not reproducible in the development environment.

5.6.1 Setup

The logger (with console appender) is set up as follows:

```
Log4js.getLogger("message")
      .setLevel(Log4js.Level.INFO)
      .setAppenders([new Log4js.BrowserConsoleAppender()]);
```

7. <https://developer.mozilla.org/en-US/docs/DOM/window.postMessage>
8. All current browsers have console.log method implemented, which outputs any message from JavaScript code to the browser console, accessible to anyone using the browser. The main advantage is the fact that console is hidden from the ordinary users, yet it can easily be accessed by anyone debugging the application or reporting a bug. See API for details: <https://developer.mozilla.org/en/docs/DOM/console.log>.

Originally, the library does not allow chaining the methods so a local variable must be created to set up the logger (or getLogger method must be called multiple times). I believe method chaining is sensible pattern so I reimplemented several methods in the library to support it. Unfortunatelly, the project is not hosted on GitHub so it is quite difficult to offer the changes back to the community.

5.6.2 Logging

Logging a message is also quite easy, it might look as the following example (cut out of `onConnect` method):

```
App.log.logger.info('Strophe is connected as ' + App.me.jid);
App.log.logger.debug(this);
```

Two notes should be mentioned about the code. Firstly, all logger which have been set up are accessible via `App.log` object. It is a wrapper so the used logging library can be changed (and it was changed several times during the development). As long as the logger objects implement common logging methods (such as `info` and `debug`), the application will work.

The second thing is logging complex objects such as `this`, pointing to the `Strophe.connection` in the example call. Log4js does not provide any layout which would not transform the value to string - [object Object] in our case. This is not very useful - the developer usually knows there is a object and he just wants the possibility to inspect it. Therefore, I enhanced the library again in order to log complex objects. First of all, the logger outputs a text message saying complex value is about to be logged. Then, the complex value is logged individually without being cast to string. Therefore, it is possible to access it interactively in the console, fold and unfold the nested properties and carry out other inspections. Logging object from the example above thus looks as follows (arrows within the object symbolize the property can be inspected):

```
INFO - Strophe is connected as pavel.smolka@jappix.com
DEBUG - [object Object]
Strophe.Connection {
 > adapter: StropheAdapter
 > addHandlers: Array[0]
 ...
}
```

5.7 Contact list

The list of contacts is essential part of every instant messaging application. The same applies to Talker, with little modification compared to classical IM clients. Talker has been designed and implemented as the part of Celebrio - comprehensive “operating system”. The contact list is not managed by Talker itself, it servers to other applications as well. On the other hand, the user can connect existing third party XMPP account to Celebrio, with its own contact list

(roster). It is crucial to deal with the contacts duplication and appropriate matching one to another.

5.7.1 Fetching Celebrio contact list

As mentioned in the previous paragraph, the contact list base comes from the server-side database. Celebrio asks the user for adding her contacts in People application, another part of the system. All contacts are stored in a relational database and available through the system, in all applications. One of the properties of the contact entity is JID. For the needs of Talker, the contacts are filtered on the server so that only those which have JID filled are returned as the user contact list.

Since Talker is JavaScript application, the contacts are passed to JavaScript in JSON format. To avoid an extra request (and thus a delay), contact list is rendered by PHP to the `<script/>` element in the server-side processed template. Then, when the Ember application is initialized, the array of plain JavaScript objects is transformed to the content of `ArrayController`, specifically `App.contactsController`. After that, the application can easily manage rendering the contact list within Handlebars template and keeping it up-to-date.

Ember class `Contact` is one of the most voluminous objects in the application. Every `Contact` object contains both the "static" properties from the server (such as contact name or avatar picture) and the dynamic information about current state in the application. The latter category includes the information whether the contact is online or not, its presence status (DND, Away, ...) and also all the messages related to the contact.

Within whole Celebrio system, there is a rule that the contacts the user has not added to her contact list (in the People application) does not affect the user in any way. In fact, all messages, updates or invites from unknown contacts are processed by the application, but they are thrown away and not displayed to the user. There is an easy possibility to extend the application to be able to communicate with the contacts that have not been added yet. However, it would break the system philosophy and making such decision is not up to the thesis author.

5.7.2 Matching the entries from XMPP roster

The contact list retrieved from Celebrio backend, as it was described in the previous section, is only underlying material for building the contact list. The other part is XMPP roster (contact list), stored at the XMPP server. The roster can possibly contain different contacts from internal Celebrio contact list, because it can be accessed from other XMPP clients as well.

When the application is connected, fetching a roster from XMPP server is one of the first steps. One of Strophe.js plugins (plugins were generally described in Section 4.2), called characteristically Roster, provides convenient API for this - a callback function is passed to `roster.get` method and executed when the roster is fetched.

