Real-time Communication in Web Browser

MASTER THESIS

Pavel Smolka

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, 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. [20]

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. [32] 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. [12] 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. [11] 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. [17][13] 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. [14] Combination of the facts that XMPP is open technology with open-sourced client and server implementations [58] 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".

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 Tok-Box 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".

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. [35] 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. [36]

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

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. [30] 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. [30] 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. [29] 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. [27] 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). [27] It is obvious that there is huge unnecessary network overhead when using HTTP polling instead of Web-Sockets.

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

Visual Paradigm for UML Standard Edition(Masaryk University)

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. [29]

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. [29] 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): [29]

```
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. [29]

8

Visual Paradigm for UML Standard Edition(Masaryk University)

Figure 2.5: HTTP streaming

2.2 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). [24]

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. [53][10] 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. [10]

2.2.1 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: [31]

```
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: [31]

```
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`, [30] 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. [31]

---

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. [31]

`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. [31] 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. [31] 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. [27]

2.2.3 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] [49] 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: [51]

---

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`. [26]
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. [52]

WebSocket interface provides at least four event handlers, to each of them a custom callback can be attached. [49] 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: [51]

```
// 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 realtime 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. [50] Particularly, all versions of Internet Explorer below 10 (which means more than 98 % of IE users in November 2012) [19] 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 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 Web-Sockets so that the topic is introduced here. SSE have been standardized as part of HTML5 standard. [38] 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. [39] 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 WebRTC

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. [46]

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. [45] 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. [18] 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. [45] 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. [9]

```
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] [43] 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. [48] 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. [42]

Microsoft came up with another approach. Even though Internet Explorer does not support WebRTC, Microsoft invented their own API standard proposal. [44] 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. [47] 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. [15] 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. [21] 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

TODO: OpenTok

2.5.3 Bayeux

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

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. [34] 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 Google Talk

TODO

2.5.6 Google hangouts API

TODO

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. [61] 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. [61] 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). [61]

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. [61]

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

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 `Stanzas`. [61] 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 successfuly 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): [61]

```
<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] [55] 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. [54] 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. [57]

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: [55]

```
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 [33]. 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 Web-Sockets 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. [60] 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

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. [56]

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. [25] 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. [22]

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. [16] 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. [37] The issue is still opened by the time this thesis is submitted.

If Google closes 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.

---

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

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". 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.

Visual Paradigm for UML Standard Editior

Application Layer

Jingle

XMPP

BOSH

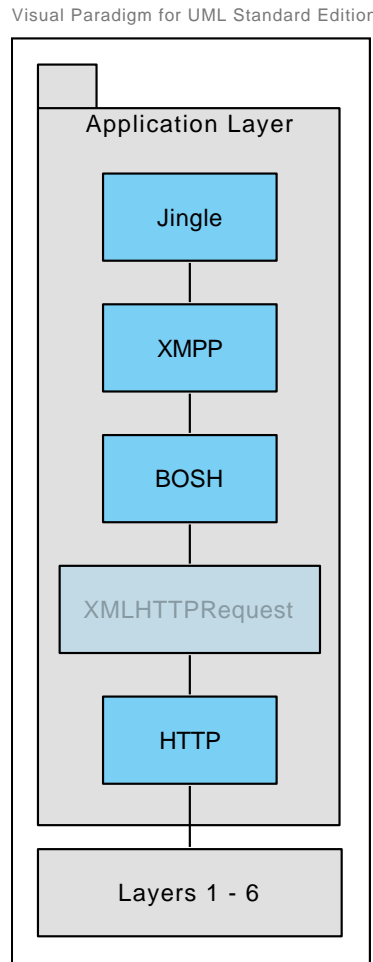XMLHTTPRequest

HTTP

Layers 1 - 6

Figure 4.1: Complexity of the communication protocols at application layer

4.1 Strophe.js

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.

[60]

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. [59]

Attaching to an existing connection in the JavaScript application using Strophe.js is quite simple, similar to initializing a new connection: [59]

```
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. [59] 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().` [23]

