Implementing extended functionality for an HTML5 client for remote desktops

by

Samuel Mannehed

LIU-IDA/LITH-EX-A—14/015--SE

6/28/14
Final thesis

Implementing extended functionality for an HTML5 client for remote desktops

by

Samuel Mannehed

LIU-IDA/LITH-EX-A--14/015--SE

June 28, 2014

Supervisors: Peter Åstrand (Cendio AB), Maria Vasilevskaya (IDA)
Examiner: Prof. Simin Nadjm-Tehrani
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/
Abstract

The rising demand to be able to work and use applications wherever you go dictates the need to be able to connect remotely to desktops. This challenge is addressed by so called remote desktop applications that allow the user to locally view and control a remote computer. Such a solution requires installation of specialized software on both the remote computer and the local computer. The software on the local computer is the client software. Most implementations of such client software are native clients which are software packages installed on the client machine. A logical evolution path for remote desktop clients is the shift to a web browser version that will not require the installation of any specific software.

HTML5 is an upcoming standard and markup language for the world wide web. HTML5 brings new features which open up new possibilities for web developers. The potential of HTML5 technologies draw remote desktop developers' attention. New HTML5 versions of remote desktop clients have started to appear due to this.

However, these new realizations are limited and far from being powerful enough to replace native clients. Thus, it is still completely unknown whether an HTML5-based remote desktop client has possibilities to replace native remote desktop clients. This thesis aims to address this knowledge gap. Essential technical features such as audio and local printing for remote desktop solutions were investigated in the context of an HTML5 client. Functionality was implemented and tested, and future development was evaluated.

The resulting features that were implemented along with the evaluated future features were subject to some limitations inherited by the HTML5 platform. As a result of this work, it was concluded that it is not possible to achieve features with the same level of functionality as the features seen in the native clients. In order to accomplish this, the browsers would have to implement specific interfaces for the required hardware and systems. These limitations prevent complete replacement of native remote desktop clients with a HTML5 based client in the near future. However, the HTML5 client has a dedicated area where its features suit their purpose. The HTML5 client is available where the native clients are not. The browser platform brings unprecedented accessibility advantages.
Acknowledgements

I would like to start by thanking my examiner Simin Nadjm-Tehrani and my supervisor Maria Vasilevskaya for examining and supervising my masters thesis. I have really appreciated the advice and the thoroughness.

I also want to thank Cendio and my supervisor and manager, Peter Åstrand. Thank you for your patience and support. I am very glad that I got this opportunity, I have learned a lot and have had a lot of fun.
Abbreviations

ADB – Android Debug Bridge
AJAX – Asynchronous JavaScript And XML
ALSA – Advanced Linux Sound Architecture
API – Application Programming Interface
CUPS – Common Unix Printing System
ESD – Enlightened Sound Daemon
HTML – HyperText Markup Language
IDE – Integrated Development Environment
IETF – Internet Engineering Task Force
IP – Internet Protocol
LPD – Line Printer Daemon
PCM – Pulse-Code Modulation
PC/SC – Personal Computer / Smart Card
PDF – Portable Document Format
PS – PostScript
PSF – Python Software Foundation
RFB – Remote FrameBuffer
RDP – Remote Desktop Protocol
SSH – Secure Shell
SVN – Apache Subversion
TCP – Transmission Control Protocol
URL – Uniform Resource Locator
VNC – Virtual Network Computing
W3C – World Wide Web Consortium
WTS – Windows Terminal Server
XHTML – Extensible HyperText Markup Language
XML – Extensible Markup Language
# Table of contents

## Chapter 1: Introduction

1.1 Background ........................................................................................................................... 4
1.1.1 ThinLinc ........................................................................................................................... 4
1.1.2 The HTML5 client ........................................................................................................... 5
1.2 Problem description ............................................................................................................. 7
1.2.1 Main features .................................................................................................................. 7
1.2.2 Additional features ......................................................................................................... 8
1.3 Goal ....................................................................................................................................... 9
1.4 Methodology ....................................................................................................................... 10
1.5 Scope and limitations .......................................................................................................... 10
1.6 Thesis structure ................................................................................................................. 11

## Chapter 2: Technical background

2.1 The concept of HTML5 ....................................................................................................... 12
2.2 Performance ....................................................................................................................... 12
2.3 Technologies and tools .................................................................................................... 12
2.3.1 WebSocket API ............................................................................................................. 13
2.3.2 Virtual network computing ......................................................................................... 13
2.3.3 Editor ............................................................................................................................ 13
2.3.4 Version control systems ............................................................................................ 14
2.3.5 Browser tools .............................................................................................................. 14

## Chapter 3: Technical issues in a remote desktop solution

3.1 Keyboard input .................................................................................................................... 15
3.2 Redirection of audio ......................................................................................................... 16
3.3 Local printing support ....................................................................................................... 18

## Chapter 4: Development of extended functionality

4.1 Comparison of solutions ..................................................................................................... 20
4.1.1 Comparison of solutions for keyboard input ............................................................... 20
4.1.2 Comparison of solutions for redirection of audio ....................................................... 22
4.1.3 Comparison of solutions for local printing support ................................................... 25
4.2 System architecture .......................................................................................................... 27
4.3 Implementation of selected solutions .............................................................................. 29
4.3.1 Implementation for keyboard input ............................................................................. 29
4.3.2 Modifications to tlwebaccess ....................................................................................... 31
4.3.3 Implementation for redirection of audio ..................................................................... 33
4.3.4 Implementation for local printing support ................................................................. 38

## Chapter 5: Evaluation

5.1 Testing of implementation ................................................................................................. 45
5.1.1 Measurement tool ........................................................................................................ 46
5.1.2 Acceptance testing ....................................................................................................... 46
5.1.3 Platform testing .......................................................................................................... 47
5.1.4 Testing setup ............................................................................................................... 48
Chapter 6: Conclusions

6.1 The desired level of functionality .......................................................... 51
6.2 Discussions and future work .................................................................. 51

References

Appendix A: Test results

Appendix B: Interview with Joel Martin
List of tables

1.1.2 Table 1: The supported browsers ......................................................................................... 6
1.2 Table 2: List of features ........................................................................................................ 7
4.1.1 Table 3: Summarized comparison for keyboard input .......................................................... 21
4.1.2 Table 4: Summarized comparison for redirection of audio .................................................. 24
4.1.3 Table 5: Summarized comparison for local printing support .................................................... 26
4.3.3 Table 6: The structure of the init packet ........................................................................... 34
4.3.3 Table 7: The structure of the stream-play packet .................................................................. 35
4.3.4 Table 8: The five printing states in three layers and when each state is entered ....................... 39
5.1.1 Table 9: Wireshark filters used for measurements of each feature in the HTML5 client ......... 46
5.1.2 Table 10: Acceptance tests with both functional and non-functional criteria ....................... 47
5.1.3 Table 11: Tested browsers and platforms ........................................................................... 47

List of figures

1.1.2 Figure 1: The HTML5 client and the native ThinLinc client ...................................................... 5
3.1 Figure 2: The Android on-screen keyboard in the HTML5 client ............................................. 15
3.1 Figure 3: Sound with PulseAudio ............................................................................................. 16
3.2 Figure 4: The different PulseAudio servers ............................................................................. 17
3.3 Figure 5: A print dialogue showing the thinlocal option ............................................................ 18
4.1.2 Figure 6: The streaming solution ....................................................................................... 23
4.2 Figure 7: An overview with parts modified in the thesis highlighted ......................................... 27
4.2 Figure 8: The network protocols used for different features .................................................... 28
4.2 Figure 9: The technologies and protocols used in different layers ........................................... 28
4.3.1 Figure 10: Gesture typing in Google Chrome on Android ..................................................... 29
4.3.4 Figure 11: The numbered transitions between the five states .............................................. 40
4.3.4 Figure 12: The notification that a print file has been received ............................................. 44
4.3.4 Figure 13: The PDF file that was received ........................................................................... 44
5.1 Figure 14: The testing phases with parts presented in the thesis highlighted ............................. 45
Chapter 1: Introduction

This thesis is in partial fulfillment of a Master of Science in computer engineering degree (30 credit points) at Linköping University. This project was carried out as an assignment for the company Cendio AB in Linköping, Sweden.