The callback does basically two things, as displayed at Figure 5.9. The retrieved contacts

(those from XMPP roster) are matched to existing contacts retrieved from Celebrio server. Contacts present only in the roster are just logged, the application does not work with them. In other words, the appropriate record in `contactsController` array is found and match to every XMPP contact retrieved in the roster.

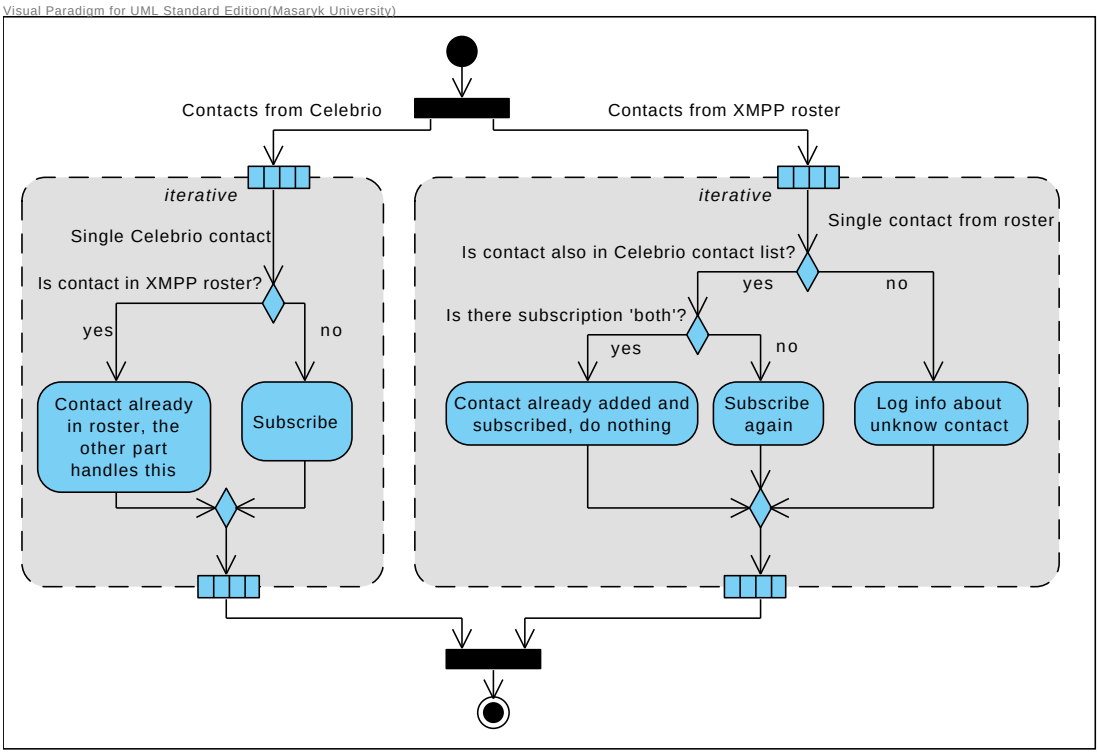

Figure 5.9: Matching XMPP roster and Celebrio contact list

The second part (left side of the picture) of matching contacts works from the opposite direction. Application processes all contacts that has been retrieved from the server and which are not in XMPP roster. Then, it sends subscription request, which consists of several steps, but it simply means adding the contact to the XMPP roster.

5.7.3 Presence and online statuses

One of the biggest advantages of real-time communication tools is the fact that one can see whether his contacts are currently available (i.e. online, logged in) or not. In terms of XMPP and Talker, we speak about *presence*. Strophe does not provide convenient way to work with presence but Roster plugin does.[9]

Presence is processed in way usual to Strophe: the developer provide a callback to a certain topic, which is triggered when anything related to that topic happens. The process of

9. It is the same Roster plugin which handles roster retrieval and processing, mentioned in the previous section.

binding the callback is described in Section 4.2. When the callback is bound, Strophe watches over the presence changes sent as presence stanzas from server and hands them over to the assigned callback - `onPresenceChange` method where the presence change is processed.

The callback function recieves two parameters. The first one contains the list of all contacts, the second parameter represents just the information about the contact whose presence has changed. This is a bit tricky because the callback is triggered even if only the presence of the currently logged user changes - in this case the second parameter (changed) is undefined.

The presence is processed in two ways. Firstly, there is basic telling online from offline status. The other part is optional online status specifying the availability (away, DND, ...), contained in `<show>` element of XMPP message. [37] The online status is processed as well and the current value is stored for each contact. Unfortunatelly, the graphic design for showing statuses has not been proposed yet so it it not displayed within the application, yet ready to be.

