MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Real-time Communication in Web Browser

MASTER THESIS

Pavel Smolka

Brno, 2013

Declaration

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

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

Acknowledgement

Thanks to everyone... TODO: Petr Kunc (help with OpenTok & RealTime communication,

Abstract

TODO

Keywords

XMPP, real-time communication, RTC, Celebrio, web browser, HTTP, Comet, JavaScript, 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 accesibility 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. [13]

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. [21] 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. [9] 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. Celebrio, simple software for the elderly simulating the operating system interface, is typical

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.

example of rich internet application. [5] 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. [8] 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.

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. [12][10] 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. [11] Combination of the facts that XMPP is open technology with open-sourced client and server implementations [28] 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 infromation 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 TokBox Inc.[4] Nevertheless, following the rule that "If you have to customize 1/5 of a reusable component, its likely better to write it from scratch", [4] 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, "XMPP client in JavaScript". Within the programming part 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

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

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. [22] 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. [23]

Finally, there are several notes about "inter-process communication" between different applications running separately in various browser frames, tabs or even windows. Chapter 6, "Inter-process Communication Framework" covers this topic and describes the issues we came accross when implementing such functionality for Celebrio, where every application runs in separate iframe.

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

Chapter 2

Bidirectional communication

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. [19] 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 - re-shaping HTTP in Section 2.1 and brand new approach in Section 2.2, bypassing HTTP at all. Unfortunatelly, every approach brings also some disadvantages.

2.1 Using HTTP requests

Historicaly the first approach, for a very long time the only one, is hacking HTTP to achieve RTC. The idea is very simple, depicted in TODO figure. Generally, the client sends an extra request and it is not awaiting the response immediately. Instead, the server keeps the request for some time. 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 (acutally implemented even before, but not covered in the RFC specification), there 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. [19] Even though the default timeout after which the server closes the connection is only a few seconds, [2] the persistent TCP connection is very useful for delivering various resources (stylesheets, scripts, images, etc.) to the client without the unnecessary overhead of creating multiple streams. However, every single transmission

1. In this thesis, this behaviour is commonly refered 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. Before AJAX became the essential part of every web app, ifram

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. TODO figure.

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 reponse is sent. [18] 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. When the server receives (or even makes up by oneself) new data, it carries out the response with the respective content. 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. [16] 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 2.1 shows the comparsion 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). [16]

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 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. [18]

2.1.2 HTTP streaming

HTTP streaming is slightly different method 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 with 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. [18] See the schema on TODO figure.

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

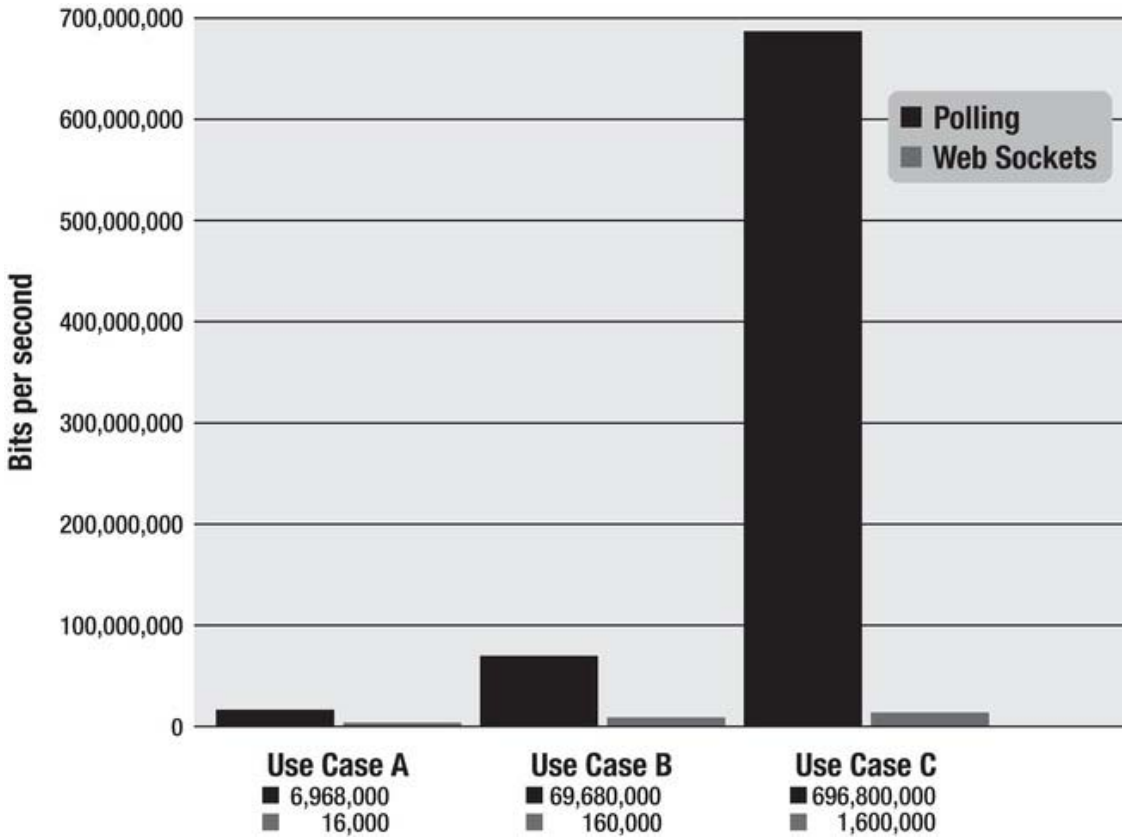

Figure 2.1: Comparison of the unnecessary network overhead between the HTTP polling and using WebSockets [16]