The user demand to access data and applications is constantly growing. IBM market researchers predicted, in 2004, that server based computing would be growing [1]. A clear indication of this, shown in 2013 by IDC and CRN, is the fact that the market for thin clients is showing substantial growth [2,3]. Thin clients are lightweight computers from which users can access applications and desktops on more powerful servers. Cendio AB, an enterprise software company provides a product to meet this growing demand [4]. This product, ThinLinc, is used by organizations all over the world to access applications running on a broad range of systems. This thesis investigates possibilities for a part of the ThinLinc product called the HTML5 client. Chapter one will introduce this HTML5 client and briefly present some background and motivation, problem statement, and evaluation approach. At the end of the chapter the structure for the rest of the document is outlined.

1.1 Background

Organizations use Cendio’s product, ThinLinc, to access applications running on a broad range of systems. This section introduces ThinLinc followed by the HTML5 client. The latter is the part of the ThinLinc product which is in focus for the work done in the thesis.

1.1.1 ThinLinc

ThinLinc is a thin client and remote desktop Linux server solution which utilizes the Virtual Network Computing system, VNC. VNC will be described further in section 2.3.2. Cendio describes the product as follows [4]:

“ThinLinc provides users with centralized remote access to both Windows and Linux desktops and applications.”

The concept of remote desktops refer, according to Cerling et al, to a software or an operating system feature which allows applications or entire desktops to be executed remotely on a server computer or a PC, while being viewed and manipulated locally [5]. The computer who’s applications are used can be referred to as the remote computer while the computer which views the applications is referred to as the local computer. It is common that the remote computer accepts keyboard and mouse input from the local computer. When the local computer connects to the remote computer through a remote desktop solution a new session is created, a remote session. This session allows the user to use applications from the remote computer.

The ThinLinc product includes software for both the local and the remote computer [6]. The ThinLinc software on the local computer is called the ThinLinc client and the software on the remote Linux computer is called the ThinLinc server. ThinLinc clients connect to ThinLinc servers. The desktop from the remote computer is provided over these connections from the ThinLinc
servers to the ThinLinc clients on the local computer. The ThinLinc server is only available for Linux and Solaris computers, but it is possible to enable Windows applications and desktops if a second server computer, a Windows Terminal Server, WTS, is available. The WTS is required to have the Remote Desktop Services activated to allow access to applications and data remotely. Native ThinLinc clients are available for all major platforms: Linux, Solaris, Windows, and OS X.

1.1.2 The HTML5 client

HTML5 is an upcoming standard for the HyperText Markup Language (HTML) [7]. HTML is, as the name suggests, a markup language. HTML is built with different elements which are defined in tags which are enclosed by the smaller-than and greater-than symbols. HTML is used to create web pages and other information displayed in a web browser. The standard is being developed by the World Wide Web Consortium (W3C) together with the Web Hypertext Application Technology Working Group (WHATWG). Lubbers et al present the main goals for HTML5 as to improve on the readability for humans and the consistency in interpretation by browsers while increasing the support for the latest multimedia and graphics [8]. New features included in HTML5 have opened up countless possibilities for web developers in many different fields.

A new part of the ThinLinc product is an HTML5 client which is based on an open source project called noVNC, which uses HTML5 features [6]. To distinguish the different ThinLinc clients from each other the clients which are applications running directly on an operating system are referred to as the native ThinLinc clients while ThinLinc's HTML5 client which is running in a browser is mostly referred to as the HTML5 client.

![Figure 1: The HTML5 client and the native ThinLinc client](image)
Figure 1 shows different types of platforms such as the common desktop operating systems Windows and Linux as well as Android or iOS devices on which the HTML5 client can be used, as well as a native client, to connect to the ThinLinc server. The terminals with the Windows and Linux icons represent Windows and Linux computers while the tablet with the Apple and Android icons represent iOS and Android devices. On the server side the computer with a Linux icon represents a Linux server computer. The lighter boxes next to the client devices and the server computer contain applications for the unit in question. The lines between the applications represent connections from the ThinLinc clients to the ThinLinc server.

The HTML5 client is currently being shipped as a part of the ThinLinc product but is marked as “Technology Preview” due to the fact that it is has been lacking important features which are available for the rest of the ThinLinc clients [6]. Cendio’s short term goal for the HTML5 client is to replace their Java applet client. The Java applet client is another ThinLinc client which runs in a browser. As by Oracle design, the Java applet client requires a Java plug-in in the browser [9].

Cendio aims for the HTML5 client to be functional in browsers supporting the HTML5 features WebSocket and Canvas. WebSocket is a technique which, as defined by W3C, allows two way communication channels over Transmission Control Protocol, TCP, sockets [10]. WebSockets will be described further in section 2.3.1. Canvas is a technique which, as described by W3C, allows for dynamic rendering of shapes and images [11]. The browsers and their earliest version which should be supported are listed below in Table 1. Note that the mobile versions of each of those browsers should be supported as well. The browsers in this list are henceforth called the supported browsers.

<table>
<thead>
<tr>
<th>Browser</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer</td>
<td>10</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td>11</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>16</td>
</tr>
<tr>
<td>Safari</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1: The supported browsers

Prior to the thesis the HTML5 client was functional for the supported browsers and supported most core functionality which was required in order for a remote desktop client to be usable: Graphics output, user interface, infrastructure and mouse and keyboard input [6]. Keyboard input was operative in all supported browsers except for in Google Chrome on Android. The most prominent features it lacked were support for audio, printing, local drive access, alternative authentication methods, access to smart card readers and serial ports. Implementation of these features is the main objective of this thesis. The core functionality of the HTML5 client is the base for the implementation that has been done in this thesis project.
1.2 Problem description

The problem in focus for the thesis project was that the limitations of the HTML5 client compared to the native ThinLinc clients in terms of functionality are unknown. The question was whether the HTML5 client can reach the same level of functionality as the native ThinLinc clients. The provider of the thesis, Cendio, is interested in analyzing to what extent any HTML5 based client could substitute their native clients in the future.

As mentioned the HTML5 client already had most core functionality. Thus the goal of this thesis was to investigate the possibility to realize other important features. The features were important and investigated because such features would allow the user to get a better remote desktop experience. The investigation includes exploration and comparison of possible technical solutions, implementation and testing. The different features were all such that they are implemented in the native ThinLinc clients but not implemented at all or lacking in some way or another in the HTML5 client. The status of each feature was defined by how far it’s functionality is from being at the same level as in the native ThinLinc clients. The list of features and their status in the HTML5 client is shown in Table 2. There are two categories of features, the main features are numbered 1-3 and the additional features are numbered 4-7, the differences are explained in the two sections below. In the table the features are ordered by priority where the highest priority goes first.