There is one more glitch about online statuses. XMPP makes it possible for one to be logged in simultaneously from several devices. In such case, each session (called resource in XMPP) should state its priority. By changing the priority appropriately, it is possible to keep the client running at home and yet receive the messages at work by being logged there in with higher priority. In case the contact is logged in at several places (i.e. several pieces of presence information are fetched from the server), Talker finds the resource with highest priority and processes its online status.

Another possible approach would be sorting the online statuses from the most desirable (i.e. online would be the first, then away, and so for) and looking for the "best" one, regardless of the priority. It could actually more correspond to the real state of matters because some clients (such as Google Talk client) set always the highest priority. However, I decided to keep the rules XMPP protocol sets.

5.7.4 Subscriptions

Subscribing in XMPP basically addressses adding the contacts to the contact list and asking them for the same. Due to the simplicity of the application, Talker performs subscriptions automatically. When the new contact (i.e. new contact added in Celebrio system) is found, the subscription request is sent when Talker starts and loads the contact list. In the opposite direction, when the subscription request is received, the application tries to find the appropriate contact and subscribe too. If no such contact is found, information is logged and nothing happens.

This approach has one big advantage: the user does not have to respond to the subscription requests and she does not need to send own requests either. Yet, there is also one disadvantage, basically the same thing from the opposite point of view. The user cannot fully control her contact list - for example refuse subscription request from the contact in roster. We decided for this trade off to make the application as simple as possible.

5.8 Video calling

TODO: Video framework, flash fallback

5.8.1 Using OpenTok library

TODO: OpenTok library, Native implementation in Android
TODO: WS in OpenTok

5.8.2 Implementation

TODO

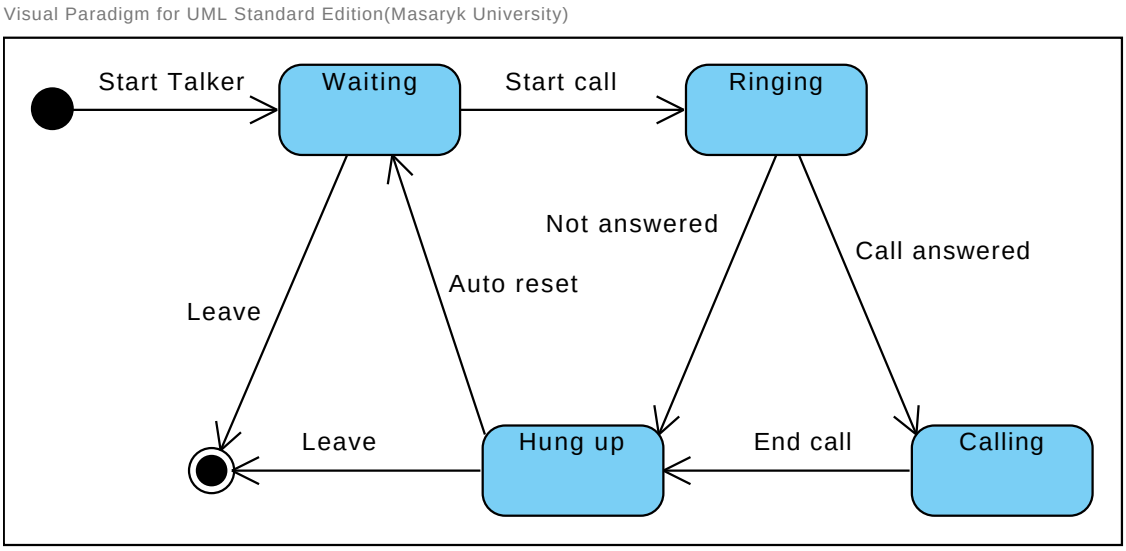

Figure 5.10: Video call states

5.9 Testing

Every application is tested during the development. Traversing the application manually is common way of testing, especially in case of web applications. It is definitely the easiest but also not repeatable and thus difficult to scale in large projects. Therefore, automatic tests are often used. There are three kinds of tests concerning web applications. The most widespread kind of tests are unit tests concerning the server-side code. Testing server-side code brings the benefit of predictable and easily modifiable runtime environment. However, Talker is client-side application and therefore server-side unit testing is not an option. The other two types of tests, both related to Talker, are JavaScript unit tests and Selenium tests, both described below.

JavaScript unit testing is relatively new technique, definitely younger than classic JUnit tests in Java or PHPUnit in PHP. Earlier, the JavaScript consisted of several action listeners bound to the HTML elements on web page. However, hardly did real application logic move to JavaScript when the application code grew bigger. Therefore, it must to be tested automatically.