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): [18]

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

25
This is the data in the first chunk

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. [18]

2.2 WebSockets

Although the World Wide Web with HTTP request/response schema has never been intended to server 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). [14]

Similarly as in HTTP, there is an unencrypted version of WebSockets working directly on top of TCP connection. The simpliest 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. Transparent 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. [27][7] 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. [7]

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: [20]

2. According to RFC6455, the protocol is designed to work over HTTP ports 80 and 443 as well to support

```
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: [20]

```
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`, [19] 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. [20]

`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 resultis returned as `Sec-WebSocket-Accept` header field, base64-encoded. [20] 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 that every message appears in the same state as it was sent.

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

HTTP proxies. However, the design is not limited to HTTP and the future implementations can use simplier handshake over a dedicated port. [20]

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. [20] 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. [16]

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] [24] 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: [25]

```
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. [26]

`WebSocket` interface provides at least four event handlers, to each of them a custom callback can be attached. [24] Those are `onopen`, `onmessage`, `onerror` and `onclose`. The names are quite self-explanatory, they serve as the event listeners watching the incoming activity. The callback registration can look as follows:

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

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`. [15]
4. Properties of window are accessible in JavaScript directly. Simple test `window.WebSocket === WebSocket` returning `true` proves it.

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: [25]

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

// Sending canvas ImageData as ArrayBuffer
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);
```

2.3 Existing RTC frameworks

Actually, there are other options such as using various browser plugins (e.g. Adobe Flash) to provide the base layer for TCP connections, but such tools has not been used within this thesis and are not further dealt with.

Describe other approaches to RTC in WB than XMPP. Tell why we didn't use them (or that we used - OpenTok).

Mention:
Bayeux: http://svn.cometd.org/trunk/bayeux/bayeux.html
OpenTok
SignalR (.NET)
WebRTC (http://www.webrtc.org/)
Google hangouts API

Chapter 3

Extensible Messaging and Presence Protocol

Extensible Messaging and Presence Protocol (XMPP) technologies were invented by Jeremie Miller in 1998. [29] 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. [29] 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.

3.1 Fundamental bulding blocks in XMPP

Mention the terms as stanza, roster, describe the subscription mechanism.

3.2 XMPP over BOSH

Describe BOSH (http://xmpp.org/extensions/xep-0124.html)
and XMPP Over BOSH (http://xmpp.org/extensions/xep-0206.html).
Later on, description of BOSH extension, including the advantages and limitations. Describe also the connection to HTTP protocol.

3.3 XMPP over WebSockets

Usually if the WebSockets are used, the XMPP server needs one extra layer, usually implemented as the server plugin, to be able to communicate directly with the web browser.
http://blog.superfeedr.com/xmpp-over-websockets/
luajit - lua jit compiler
It was neccessary to install: apt balíky luajit + liblua5.1-bitop0

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. [1]

TODO create diagram here (XMPP server - plugin - incoming connection vs xmpp server - standard connection).

3.4 Jingle

Jingle extension - multimedia.
http://xmpp.org/extensions/xep-0166.html

3.5 Inter-process communication

Pub-sub extensions for inter-process communication.
http://xmpp.org/extensions/xep-0060.html

Chapter 4

XMPP client in JavaScript

Describe the tools that can be used to implement RTC in WB (and which were used to implement Celebrio Talker)

Strophe (simple XMPP in Javascript)
Strophe connecting/attaching - security issues. TODO programming
Strophe plugins
Possible server-side implementations (JAXL, XMPPHP, ...)

Chapter 5

Talker

Describe the Talker application in Celebrio.

Mention what we expected from the app (value proposition)

Then, application analysis, design and implmementation.

Describe the architecture and used tools&frameworks: JS + Ember.js, OpenTok, WebSockets in new OpenTok

Don't forget to use UML: use case diagram, class diagram (if any), sequence/action diagram

5.1 Ember.js

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

5.1.1 Subsection about Ember...

Chapter 6

Inter-process Communication Framework

Implement and describe framework for inter-process communication in Celebrio.

First, the lightweight one (which we already have), then implement the "heavy" one, if there's enough time.

Chapter 7

Conclusion

conclusion

7.1 Another part of the conclusion

Another part of the conclusion... just to have subchapter here

Bibliography

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

[2] 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

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

[4] 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

[5] 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

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

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

[8] 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

[9] 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

[10] 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

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

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

[13] 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

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

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

[16] Lubbers, P. and Salim, F. and Albers, B.: *Pro HTML5 Programming*, Apress, 2011, 978-1-4302-3864-5, 352 (165, ...). 2.1.1, 2.1, 2.2.2

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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

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

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

[25] 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

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

[27] 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

[28] 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

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

Index

Appendix A

Screenshots of the application

Some screenshots from Celebrio Talker