Table 2: List of features

<table>
<thead>
<tr>
<th>No.</th>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Keyboard input</td>
<td>Not fully functional in Google Chrome</td>
</tr>
<tr>
<td>2</td>
<td>Redirection of audio</td>
<td>Not implemented</td>
</tr>
<tr>
<td>3</td>
<td>Local printing support</td>
<td>Not implemented</td>
</tr>
<tr>
<td>4</td>
<td>Access to local drives</td>
<td>Not implemented</td>
</tr>
<tr>
<td>5</td>
<td>Kerberos authentication</td>
<td>Not implemented</td>
</tr>
<tr>
<td>6</td>
<td>Access to smart card readers</td>
<td>Not implemented</td>
</tr>
<tr>
<td>7</td>
<td>Access to serial ports</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>

1.2.1 Main features

The first three items in the list of features for the HTML5 client are the most important ones for Cendio. The goal was to compare possible solutions, implement a solution, test the solution and evaluate the result for the first three items. This section introduces these three main features.

**Keyboard input**

A core functionality for a remote desktop client is, according to Lubbers et al, to be able to send keyboard input from the local computer to control the desktop or application from the remote computer which is being viewed [5]. The problem for keyboard input in the HTML5 client was that the existing implementation was not functional in Google Chrome, one of the supported browsers, on the Android platform with an on-screen keyboard. The task was to implement a solution which works in all the supported browsers.
Redirection of audio

Running an application from a remote computer through a remote desktop client on the local computer depends on that the user gets the necessary output from the application. In many cases the only necessary output is the graphic image, but obviously there are also a lot of applications which rely on being able to transmit sounds to the user. A very important feature for a remote desktop client is thus to be able to redirect audio from applications on the remote computer to the speakers on the local computer. The HTML5 client did not support audio and the task was to implement such support.

Local printing support

When viewing a document in a remote desktop session, the user might want to print it using a printer located close to where he or she is located. Normally only printers connected to the remote computer, the server, is available from the printing dialogues within the session. When connecting to a ThinLinc server a new printer queue is presented to the user in the printing dialog – thinlocal. The native ThinLinc clients use the thinlocal queue to be able to print to locally. The HTML5 client did not support thinlocal to print to local printers and the task was to implement such support.

1.2.2 Additional features

The intention with the last four items in the list of features was to investigate to what extent the functionality of each feature will be limited, if the feature is implemented in the future, compared to the native ThinLinc clients functionality. The conclusion of this investigation was based on acquired knowledge during the implementation of the main features and the tests of these. This section introduces these four additional features.

Access to local drives

Access to local drives such as an USB flash drive or the user's local home directory within the remote desktop session is something which can be very useful. The native ThinLinc clients solve this by exporting and mounting chosen directories on the remote computer [6]. The HTML5 client did not support any access to local drives and the task was to investigate related limitations for a possible future implementation.

Kerberos authentication

As stated by Lubbers et al, when connecting to a new remote desktop session the user must authenticate herself [5]. There are many different authentication methods. One such method by MIT is Kerberos, this method allows the user to authenticate herself towards a trusted third party, the Kerberos server [12]. Only the user and the Kerberos server will see the provided credentials. If the authentication is successful the Kerberos server will provide the user with a ticket which grants access to the service, the remote session. The ThinLinc server accepts this ticket and the user is logged in. The HTML5 client did not support using Kerberos as an authentication method and the task was to investigate related limitations for a possible future implementation.
Access to smart card readers

A smart card reader, as specified in the patent by Landry et al, allows a computer to read smart cards [13]. Access to a smart card reader, plugged in to the local computer, within a remote desktop session would allow users to encrypt files, authenticate herself in applications and more. All this could be done by using their smart card within the session as seen in Cendio's product [6]. The HTML5 client did not support access to smart card readers and the task was to investigate related limitations for a possible future implementation.

Access to serial ports

Access to serial ports on the local computer within the remote desktop session would allow the user to use older printers in the session. Dial-up modems and other hardware using the serial port interface such as certain USB flash drives, would also be made accessible by this feature. The HTML5 client did not support access to serial ports and the task was to investigate related limitations for a possible future implementation.

1.3 Goal

The ultimate goal for this thesis was to come to a conclusion whether an HTML5 client for remote desktops could replace the native ThinLinc clients. Since remote desktop solutions often compete with regular desktops, Cendio will always strive to have as few limitations as possible in their solution compared to the regular desktop experience. Cendio can not, therefore, replace the constantly developing native ThinLinc clients with an HTML5 client if there are limitations in the HTML5 platform. The limitations in question would be such that they prevent the HTML5 client from reaching the same functionality level as the native ThinLinc clients. These limitations will affect all future versions of the native ThinLinc clients.

The list of features which was provided consisted of three items which were implemented as a part of this thesis (the main features) and four which were not (the additional features). The goal was to evaluate the results of each main feature. For the additional features, the goal was to identify potential limitations of future implementations.

The hypothesis for this thesis is that there are limitations in the HTML5 platform which prevents any HTML5-based remote desktop client to reach the same level of functionality as the native ThinLinc clients. The hypothesis included that the features numbered 4, 5, 6 and 7 were subject to such limitations. The hypothesis also included that the features numbered 1, 2 and 3 would be possible to implement with adequate performance. This meant that they would pass the same set of acceptance tests as the native ThinLinc clients.
1.4 Methodology

This thesis tested the hypothesis by analyzing the limitations within the list of features through literature studies and experimental development. Experimental development includes implementation, testing and evaluation. The testing was done in two parts, platform tests and acceptance tests. The acceptance test criteria which will be detailed in section 5.1.2 was established from the state of the features in the native ThinLinc clients. An analysis was conducted regarding the main features. In particular, possible solutions was explored and compared. The most suitable solutions were implemented and tested. Literature regarding the additional features was studied and it was explored whether each additional feature would be possible to implement in the future. An interview with a professional in the field was also carried out. The results of the interview were used both to provide technical insight for the background and to provide perspective at the evaluation stage.

The problem was approached with the aim to systematically integrate knowledge acquired during the past studies, to demonstrate knowledge and understanding in the main field of study and to be able to assimilate the contents of the relevant literature and relate the work to this. Specifically, the summarized tasks in chronological order as they were carried out are as follows:

- Researched possible solutions for the main features based in the HTML5 standard, other relevant theses, articles and expert statements.
- Compared and evaluated solutions for each of the main features in order to decide which one to implement for each feature.
- Implemented and tested a solution for each main feature.
- Evaluated the results of the tests for each of the main features.
- Researched possibilities for the additional features based in the HTML5 standard.
- Concluded the possibilities of the additional features.

1.5 Scope and limitations

The thesis was limited to only consider the functionality level within the list of features for comparison, other functionality was not considered. Since it is expected that the native ThinLinc clients will gain additional functionality in the future the comparison aspect in this thesis was limited to the state of the native ThinLinc clients as of when the thesis work started, the fall of 2013.

Solutions which required additional software on the client side was not considered since one of the main points of using an HTML5 based client is to avoid the need of such extra software. The only software required is a browser that supports the HTML5 standard. Support for browsers not present, or older than the browsers in the list of supported browsers (see Table 1) were not taken into account when developing or testing. Network security was not a main focus of the thesis and was thus not always considered when comparing and implementing features. The work was focused on the ThinLinc HTML5 client and no other HTML5-based remote desktop clients were investigated.
1.6 Thesis structure

The rest of this document is structured as follows:

- Chapter 2 describes the background of HTML5 concept, performance within the context, and technologies and tools used.
- Chapter 3 describes the main features in detail and presents the requirements for each feature.
- Chapter 4 presents the process of development of extended functionality for an HTML5 client for remote desktops. This includes the comparison of solutions, the architectural choices, and the implementation of each chosen solution.
- Chapter 5 describes the testing of each implementation and evaluates the test results and future development.
- Chapter 6 discusses the conclusions for the thesis and the prospects for future work within the subject.
Chapter 2: Technical background

Chapter two will explain HTML5, give brief information about performance in the context of browsers as well as technologies and tools used during the thesis project. These constitute the background to the work and are essential to explain before going further with the features which were investigated.