There are several available solutions for unit testing in JavaScript. Some solutions cover whole testing "stack", such as Jasmine or QUnit. On the other hand, there are tools which separate the problem of testing (setting up the environment and running tests) from assertion, which is responsible for injecting the functions which help with checking whether the tested method behaves correctly and return correct values. I believe the separation of concerns is good pattern overall so I have used the tools from the second group: Mocha and Intern testing frameworks, both using Chai as the assertion library.

JavaScript is specific by not having strictly defined the runtime environment. Usually, client-side JavaScript runs in web browser interpreter. However, there are also many tools running with Node.js, JavaScript server-side platform and runtime environment. Since Talker is web-based application, it is crucial to have it tested in web browser environment. All three solutions described below perform the automatic tests with web browser environment.

5.9.1 Chai assertion library

Before the unit testing frameworks can be described, another part of testing must be mentioned. It is assertion, i.e. checking whether the code resulted to the expected values. Both Intern and Mocha, testing frameworks described later on, use assertion library Chai. It provides convenient and easily readable way to check various conditions.

There are two assertion styles which can be used with Chai: behavior-driven development (BDD style) and test-driven (TDD). They differ only by syntax, their expression power is equal. TDD style looks more similar to JUnit or PHPUnit: [10]

```
assert.include([ 1, 2, 3 ], 3, 'array contains value');
assert.deepProperty({ tea: { green: 'matcha' }}, 'tea.green', 'matcha');
```

On the other hand, BDD provides easy-to-read word chaining, resulting to a sentence. The following example (BDD) equals the previous one (TDD), only the used style differs: [9]

```
expect([1,2,3]).to.include(3);
expect({ tea: { green: 'matcha' }})
    .to.have.deep.property('green.tea', 'matcha');
```

5.9.2 Unit testing with Intern framework

Intern is a tool for creating, running and managing JavaScript unit tests, all with minimal need to set up another third party tools and environments. Intern is not yet available as npm[10] package [25] so it has to be installed manually. After the installation, a configuration

10. Node.js package

file must be set up to tell the framework where the tests are and how the testing environment should look like.

Similarly as with Chai, the tests can be implemented in two syntax ways: TDD and BDD. The following examples are actual tests implemented for Talker application, running with Intern, using BDD style. The overall test structure uses Dojo (library Intern depends on, providing define method) and it look as follows:

```
define([
  /* load dependencies */
], function (bdd, expect) {
  with (bdd) {

    /* test suite */
    describe('StropheAdapter', function () {

      /* before each test */
      beforeEach(function() {
        adapter = new StropheAdapter();
        send = adapter.connection.send;
      });

      /* inner test suite = set of tests */
      describe('#constructor', function () {

        /* test itself, with description */
        it('should do something...', function () {
          /* perform test, make assertions */
        });
      });
    });
  }
});
```

First, there are dependencies defined so the framework loads them before the test is executed. Almost all JS files included in the application are defined here to emulate the environment of the application as precisely as possible. The next statement tells BDD style is going to be used. Later on, actual test suites with tests are listed. Each suite can mention `before`, `beforeEach`, `after` and `afterEach` functions to be run, similarly as other testing frameworks allow. A test itself consists of a description label. This label should describe the test aim and hence identify it. Inside the test, any code can be executed as well as assertions can be carried out.

JavaScript differs from common programming languages in the way the asynchronous code is executed. JavaScript does not provide multithreading by default [30]. Hence, the pattern of passing a callback to the asynchronous operation is used instead. Unfortunatelly, it is nontrivial task to test asynchronous doce in JavaScript. Intern provides convenient way to do so using promises pattern. The following example tests that `start` method connects

the application to XMPP server in 10 seconds, using Chai assertion to check it.

```
it('should connect and call onConnection callback', function () {
    var dfd = this.async(10000);
    adapter.start(dfd.rejectOnError(function (status) {
        if (!(status === Strophe.Status.ATTACHED
                || status === Strophe.Status.CONNECTED)) {
            return;
        }

        expect(adapter.connection.connected).to.be.true;
        // explicitly resolve the test since
        // it was successful to this point
        dfd.resolve();
    }));
});
```

The callback is hand over to start method to be triggered every time the connection status changes. This means it is also triggered if Strophe only performs the authentication or even if the connection fails! Therefore, the promise is resolved (so the test succeeds) only if the correct status is passed as an argument.

It has to be admitted that the previous test is not real unit test, mocking all surrounding objects the function interacts with. Considering that the goal of testing is making sure the implemented methods work as expected, being implemented in this way is the easiest and most straightforward way to achieve it. Mocking the libraries and network requests would require substantial architecture changes in the application, leading to inappropriate complexity and problems for anyone to work with it "out of the box".