Strophe.js comes up with completely independent XML creator, however inspired by jQuery. [59] The functionality is to be found within `Strophe.Builder` object. It allows the library user to create XML elements such as follows: [40]

```
$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".

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`. [59] 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. [59] 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

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 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 and design

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. [41] 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.
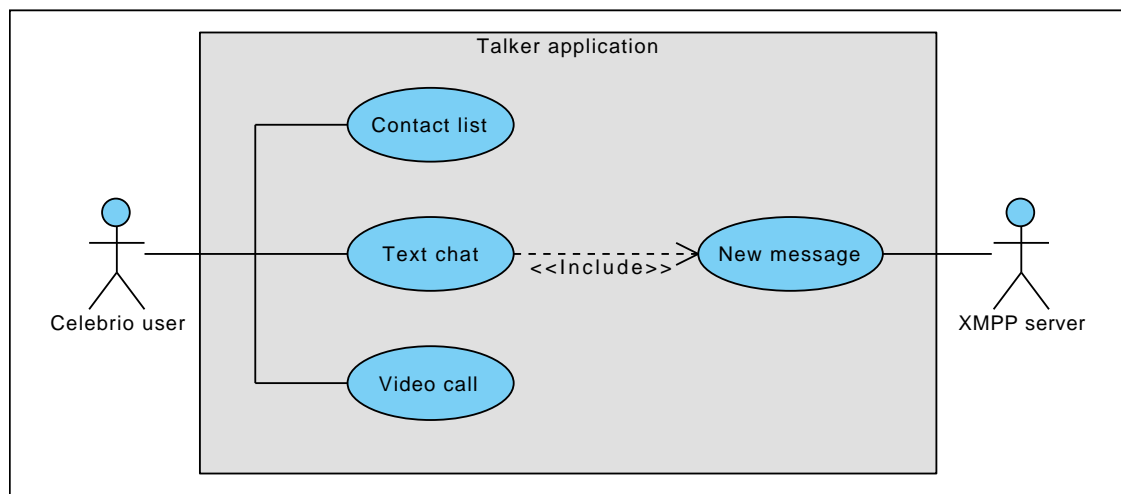

Figure 5.1: Talker use case diagram

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 5.5. 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 Used 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 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 5.2.

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.

5.1.4 Application architecture

TODO: application analysis + design.

TODO: class diagram (if any),

TODO: sequence/action diagram - is there any? maybe communication... js client - php server - boshserver - xmppserver. Move video call state machine diagram from video call section here?

5.2 Ember.js

Describe Javascript client-side MVC frameworks overall, compare, tell why we used Ember

5.2.1 Client-side MVC

5.2.2 Comparsion with other frameworks

Tell why Ember is the best :-)

5.2.3 Handlebars templates

5.2.4 Routing

5.3 Initializing the connection

Strophe init, passing credentials, attaching to the existing connection?
    TODO programming - attach connection

5.4 Processing events

everything about processing incoming events. mention appending messages to the message list

5.4.1 Notifications

Webtop notifications + sound

5.4.2 Logging

TODO: mention Log4js or another tool we might use.

5.5 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.

5.5.1 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.[1] 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.

5.5.2 Retrieving the roster

TODO: how the roster is retrieved and handled (do not duplicate XMPP chapter)

5.5.3 Subscriptions

TODO: subscriptions. do not duplicate XMPP chapter (however this is not very much mentioned there)

5.6 Video calling

TODO: Video framework, flash fallback

5.6.1 OpenTok

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

---

1. Jabber ID

5.6.2 Implementation
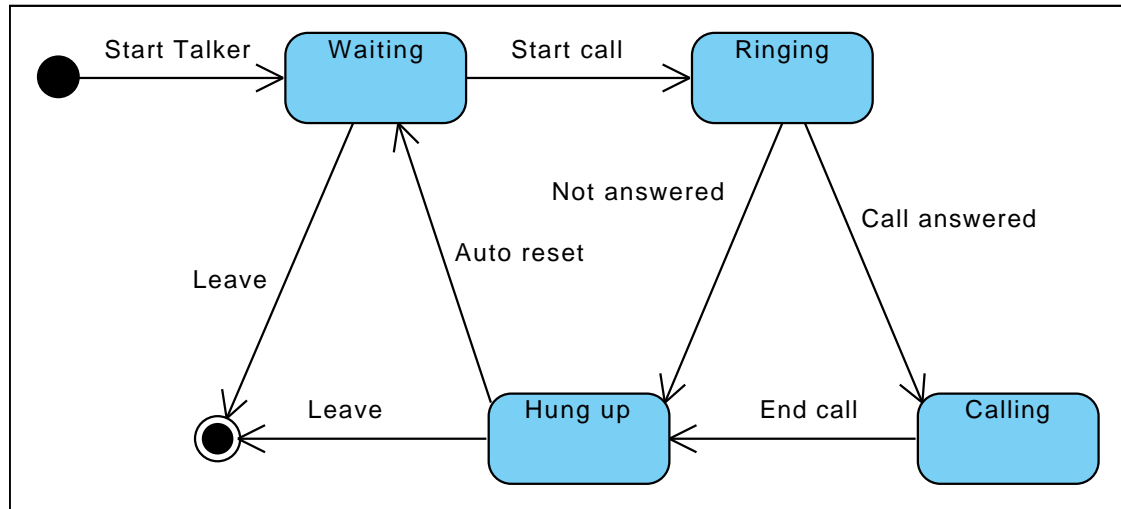
Figure 5.2: Video call states

5.7 Testing

TODO: Unit tests, performance tests if there is enough time
    TODO: programming

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.

[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] 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

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

[11] 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

[12] 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

[13] 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

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

[15] 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

[16] 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

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

[18] 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

[19] 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

[20] 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

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

[22] 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

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

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

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

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

[27] 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

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

[29] 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

[30] 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

[31] 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

[32] 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

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

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

[35] 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

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

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

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

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

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

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

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

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

[51] 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

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

[53] 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

[54] 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

[55] 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

[56] 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

[57] 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

[58] 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

[59] 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

[60] 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

[61] 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

Index

List of Figures

List of Tables

Appendix A

Screenshots of the application

Some screenshots from Celebrio Talker