2.1 The concept of HTML5

The HyperText Markup Language, HTML, is, according to W3C, the representation language of the web [7]. HTML5 is the new version of HTML. Lubbers et al wrote in 2011 that the concept HTML5 could be used in different ways [8]. HTML5 is firstly a specification that is being developed by the W3C and WHATWG. HTML5 has not been finalized at the moment of writing this thesis in the fall of 2013. It is constantly being developed and extended with new features.

HTML5 has also become a commonly used term which sometimes covers more than the narrow specification that W3C provides. The term HTML5 can in some cases refer to new technologies which may only be proposed to be included in W3C’s HTML5 specification. Further use of the HTML5 term in this document refers to what can be found in the HTML5 specification by W3C. A new important feature which is included in the HTML5 specification is WebSocket. This feature is central to this thesis. The WebSocket application programming interface (API) is described below in section 2.3.1.

2.2 Performance

Performance in a browser application is most of the time going to be lower than what can be achieved in a native application. This is because native applications can take direct advantage of the system’s UI functions, multi-threading and hardware acceleration, and do not have a browser in between which slows things down due to the lack of the aforementioned options. Rodriguez-Alsina et al found in 2012 that using native applications can give a better user experience due to this [14]. Svedestedt also concluded, in 2013, that JavaScript engines can not compete with native applications [15]. Martin consolidate the assertion by declaring that the case is the same for noVNC now at the time of writing [16]. It also is important to note that when comparing performance there might be differences between browsers since they have different JavaScript engines with different performance. There are performance tests for browsers which measure these differences. Due to these browser differences, any performance related tests for implementations in this thesis was conducted with this taken into consideration.

2.3 Technologies and tools

This section presents two essential technologies which were used during the development phases of this thesis project as well as the six most important tools. The described technologies are the WebSocket API and VNC. The introduced tools are the editor Emacs, the version control systems Git and Subversion, the JavaScript Console in the browser, the Android Debug Bridge plug-in and the Safari Web Inspector.
2.3.1 WebSocket API

Lubbers et al describe the interface called WebSocket as a new communication feature in the HTML5 specification which defines a full-duplex communication channel which uses a single socket [8]. Use of WebSocket gives a big improvement compared to old Asynchronous JavaScript And XML (Extensible Markup Language), AJAX, solutions combined with XMLHttpRequests. XMLHttpRequest is an API detailed by W3C that provides a way for transferring data from and to a client and a server using scripts [17]. Use of the WebSocket interface allows highly interactive networked applications to run well in a browser through providing a live stream of data which, like TCP, work in both directions. Application level protocols such as the Remote Frame Buffer protocol (RFB) can be, as with TCP, run over WebSocket but, unlike TCP, WebSocket does not connect to an Internet host and port but connects to URLs (Uniform Resource Locators) instead.

2.3.2 Virtual network computing

Virtual Network Computing (VNC) is a widely used cross platform system for remote desktop computing which uses the RFB protocol. VNC servers provide applications, data and entire desktops to the VNC clients, and wherever a desktop is accessed the state and the configuration is preserved from the last session.

The underlying technology is a simple remote display protocol which makes VNC very powerful. A remote display protocol is a specific set of rules for data transfer that make remote desktops possible. Richardson et al write that VNC has great performance due to the flexibility in encoding schemes and due to the ability to counter factors such as limited network bandwidth, fast client drawing speed and low server processing speed [18]. Encoding schemes can for example be used to send only the parts of the screen that was changed since the last screen-update. This would reduce bandwidth usage.

2.3.3 Editor

For programming no full blown IDE was used. Instead the editor Emacs was used. Emacs is an extensible, customizable, self-documenting real-time display editor produced by the GNU Project [19]. JavaScript, HTML and Python were the languages that were used and Emacs has built-in modes that support development in these programming languages. JavaScript is described as an interpreted programming language that is a part of web browsers by Flanagan et al [20]. Python is a general-purpose programming language by Python Software Foundation (PSF) which is often used as a scripting language [21].

There are modes which Emacs can enter which are defined by the type of the file opened [19]. These modes are referred to as major modes. Only one major mode can be activated at the same time. The major modes for JavaScript, HTML and Python were automatically activated when files of the formats .js, .html, and .py were opened. The different major modes provide syntax highlighting, automatic indentation and special editing commands appropriate to the corresponding programming language. There are additional modes in Emacs, the minor modes. Emacs can use multiple minor modes simultaneously. Minor modes can for example be used for different indentation styles within the same programming language.
2.3.4 Version control systems

Version control, also known as revision control and source control, gives a developer a logical way of organizing and controlling changes made to the code. Changes are usually referred to as revisions. Each revision is identified by a number or letter code. The revision can include various metadata such as a time-stamp, the author who made the change, and comment. The purpose is to give a structured way to make comparisons, merges and roll-backs of the revisions. The concept of branches within the version control context refers to different modifiable instances of an object which exist in parallel. Version Control Systems (VCS) are applications which give the aforementioned benefits of version control. Commands are provided for making comparisons, merges, roll-backs and more.

The VCS used for the noVNC project, which the HTML5 client is based on, is Git. Git is, as written by Chacon and Hamano, a distributed VCS designed with speed and efficiency in focus [22]. The noVNC project is hosted on GitHub, a web-based hosting service for projects using Git. For version control for Cendio's product ThinLinc, the VCS called Apache Subversion (SVN) was used. SVN is, according to Apache, a centralized VCS which is characterized by its simple model and wide usage [23]. Most of the produced code was committed to ThinLinc's SVN repository.

2.3.5 Browser tools

During development the browser Google Chrome were primarily used. Google Chrome has a built-in JavaScript Console which was used for troubleshooting and helpful logging messages in real time [24].

For testing and debugging on the Android device Nexus 7 a plug-in by Google, called Android Debug Bridge, ADB, for Google Chrome was used. There is also a command line tool with the same name which runs in a terminal on Linux [25]. The browser requires that the ADB plug-in is installed as well as the ADB command line tool running. The Android device shall be connected via a USB cable to the platform on which ADB is running. The ADB command line tool was started by running `# adb -logcat`.

For testing and debugging on the iPad device the Safari Web Inspector interface was used. The Web Inspector is, as defined by Apple, a built-in web development tool in the browser Safari which allows debugging output and inspection of separate HTML elements in real time [26]. It had to be enabled using the Settings app on the iPad, under advanced Safari settings Web Inspector. The iPad was connected to a Mac computer with a USB-Lightning cable. In Safari on the Mac under the tab called Developer the iPad is found along with the open web pages in the iPad's Safari.
Chapter 3: Technical issues in a remote desktop solution

The third chapter describes the technical issues in focus for the three main features in detail. Through explaining how the corresponding features are realized in the native ThinLinc clients the requirements for each main feature are identified. The three main features are keyboard input, redirection of audio, and local printing support.

3.1 Keyboard input

The purpose of a keyboard input feature is to allow the user to send keyboard input to the remote session and in that way control the applications within the session. As mentioned, prior to the thesis the state of the keyboard input feature in the HTML5 client was that it worked on most platforms. The feature was functional, as mentioned, on all platforms except for Google Chrome, on Android devices. This behavior is caused by the fact that the browser did not produce the key events from the default on-screen keyboard, the most commonly used type of keyboard on Android. The key events were generated correctly by the keyboard and work without any major problems in the other supported browsers.