5.9.3 Mocha unit tests

Mocha is another framework for creating unit tests in JavaScript. The structure of tests is similar to Intern, Mocha also supports BDD style with `describe` and `it` keywords. What is different is the way Mocha tests are executed and dependencies are loaded. Mocha is more lightweight than Intern, it is just Node package, not including any configuration files and environment administration.

The easiest way to run Mocha is from command line: `mocha test-file.js`. However, to be able to use Chai to do assertions, each test file must declare it with Node `require` function. The unit test implemented with mocha then looks as the following example - similar to Intern test:

```
var expect = require('chai').expect;

describe('StropheAdapter', function(){
    describe('#constructor', function(){
        it('should do something...', function () {
          /* perform test, make assertions */
```

```
        });
    })
});
```

5.9.4 JsTestDriver framework

Running Mocha tests in console with Node brings one huge disadvantage. There is no easy option to load custom JavaScript files which are necessary to perform the test, unless they are packed as npm packages. Therefore, I used another tool called JS Test Driver to wrap Mocha tests, load dependencies before executing them and also run them in the browser environment.

JS Test Driver (JSTD) is a Java library allowing the developers to run the tests in certain environment, according to the specified configuration file. Although it has not originally been intended to be used with Mocha, there is an adapter aviailable which bridges this imperfection. Furthermore, JSTD can be easily run from PHPStorm or WebStorm IDEs (there is a plugin in the IDE especially for JSTD).

There is a configuration file for JSTD test suite, specifying the scrpits to be loaded and test files to be executed, similarly as Intern does. The structure is self-explaining:

```
load:
  - "lib/mocha.js"
  - "lib/chai.js"
  - ".../strophe_adapter.js"
test:
  - "test-file.js"
```

Mocha and Chai have to be loaded explicitly. Moreover, Mocha and Chai must be set up manually in each test file:

```
mocha.setup('bdd');
expect = chai.expect;
```

Although JSTD looks like worthwhile tool, I had serious problems with keeping it running. Nondeterministic behavior, when the same test (simple test without side effects, network dependencies and so on) sometimes passes and sometimes results in infinite loop never ending the process, makes JSTD almost impossible to be used.

5.9.5 Selenium tests

Chapter 6

Conclusion

TODO: conclusion

Bibliography

[1] Wauters, G.: *Adobe Flash for Android: Gone with barely a whimper*, Digital Trends, 8/17/2012 [retrieved 4/5/2013], from <http://www.digitaltrends.com/mobile/adobe-flash-for-android-gone-with-barely-a-whimper/>. 2.5.1

[2] AOL Inc.: *AOL Trademark List*, 3/15/2011 [retrieved 2/20/2013], from <http://legal.aol.com/trademarks/>. 1

[3] The Apache Software Foundation: *Apache Core Features*, 2013 [retrieved 2/23/2013], from <http://httpd.apache.org/docs/2.2/mod/core.html>. 2.1

[4] Burnham, T.: *Async JavaScript*, The Pragmatic Programmers, 2012, 978-1-93778-527-7, 104 (3). 5.5

[5] Moffitt, J.: *Getting Attached To Strophe*, 8/3/2008 [retrieved 4/15/2013], from <http://metajack.im/2008/10/03/getting-attached-to-strophe/>. 4.1.2

[6] Russell, A. and Wilkins, G. and Davis, D. and Nesbitt, M.: *The Bayeux Specification*, 2007 [retrieved 4/5/2013], The Dojo Foundation, from <http://svn.cometd.org/trunk/bayeux/bayeux.html>. 2.5.3

[7] Staatss, B. (Brianstaats): *"If you have to customize 1/5 of a reusable component, its likely better to write it from scratch @trek at #embercamp"*, 2/15/2013 [retrieved 2/23/2013], Tweet. 1

[8] Donko, P. and Kunc, P. and Novák, M. and Smolka, P. and Volmut, J.: *Celebrio System*, 2013 [retrieved 2/19/2013], from <http://www.celebriosoftware.com/celebrio-system>. 1

[9] Luer, J.: *Chai API - BDD*, 2013 [retrieved 5/4/2013], from <http://chaijs.com/api/bdd/>. 5.9.1

[10] Luer, J.: *Chai API - TDD*, 2013 [retrieved 5/4/2013], from <http://chaijs.com/api/assert/>. 5.9.1

[11] Freeman, A.: *The Definitive Guide to HTML5 - TODO cite from page 873 multimedia chapter*, Apress, 2011, 978-1-4302-3960-4, 1080 (880). 2.4.2

[12] Wang, V. and Salim, F. and Moskovits, P.: *The Definitive Guide to HTML5 WebSocket*, Apress, 2012, 978-1430247401, 210 (140, 156,). 2.2

[13] Smolka, P. and Novák, M.: *Elderly people and the computers*, 2/11/2013 [retrieved 2/19/2013], from <http://infogr.am/Seniori-a-pocitace>. 1

[14] Bodmer, M.: *Ember.js Application Development How-to*, Packt Publishing, 2013, 978-1-78216-338-1, 40 (16). 5.3.3

[15] Skeie, J.: *Ember.js in Action*, Manning Publications, 2013, 978-1617291456, 325 (21). 5.3.2

[16] Olanoff, D.: *Facebook Announces Monthly Active Users Were At 1.01 Billion As Of September 30th*, TechCrunch, 10/23/2012 [retrieved 2/19/2013], from <http://techcrunch.com/2012/10/23/facebook-announces-monthly-active-users-were-at-1-01-billion-as-of-septem 1

[17] Cairns, J.: *"Fat model, skinny controller" is a load of rubbish*, 4/11/2013 [retrieved 4/21/2013], from <http://joncairns.com/2013/04/fat-model-skinny-controller-is-a-load-of-rubbish/>. 5.2.3

[18] Facebook Developers: *Facebook Chat API*, 2/12/2013 [retrieved 2/20/2013], from http://xmpp.org/about-xmpp/history/ <http://legal.aol.com/trademarks/>. 1

[19] Letuchy, E.: *Facebook Chat*, 5/14/2008 [retrieved 2/23/2013], from <https://www.facebook.com/note.php?note_id=14218138919>. 1

[20] Wauters, R.: *Flash Player To Come Bundled With Google Chrome, New Browser Plugin API Coming*, TechCrunch, 3/30/2010 [retrieved 4/5/2013], from <http://techcrunch.com/2010/03/30/flash-player-to-come-bundled-with-google-chrome-new-browser-plugin-api-co 2.5.1

[21] Sullivan, J.: *Google backslides on federated instant messaging, on purpose?*, 3/15/2013 [retrieved 3/31/2013], from <https://www.fsf.org/blogs/sysadmin/google-backslides-on-federated-instant-messaging-on-purpose>. 3.5

[22] Google Developers: *Google Talk Developer Documentation*, 3/23/2012 [retrieved 2/20/2013], from <https://developers.google.com/talk/talk_developers_home>. 1

[23] Rosenberg, J. and Keranen, A. and Lowekamp, B. and Roach, A.: *TCP Candidates with Interactive Connectivity Establishment (ICE)*, 5/18/2012 [retrieved 4/6/2013], from <http://tools.ietf.org/html/draft-ietf-mmusic-ice-tcp-16>. 2.4.2

[24] Terabyte Media: *Web Browser Usage Statistics*, 12/2012 [retrieved 3/3/2013], from <http://www.statowl.com/web_browser_usage_by_version.php?limit%5B%5D=ie>. 2.2.3

[25] Bouchon, P.: *Meet your newest Intern*, 5/1/2013 [retrieved 5/4/2013], from <http://www.sitepen.com/blog/2013/05/01/intern-javascript-testing/>. 5.9.2

[26] Miniwatts Marketing Group: *Internet Users in the World - 2012 Q2*, Internet World Stats, 2/17/2013 [retrieved 2/19/2013], from <http://www.internetworldstats.com/stats.htm>. 1

[27] Apple Support Team: *Does the iPhone support Flash?*, 2007 [retrieved 4/5/2013], from <http://www.iphonefaq.org/archives/9730>. 2.5.1

[28] Cridland, D.: *Google: "The Future is Jingle"*, 6/23/2011 [retrieved 3/31/2013], from <http://xmpp.org/2011/06/the-future-is-jingle/>. 3.4

[29] The jQuery Foundation: *jQuery API Documentation*, 2013 [retrieved 4/10/2013], from <http://api.jquery.com/append/>. 4.1.4

[30] Flanagan, D.: *JavaScript: The Definitive Guide*, O'Reilly Media, 2011, 978-0-596-80552-4, 1100 (320, 322, 333). 2.2, 5.5, 5.9.2

[31] Lindley, C.: *JavaScript Enlightenment*, O'Reilly Media, 2013, 978-1449342883, 166 (49). 5.5

[32] Google Developers: *About libjingle*, 3/23/2012 [retrieved 3/31/2013], from <https://developers.google.com/talk/libjingle/>. 3.4

[33] Mozilla Developers: *WebSockets*, 2/4/2013 [retrieved 2/25/2013], from <https://developer.mozilla.org/en-US/docs/WebSockets>. 3

[34] Ziadé, T.: *A new development era (essay)*, 1/25/2013 [retrieved 4/26/2013], from <http://blog.ziade.org/2013/01/25/a-new-development-era-essay/>. 5.3.1

[35] Lubbers, P. and Salim, F. and Albers, B.: *Pro HTML5 Programming*, Apress, 2011, 978-1-4302-3864-5, 352 (165, ...). 2.1.1, 2.2.2

[36] Lengstorf, J. and Leggetter, P.: *Realtime Web Apps*, Apress, 2013, 978-1430246206, 400.