Since the ThinLinc server uses VNC and therefore the RFB protocol the server requires a key event which is a keyPress or keyRelease containing a keysym value. As hinted by their names, a keyPress event is expected when a keyboard key is pressed. Cendio details that in the same sense, a keyRelease event is expected when the key is released [27]. A keysym value is a key symbol defined by the X Window System. The X Window System, X11 or simply X is a common windowing system on Linux operating systems. There is a unique keysym value for each character.

On a normal keyboard, when a key is pressed, a scan code is generated by the keyPress event. The scan code is then translated to a key code which eventually can be translated to a keysym value. The scan code corresponds to the position of the physical key on the keyboard and is independent of keyboard layouts. The key code represents the physical key but is dependent of the current keyboard layout. The keysym corresponds to the symbol which was generated. On a keyboard with Swedish layout, pressing \textit{SHIFT}
and 7 at the same time will generate two scan codes and two key codes representing the two keys but only one keysym value. The keysym value is 0x002F which represent a forward slash.

In order to bring up the on-screen keyboard on Android touch devices in the browser the focus has to be set on an input field. Such an input element existed in the solution for the HTML5 client which was in place prior to this thesis. A button at the top of the screen exists in a control-bar which brings up the keyboard, this is depicted in Figure 2. When this button is pressed the focus is set to a hidden input element, this causes the on-screen keyboard to show. The input element is hidden by having the width and height properties of the element set to zero. In Google Chrome on Android devices where key events are not generated the text input will be written into the hidden input element. That text input will generate input events instead of key events.

The native ThinLinc clients are not subject to specific limitations from any browsers because they run directly in the operating systems. Because of this, this issue with keyboard input is not found there. Google Chrome is part of the supported browsers (see Table 1) for the HTML5 client. Thus the requirement was that keyboard input should be fixed in Google Chrome.

### 3.2 Redirection of audio

The purpose of a redirection of audio feature is to allow applications in the remote session to output sound to the local computer's speakers. Prior to the work made in this thesis the HTML5 client did not support audio redirection in any form. The goal was to implement a solution which would function on the same level as the audio redirection feature in the native ThinLinc clients. The native ThinLinc clients come with a built-in PulseAudio server which handles the sound playback. In the context of sound, the sound server is the part which interface with the sound card. This implies that the applications have the client role. In the context of remote desktops the applications are on the server computer and the sound card is on the client computer. This explains why there is a sound server is on the client computer.

The organization freedesktop.org defines PulseAudio as a sound server system, a proxy for sound applications [28]. Network transfer of audio and advanced operations on the sound are made possible by a PulseAudio server as depicted in Figure 3. The lines between the different parts illustrate the audio data flow. Applications use the PulseAudio library libpulse to interface with the PulseAudio server. The PulseAudio server in turn interfaces with both the network stack and the hardware drivers for the sound card. Over the network other PulseAudio servers and applications on the network can get audio data. This data goes from the network stack to the hardware network adapter out on the network.

![Figure 3: Sound with PulseAudio](image)
The hardware drivers for the sound card can transmit the sound to the speakers. As mentioned, PulseAudio servers can redirect audio data to other PulseAudio servers. Thus, there are PulseAudio servers which do not interface with any sound cards. Such PulseAudio servers are merely *middlemen*. The PulseAudio server on the server computer in a ThinLinc context is such a middleman. PulseAudio servers can accept input from applications or other PulseAudio servers. This input can be in multiple protocols and different formats. PulseAudio servers can output in multiple different protocols and formats as well [28]. Each protocol and format has its own module. Modules for input are called sources. Modules for output are called sinks. PulseAudio is designed for Linux but has been ported to other operating systems such as OS X, Windows and Solaris.

There is a PulseAudio server on the client computer, as mentioned. There is also a PulseAudio server on the server computer as depicted in Figure 4. The red lines represent the audio data flow. When installing, the ThinLinc server checks that a PulseAudio server is available on the server computer. The ThinLinc server then opens up a Transmission Control Protocol port, a TCP port. The ThinLinc server then configures that PulseAudio server to transmit data to the opened TCP port. Thus, the applications on the server computer which interface with libpulse transmit audio data over the TCP port. The green lines represent the actions by the ThinLinc server to open the TCP port and to configure the PulseAudio server. The ThinLinc client’s PulseAudio server on the client computer listens to that TCP port on the server computer and get audio data over the network through a Secure Shell (SSH) tunnel. The built-in PulseAudio server on the client computer can play the received audio data on the local speakers.

Next, the problems that have to be solved for a client connected to the PulseAudio server via a browser are described. One problem was discovered when investigating the use of the above technology for our purposes. The PulseAudio server which is used in the native ThinLinc clients uses system API’s to access the sound card and the speakers. The system API’s are different depending on the operating system of the client computer. On Linux computers the API that is used to interface with the sound card drivers is called Advanced Linux Sound Architecture, ALSA. The problem is that a browser does not have direct access to system API’s. In order to be able to use a similar solution in the HTML5 client one would have to implement a new audio server in JavaScript which uses browser API’s to play the audio.

When investigating the current web server in the ThinLinc server another obstacle was encountered. That web server which the HTML5 client connects to is called tlwebaccess. The problem was that tlwebaccess could only handle VNC communication, that is data following the RFB protocol format. To the native ThinLinc clients the server can communicate in a multitude of different protocols, each protocol over its own SSH tunnel. The tlwebaccess service would thus have to be extended to handle audio communication as well. The ThinLinc server works in such a way that multiple tlwebaccess processes would run if communication over multiple protocols were to be implemented. It is similar to how it runs multiple SSH tunnels for communication with the native ThinLinc clients, one SSH tunnel for each type of communication [6]. The VNC...
communication to and from the HTML5 client and tlwebaccess is handled via a WebSocket. It would be preferable to use WebSockets for audio communication as well and in this case the tlwebaccess service would have to be able to check which type of data it is transferring over the WebSocket connection in order to fetch data from the correct port. For audio communication the TCP port on the server computer, which the sound applications are playing audio on, is the one for tlwebaccess to fetch data from. After the data has been received in tlwebaccess it is sent over the network using the WebSocket and on the other end the HTML5 client is receiving that data.

Multiple different solutions could be used thanks to the flexibility of the server-side. This is due to the fact that PulseAudio has different sinks. The PulseAudio server on the server computer can output audio in multiple formats and protocols. Thus the solution on the client-side, in the HTML5 client, could be set up in many different ways. The requirement was to implement support for audio in the remote session. Following the architecture found in the native clients was not strictly required.

3.3 Local printing support

The purpose of a local printing feature is to allow the user to print data from the remote session to a local printer. No local printing support was present in any form in the HTML5 client prior to the work performed in this thesis. The goal was to implement a solution which would function on the same level as the local printing feature in the native ThinLinc clients. The native ThinLinc clients have a built in Line Printing Daemon (LPD) server called service_lpd. The LPD server communicates with printers by using a protocol by the same name. The Internet Engineering Task Force (IETF) defines LPD as a network protocol intended to be used for submitting print jobs to a remote printer but can also be used to transfer a print job from one computer to another [29].

As mentioned in section 1.2.1, the ThinLinc server sets up a new printing queue called thinlocal in the remote session which is available for printing. The ThinLinc server configures a back-end for thinlocal in the Common Unix Printing System (CUPS). This back-end converts the print job to the Portable Document Format (PDF). The conversion is handled by a component on the operating system the ThinLinc server is running on. Thus, any print job sent to thinlocal is converted to PDF. This means that over the network the LPD protocol is transmitting the print job in PDF format. When printing dialogues are opened in any remote session from a ThinLinc server, the thinlocal option is available as depicted in Figure 5. Even prior to the work performed in

Figure 5: A print dialogue showing the thinlocal option
this thesis, the thinlocal option was available in printing dialogues seen in the HTML5 client. What was missing, in order to support local printing, was an interface talking to thinlocal. Nothing happened when the user chose to print to thinlocal. Since the ThinLinc server protocol was already established, that protocol would have to be used to communicate with. Thus, a LPD server, similar to service_lpd, would have to be implemented for the HTML5 client. Implementing a LPD server implies the usage of the LPD protocol.