[37] Saint-Andre, P.: *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*, 2004 [retrieved 5/4/2013], Jabber Software Foundation, from <http://xmpp.org/rfcs/rfc3921.html>. 5.7.3

[38] Loreto, S. and Saint-Andre, P. and Salsano, S. and Wilkins, G.: *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*, 4/2011 [retrieved 2/23/2013], from <http://www.ietf.org/rfc/rfc6202.txt>. 2.1.1, 2.1.1, 2.1.2, 2.1.2

[39] Fielding, R. and Gettis, J. and Mogul, J. and Frystyk, H. and Masinter, L. and Leach, P. and Berners-Lee, T.: *Hypertext Transfer Protocol – HTTP/1.1*, 6/1999 [retrieved 2/23/2013], from <http://www.w3.org/Protocols/rfc2616/rfc2616.html>. 2, 2.1, 2.2.1

[40] Fette, I. and Melnikov, A.: *The WebSocket Protocol*, 12/2011 [retrieved 2/24/2013], Internet Engineering Task Force (IETF), from <http://tools.ietf.org/html/rfc6455>. 2.2.1, 2, 2.2.2

[41] Ward, J.: *What is a Rich Internet Application?*, 10/17/2007 [retrieved 2/19/2013], from <http://www.jamesward.com/2007/10/17/what-is-a-rich-internet-application/>. 1

[42] Prosody: *Setting up a BOSH server*, [retrieved 3/10/2013], from <http://prosody.im/doc/setting_up_bosh>. 3.2

[43] Edwards, D.: *SignalR FAQ*, GitHub, 9/7/2012 [retrieved 4/5/2013], from <https://github.com/SignalR/SignalR/wiki/Faq>. 2.5.4

[44] Smith, A.: *Does SkypeKit work on Android?*, 8/7/2012 [retrieved 2/23/2013], from <http://devforum.skype.com/t5/SkypeKit-FAQs/Does-SkypeKit-work-on-Android/m-p/16490/thread-id/78>. 1

[45] Microsoft: *Skype URIs*, 2013 [retrieved 2/23/2013], from <http://dev.skype.com/skype-uri>. 1

[46] Gustafsson, P.: *Spammy invites*, 2/13/2013 [retrieved 3/31/2013], from <http://mail.jabber.org/pipermail/operators/2013-February/001571.html>. 3.5

[47] Gustafsson, P.: *Update on spammy invites*, 5/4/2013 [retrieved 4/9/2013], from <http://mail.jabber.org/pipermail/operators/2013-April/001672.html>. 3.5

[48] Hickson, I.: *Server-Sent Events*, 3/29/2013 [retrieved 4/5/2013], W3C, from <http://dev.w3.org/html5/eventsource/>. 2.3

[49] Sharp, R.: *Server-Sent Events*, 1/24/2012 [retrieved 4/5/2013], HTML5Doctor, from <http://html5doctor.com/server-sent-events/>. 2.3

[50] Moffitt, J.: *Strophe.js API Documentation*, 2013 [retrieved 4/10/2013], from <http://strophe.im/strophejs/doc/1.0.2/files2/strophe-js.html>. 4.1.4

[51] Barnes, C. and Blake, H. and Pinder, D.: *Creating and Delivering Your Value Proposition: Managing Customer Experience for Profit*, Kogan Page Publishers, 2009, 978-0749458591, 232. 5.1.1

[52] Alund, S.: *Bowser – The World's First WebRTC-Enabled Mobile Browser*, 10/19/2012 [retrieved 4/6/2013], from <https://labs.ericsson.com/blog/bowser-the-world-s-first-webrtc-enabled-mobile-browser>. 2.4.3

[53] Reavy, M. and Lachapelle, S.: *Hello Firefox, this is Chrome calling!*, 2/4/2013 [retrieved 4/6/2013], from <http://blog.chromium.org/2013/02/hello-firefox-this-is-chrome-calling.html>. 2.4.3