As LPD and thinlocal have been described, below follows how the native ThinLinc clients handle received print jobs. When received on a Linux, Solaris, or Mac client computer the PDF is converted to PostScript (PS) before being printed. When received on a Windows client computer it kept as PDF. It is possible to automatically print a file after receiving it at the local computer. If the file is automatically printed the user does not have to see a printing dialogue both in the remote session and on the local computer. The native ThinLinc clients store the print file as a temporary file on the local file system. When the print is completed, the temporary print file is deleted.

When implementing a solution for local printing support in the HTML5 client a few points would have to be investigated. The first one being whether a print can be initiated through JavaScript and if it is possible to do this without showing a printing dialogue. The second one being how to store the print file after receiving it from the server, from thinlocal. Similar as for redirection of audio it would also be required that a new WebSocket connection was made for the printing communication. The tlwebaccess service would again have to be extended to handle the printing communication and the process would have to have a way to detect which type of data it was transferring in order to fetch data from the correct port. The data fetched from the printing port on the server computer would be sent over the WebSocket to the client.

Compared to the case for the redirection of audio feature this feature is more bound to one architecture. This is due to the fact that the server-side does not open up for different possibilities. To interface with thinlocal a printing server speaking LPD would, as mentioned, have to be implemented. The requirement was to implement support for local printing, and thus the requirement is reduced to implementing a LPD server.
Chapter 4: Development of extended functionality

Chapter four will start by outlining the comparisons of different solutions for each main feature. Furthermore the architecture of the chosen solutions are presented together for a good overview of the resulting system. Finally the implementation process and the choices made for each implemented solution are described.

4.1 Comparison of solutions

The comparison of the solutions was based on possibilities in terms of functionality. An estimate of how much time each solution would require was made. The time estimate was based on the author's previous experience. It was determined whether the implementation of a solution would be feasible within the scope of the thesis. The goal was to choose a solution whose result was close in level of functionality, compared to the corresponding feature in the native ThinLinc clients. One solution for each main feature should be chosen for implementation. The following three sections describe this comparison process individually for each main feature.

4.1.1 Comparison of solutions for keyboard input

Two different solutions were found to solve the keyboard input problem:

- Special solution for Google Chrome
- New on-screen keyboard for the HTML5 client

The first solution is to use the input events which were generated when there were no key events. The input events could be caught from the hidden input field which was given focus to bring the on-screen keyboard up. As mentioned, the key events did not get produced when using the on-screen keyboard in Google Chrome on Android devices. In the case when the key events does not get generated, the input from the keyboard is written to the hidden input field. The input events could be parsed to find which characters were written and through that generate the appropriate keyPress events manually to send to the ThinLinc server on the remote computer.

One advantage of such a solution was that aside from being easy to implement it would not require a complete restructuring of the present code. An extensive rewrite was not desirable, since the problem was specific to one platform. Another benefit was that using input events allows for the code to support gesture typing or “swipe” and other types of input which generates multiple characters at a time. Google gives developers the advice to stop expecting normal key events on Android. Google came with this advice since it is normal that users input multiple characters at a time [30]. Input of multiple characters simultaneously do not generate normal key events.

A drawback of such a solution was that the code would require special cases for spaces and backspaces since these keys do not always generate input events. As specified by WHATWG, no input events are generated when SPACE or BACKSPACE are pressed and the input field is empty [31]. Another disadvantage was that it would add confusion since the behavior would be different across different browsers. Gesture typing would be available in Google Chrome on Android but not in other browsers.
The second solution was to implement a new specific on-screen keyboard for the HTML5 client. This solution would mean that the built-in on-screen keyboard in Android would not be used. The keyboard would be a part of the HTML5 client, a part of the content displayed by the browser.

The biggest advantage of such a solution is that the developer would have full control and would not depend on different browser implementations. The developer would be able to make a proper solution including for example modifier keys such as Alt, Shift and Ctrl which usually is not present on on-screen keyboards. Modifier keys are useful for a remote desktop user. The desktop and applications within the session are often designed for desktop computers and not touch devices. Such desktops and applications are likely to require modifier keys. An example is a simple terminal application such as xterm. The Ctrl + C command in terminals to stop a running program is common.

A drawback of such a solution is that the user of the HTML5 client would get a different keyboard experience than what he or she is used to. The look-and-feel would be different compared to the regular on-screen keyboards. It would not be possible to take the user's Android settings and preferences into account. Thus, the behavior would be different in terms of configuration since the keyboard would be completely detached from the operating system.

Table 3: Summarized comparison for keyboard input

<table>
<thead>
<tr>
<th>Solution</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Special solution for Google Chrome</td>
<td>• Easy to implement</td>
<td>• Would require special handling for spaces and</td>
</tr>
<tr>
<td></td>
<td>• Would not require extensive rewrite</td>
<td>backspaces</td>
</tr>
<tr>
<td></td>
<td>• Support for gesture typing</td>
<td>• Different behaviour across browsers</td>
</tr>
<tr>
<td>(2) New on-screen keyboard for the HTML5 client</td>
<td>• Full control for developer</td>
<td>• New keyboard look-and-feel</td>
</tr>
<tr>
<td></td>
<td>• Independent of browser differences</td>
<td>• Not being able to use user's settings and</td>
</tr>
<tr>
<td></td>
<td>• Proper modifier keys</td>
<td>preferences</td>
</tr>
</tbody>
</table>

An overview of the comparison, the advantages and disadvantages that have been discussed for the two solutions is found above, in Table 3. When comparing the two solutions for keyboard input it was taken into account that the solution involving implementing a new specific on-screen keyboard for the HTML5 client (solution 2) would be very time-consuming and would not be possible to fit into the tight time frame of the thesis project. The fact that the solution present prior to the thesis was working in all but one of the supported browsers suggested that the case where it did not work was a problem with the browser vendor in question, Google. W3Counter's page view trackers showed that Google Chrome held approximately a third of web browser market share in February 2014 [32]. A specific solution to a problem on a specific platform was suitable. Modifier keys are available from a drop-down menu in the HTML5 client. Such a menu is less convenient for the user than keys directly on the on-screen keyboard, but that advantage of the second solution was still partially neutralized. It was thus decided that the special solution for Google Chrome using input events (solution 1) would be implemented.
4.1.2 Comparison of solutions for redirection of audio

Two different ways of solving redirection of audio were investigated. The first is a streaming solution and the second consisted of implementing an audio server. This section will start by describing the use of the HTML5 subjects that are required for these respective ways of solving the problem. The two HTML5 subjects are the Audio element and the Web Audio API. The description of these two subjects is followed by the actual comparison of solutions for redirection of audio.

The Audio element

The Audio element was added as a part of HTML5 and is used to represent sounds or audio streams. A streaming solution means that the audio would be streamed from the remote computer, the server, and remotely played at the local computer, the client. An Audio element could be used since the audio data would already have been processed on the remote computer. The Audio element is by W3C's design simple and does not provide any processing functions. This element would simply be used to play the stream in the HTML5 client. The processed data would be streamed by a streaming server on the remote computer. The stream would be accessible by a URL over the network. HTML elements such as Audio have attributes. These attributes are specified within the tags of the element. One attribute to the Audio element, the src attribute specifies the URL of the stream source. The Audio element can be used as seen below, in HTML5 code.

```html
<audio src="STREAM_URL;"></audio>
```

Web Audio API

Web Audio API is a new high-level JavaScript API in the HTML5 specification by W3C which can be used for processing and synthesizing audio in web applications. Web Audio API consists of many different objects, but the most important ones are AudioContext and AudioNode. The API is built around routing different AudioNodes, referencing to them, and connecting them. An AudioBufferSourceNode is a reference to an AudioNode and has a buffer to use for small audio data chunks. The speakers are accessed by the DestinationNode in the AudioContext. The DestinationNode also is a reference to an AudioNode. All routing is done in an AudioContext and the different AudioNodes handle processing.

The way of solving the problem which would require Web Audio API consisted of implementing a proper audio server in the HTML5 client which could receive the sound. The audio processing would in such a solution be made on the local computer instead of remotely. The audio element is thus insufficient for this type of solution since it does not provide any processing tools for the audio data. Web Audio API does however provide such tools. Web Audio API would thus be used for any audio server solution to interpret the audio data and play the sound on the local speakers. Assuming the buffer soundBuffer contains a small amount of audio data, Web Audio API could be used to play the audio as seen in the pseudo-code in Listing 1.

```
// Creates a new buffer source which is a reference to an AudioNode.
// @return AudioBufferSourceNode, the new buffer source
AudioContext.createBufferSource()

// Connects the output of an AudioNode to the input of another.
// @param AudioNode destination, the AudioNode to connect to
AudioNode.connect(AudioNode destination)
```
The comparison

Two solutions which would use the Web Audio API way of solving the problem were investigated. Thus, in this section one streaming solution and two audio server solutions will be compared:

- Streaming solution with IceCast2 (Audio element)
- PulseAudio server in the HTML5 client (Web Audio API)
- EsounD server in the HTML5 client (Web Audio API)

The streaming solution required a media streaming server and the one which was inspected was IceCast2. IceCast2 streams audio over the Internet to listeners and is an Open Source project by Xiph.org [35]. IceCast2 was chosen for inspection by recommendation from the company that commissioned the thesis project, Cendio. The PulseAudio middleman server running, which was mentioned earlier, on the remote computer, would be used. Mulfiari et al showed through successful implementation that such a PulseAudio server could, together with GStreamer pipelines and Lame MP3 encoding tools manage and compress the audio signal before sending it to the IceCast2 server [36]. The above process is depicted in Figure 6. The streaming server handles the audio signal and transmits it over the network. A benefit to such a solution would be that it is easy to implement and does not require a lot of programming. The audio processing is done in finished components.

Disadvantages include delay on the sound and high bandwidth usage. Since the data would be streamed there would be no possibilities to synchronize the audio and latency would therefore lead to incremental delay. This solution does not offer any control over the audio processing. This control is missing since, as mentioned, the audio processing is done in finished components. As specified by Xiph.org, the IceCast2 streaming server has a configurable buffer size which allows for adjustments to minimize delay [35]. The delay will however always be a significant factor. Another drawback is that even with the applications in the session is not playing any sounds the silence will be streamed and thus require constant bandwidth which is very wasteful.

Listing 1: Web Audio API functions and example

```javascript
// Starts playing the sound from the node's buffer at the specified time.  
// @param int when, when to start playing the sound

AudioBufferSourceNode.start(int when)
```

Figure 6: The streaming solution
The second option was to use a PulseAudio server. This is how it is done in the native ThinLinc clients. As mentioned such a server would have to be implemented in JavaScript and added to the HTML5 client. The PulseAudio subsystem on the ThinLinc server computer would be relaying the sound over the network using the PulseAudio protocol to the JavaScript PulseAudio server in the HTML5 client.

The biggest advantage of a solution using a PulseAudio server in JavaScript would be the fact that it would be a proper solution. Full control over the audio processing would be available. A PulseAudio server is very advanced and includes a lot of useful features including possibilities to synchronize the audio perfectly. Such a solution would also keep the same architecture as seen in the native ThinLinc clients which would reduce complexity and make maintenance easier. The only disadvantage would be that the work to implement a PulseAudio server would be extremely time-consuming.

The third solution was to implement a EsounD server. Similar to the PulseAudio server solution a EsounD server would have to be implemented in JavaScript and added to the HTML5 client. The Enlightened Sound Daemon, EsounD or ESD protocol is a network compatible audio protocol similar to PulseAudio. The author ymnk described the protocol since no official protocol exist [37]. Red Hat's Lennart Poettering reported that as of 2009 the protocol and the EsounD server will not be developed or supported any more [38]. According to freedesktop.org, the PulseAudio subsystem running on the remote computer could be used with a built-in module to send audio data in the EsounD format over the network to the JavaScript EsounD server in the HTML5 client [39]. The EsounD protocol is not as advanced as the PulseAudio protocol but this brings the advantage of it being easier and less time-consuming to implement. Synchronization options are not available over the EsounD protocol.

### Table 4: Summarized comparison for redirection of audio

<table>
<thead>
<tr>
<th>Solution</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Streaming with IceCast2 (Audio element)</td>
<td>• Easy to implement</td>
<td>• No synchronization</td>
</tr>
<tr>
<td></td>
<td>• Audio processing finished</td>
<td>• No control over audio processing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Incremental delay</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• High bandwidth</td>
</tr>
<tr>
<td>(2) PulseAudio server in the HTML5 client (Web Audio API)</td>
<td>• Control over audio processing</td>
<td>• Time-consuming to implement</td>
</tr>
<tr>
<td></td>
<td>• Synchronization</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Keep architectural style</td>
<td></td>
</tr>
<tr>
<td>(3) EsounD server in the HTML5 client (Web Audio API)</td>
<td>• Control over audio processing</td>
<td>• No synchronization</td>
</tr>
<tr>
<td></td>
<td>• Easier to implement</td>
<td></td>
</tr>
</tbody>
</table>

An overview of the comparison, the advantages and disadvantages that have been discussed for the three solutions is found above, in Table 4. When comparing the three solutions for redirection of audio is was quickly decided that it would have to be one of the two audio server solutions (solution 2 or 3) due to the heavy disadvantages the streaming solution (solution 1) brought. The delay disadvantage of a streaming solution is critical in the context of remote desktops. However, the most important disadvantage of the streaming solution (solution 1) was found to come from using the Audio element. Ideally the HTML5 client would even be able to synchronize the audio,
something which would simply not be possible with the Audio element. Research conducted by Yan et al. showed that any application which could need manipulation or analyzing of audio greatly benefit from the use of an API [40]. The backing from this research and the heavy limitations led to this decision. Due to the fact that the time required to implement a JavaScript PulseAudio server would be impossible to fit in the time schedule of this thesis project that solution had to be discarded as well. Despite being an old and simple protocol it was decided that the EsounD is a functional protocol which fits very well for the purposes of this thesis project, thereby the JavaScript EsounD server was to be implemented.

4.1.3 Comparison of solutions for local printing support

An aspect which was interesting for all solutions for local printing support were automatic printing. In the native ThinLinc clients it is, as mentioned, possible to automatically print the file when it was received on the local computer saving the user from an extra printing dialogue. It was concluded that it was not possible to print a normal PDF from within a browser solely using JavaScript without showing a printing dialogue. A possible solution to this problem was PDFs that were modified to automatically print themselves when opened. In order to modify a PDF it would have to be done once created. The print file was converted to a PDF by a component on the server computer which was not shipped as a part of the ThinLinc server, this modifying the PDF at the point of creation would mean that a new component would have to be added to the ThinLinc server. This component would change the architecture for all the ThinLinc clients since it would not be possible for the ThinLinc server to, at the time of PDF creation, to know if it should be sent to a native ThinLinc client or the HTML5 client. To introduce architecture changes that change the way the native ThinLinc clients work was outside the scope of this thesis and thus this option had to be excluded.