[54] Aboba, B. and Thomson, M.: *Customizable, Ubiquitous Real Time Communication over the Web (CU-RTC-Web)*, 8/9/2012 [retrieved 4/6/2013], Microsoft, from <http://html5labs.interoperabilitybridges.com/cu-rtc-web/cu-rtc-web.htm>. 2.4.3

[55] Bergkvist, A. and Burnett, D. and Jennings, C. and Narayanan, A.: *WebRTC 1.0: Real-time Communication Between Browsers*, 3/22/2013 [retrieved 4/6/2013], W3C, from <http://dev.w3.org/2011/webrtc/editor/webrtc.html>. 2.4.2

[56] Dutton, S.: *Getting Started with WebRTC*, 7/23/2012 [retrieved 4/6/2013], HTML5 Rocks, from <http://www.html5rocks.com/en/tutorials/webrtc/basics/>. 2.4.1

[57] Roettgers, J.: *Scoop: Microsoft bets on WebRTC for Skype's browser future*, 6/26/2012 [retrieved 4/6/2013], from <http://gigaom.com/2012/06/26/skype-webrtc-web-client/>. 2.4.3

[58] Doubango Telecom: *webrtc4all*, 2012 [retrieved 4/6/2013], from <https://code.google.com/p/webrtc4all/>. 2.4.3

[59] Hickson, I.: *The WebSocket API*, 2/9/2013 [retrieved 2/24/2013], W3C, from <http://dev.w3.org/html5/websockets/>. 2.2.3

[60] StatCounter GlobalStats: *Can I use Web Sockets?*, 2/2013 [retrieved 3/3/2013], from <http://caniuse.com/websockets>. 2.2.3

[61] Ubl, M. and Kitamura, E.: *Introducing WebSockets: Bringing Sockets to the Web*, 2/13/2012 [retrieved 2/25/2013], from <http://www.html5rocks.com/en/tutorials/websockets/basics/>. 2.2.3

[62] IANA: *WebSocket Protocol Registries*, 11/13/2012 [retrieved 2/25/2013], from <http://www.iana.org/assignments/websocket/websocket.xml>. 2.2.3

[63] Lubbers, P.: *How HTML5 Web Sockets Interact With Proxy Servers*, 3/16/2011 [retrieved 2/24/2013], InfoQ, from <http://tools.ietf.org/html/rfc6455>. 2.2

[64] Hildebrand, J. and Kaes, C. and Waite, D.: *XEP-0025: Jabber HTTP Polling*, XMPP Standards Foundation, 2009 [retrieved 3/10/2013], from <http://xmpp.org/extensions/xep-0025.html>. 2

[65] Paterson, I. and Saint-Andre, P. and Smith, D. and Moffit, J.: *XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH)*, XMPP Standards Foundation, 2010 [retrieved 3/10/2013], from <http://xmpp.org/extensions/xep-0124.html>. 3.2

[66] Ludwig, S. and Beda, J. and Saint-Andre, P. and McQueen, R. and Egan, S. and Hildebrand, J.: *XEP-0166: Jingle*, XMPP Standards Foundation, 2009 [retrieved 3/31/2013], from <http://xmpp.org/extensions/xep-0166.html>. 3.4

[67] Paterson, I. and Saint-Andre, P.: *XEP-0206: XMPP Over BOSH*, XMPP Standards Foundation, 2010 [retrieved 3/10/2013], from <http://xmpp.org/extensions/xep-0206.html>. 2, 5.4

[68] The XMPP Standards Foundation: *History of XMPP*, 1/27/2010 [retrieved 2/20/2013], from http://xmpp.org/about-xmpp/history/ <http://legal.aol.com/trademarks/>. 1

[69] Moffitt, J.: *Professional XMPP Programming with JavaScript and jQuery*, John Wiley & Sons, 2010, 978-0470540718, 432 (58, 380, 402). 4.1.2, 4.1.3, 4.1.4, 4.2

[70] Moffit, J.: *An XMPP Sub-protocol for WebSocket*, 2/25/2013 [retrieved 3/10/2013], from <http://datatracker.ietf.org/doc/draft-moffitt-xmpp-over-websocket/>. 3.3, 4.1

[71] Saint-Andre, P. and Smith, K. and Tronçon, R.: *XMPP: The Definitive Guide*, Sebastopol: O'Reilly, 2009, 978-0-596-52126-4, 310 (7, 13, 14, 16,). 3, 3.1

[72] Katz, Y.: *Building Web Applications with Ember.js*, 4/13/2013 [retrieved 4/26/2013], YouTube, from <http://www.youtube.com/watch?v=u6RFyVN9sNg>. 5.3.1

Index

Appendix A

Screenshots of the application

Some screenshots from Celebrio Talker