As mentioned, the question which was in focus when finding solutions for local printing support was how to store the print file. The actual printing communication between the HTML5 client and the ThinLinc server was required to be over the LPD protocol. Thus the three different solutions for local printing support were divided upon this factor:

- Store on local computer using the data URL scheme
- Store on local computer using the File System API
- Store on remote computer and present as PDF for download

The first solution was to store the print file in the browser using data URL. The data URL scheme defined by IETF is a way to store data in-line in web pages. The data is stored in the URL. IETF specify that data stored using data URL is stored in computing memory [41]. This implies that this solution would store the print file in the memory of the local computer. Such a solution would require the implementation of a JavaScript LPD server in the HTML5 client. A benefit to using data URL was the control it brought over the data. No layer or obstacle would be in the way. Using data URL implies long URLs for big files. The size limitations brought by this brings the biggest disadvantage to a data URL solution.

The second solution was to use the File System API to store the print file on the local computer. The File System API was a new means in the HTML5 specification by W3C to navigate sandboxed sections of a users local file system [42]. File System API was in fact so new that it was only implemented by Google Chrome in the list of supported browsers at the time of writing. This was shown by tests conducted by Deveria [43]. The File System API would not have the tight size limitations seen in the first solution and would provide quick file access. There would however still be a size limitation, which is decided by the browser. The disadvantage of a File System API solution would be that the solution would not work across all supported browsers.
The last solution was to store the print file on the remote computer, the server-side, and to present it for the HTML5 client similar to how PDF files are normally presented on the web. This solution would allow the user to, when he or she wanted, download the file and then print it. A Python LPD interface would have to be implemented in the tlwebaccess service and the additions required in the HTML5 client would be minimal user interface additions.

The benefits of such a solution would include no limitations on file size and a stable transfer of the file through letting the browser handle it as it normally does with PDF files. The disadvantage would be that it would break the architecture style and the behaviour would be different from the native ThinLinc clients where the file is sent and stored on the local computer, the client.

### Table 5: Summarized comparison for local printing support

<table>
<thead>
<tr>
<th>Solution</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Store on local computer using the data URL scheme</td>
<td>• Easy control over data</td>
<td>• Tight size limitations</td>
</tr>
<tr>
<td></td>
<td>• Keep architectural style</td>
<td></td>
</tr>
<tr>
<td>(2) Store on local computer using the File System API</td>
<td>• Keep architectural style</td>
<td>• Size limitations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Only works in Chrome</td>
</tr>
<tr>
<td>(3) Store on remote computer and present as PDF for download</td>
<td>• No size limitations</td>
<td>• Not same architecture as native clients</td>
</tr>
<tr>
<td></td>
<td>• Stable file transfer</td>
<td></td>
</tr>
</tbody>
</table>

An overview of the comparison, the advantages and the disadvantages that have been discussed for the three solutions is found above, in Table 5. When comparing solutions the big disadvantage of the File System API solution outweighed everything else quickly with its low availability. The choice stood between the first and the last solution. The fact that the first solution kept the architecture seen in the native clients took the upper hand over less limitations on the file size. Ramamoorthy et al writes that reusing design and architecture provides great efficiency and understandability benefits [44]. Additionally, research by Boehm et al conclude that to be understandable the software is required to be both structured and consistent [45]. Having found sufficient support for the approach of keeping the architecture led to the decision, the data URL solution would be implemented.
4.2 System architecture

The ThinLinc product consists of many different parts, but not all of them are relevant to the thesis. This section will focus on the architectural choices of the components which are relevant. The parts which were modified in the work done in the thesis are highlighted in Figure 7. The ThinLinc server side corresponds to the ThinLinc software on the remote computer, the server. The ThinLinc client side corresponds to the ThinLinc software on the local computer, the client. Each bubble on the ThinLinc server side corresponds to significant components in the ThinLinc server software. The four ThinLinc icons on the ThinLinc client side correspond to the four main ThinLinc clients. The four ThinLinc clients shown are the HTML5 client, the Linux client, the OSX client, and the Windows client.

The modified parts are the HTML5 client and tlwebaccess. These two are also the two relevant components for the thesis. The server side component tlwebaccess is, as mentioned, the web server used for the HTML5 client. All communication between all the ThinLinc server components and the HTML5 client goes through tlwebaccess. The protocols used for the different types of communication between tlwebaccess and the HTML5 client is shown in Figure 8. The VNC protocol is used for video, mouse, and keyboard input. The chosen solution for the redirection of audio resulted in the EsounD, or ESD, protocol being used for audio. Lastly, as mentioned, the LPD protocol is used for printing communication.

*Figure 7: An overview with parts modified in the thesis highlighted*

*Figure 8: The network protocols used for different features*
Compared to the architecture seen in the native ThinLinc clients the chosen solutions were similar in many ways. This is good since, as mentioned, similar architecture would reduce complexity and make maintenance easier. The biggest difference was in the redirection of audio solution where the audio data is transferred using the EsounD protocol instead of the PulseAudio protocol.

The protocols which are used for the different features have been mentioned. From where the data over these protocols origin and how it is handled on the other end is however also relevant. An overview of the different technologies used for these purposes in different layers are shown in Figure 9. The green, red and blue lines represent the data flow. The hardware on, or connected to, the client computer, which are the end point of the data on the client side are the monitor, the mouse and keyboard, the speakers, and the printer. In the web browser on the client computer, the technique used for the image is, as mentioned, canvas. The technique used for the audio is, due to the fact that an audio server solution was chosen, Web Audio API. Lastly, the print job, when presented in the browser, is in PDF format. Over the network, the data in all three of the mentioned protocols use WebSocket. On the server computer, the VNC data goes through tlwebaccess to and from the Xvnc component. Xvnc is the VNC server component of the ThinLinc server software. The EsounD data does originate from the PulseAudio middleman server. This PulseAudio server is, as explained earlier, used with an EsounD plug-in in order to output data in the EsounD format. The LPD communication goes through tlwebaccess to and from thinlocal and the CUPS back-end.

Overall code architecture and design followed the existing architecture which were in place. The existing architecture was built using different design patterns. Modules, which is a structural design pattern was used as well as the behavioural pattern called states. The module pattern is defined by using modular programming. Zimmer writes that modular programming emphasizes on separating functionality into modules [46]. The aim with using the states pattern is to divide different behaviour of a routine into states. The different states would be a way of distinguishing the changes of this behaviour. These design patterns was used in the new code that was produced and fit well into the overall structure.

Figure 9: The technologies and protocols used in different layers
4.3 Implementation of selected solutions

The implementation process only concerned the main features. Depending on the feature being developed and the nature of the code, all programming was done either in Cendio’s internal repository specifically for ThinLinc or in the open GitHub repository for noVNC. Due to extensive previous experience in working with ThinLinc all of the practices and tools required were already familiar, thus no learning phase was needed. The aim was to, with the written code, match the coding style of any already present code especially when modifying a file written by someone else. The code was to be kept modular and no violations of any logical layering or present APIs was intended to be done. The implementation resulted in a total of 1653 lines of added code. All this code was written by the author of the thesis and as a product of this thesis project.

The implementation of the redirection of audio feature started with modifying tlwebaccess to support different types of communication aside from the RFB protocol. Since this modification would be required for the local printing feature as well, a generic solution was implemented. The four sections below describe the implementation process for the keyboard input feature, the modifications to tlwebaccess, the audio feature, and the local printing feature.

4.3.1 Implementation for keyboard input

The implementation of the keyboard input feature started with identifying a way to simulate the missing key events. As mentioned, the RFB protocol required a key event containing a keysym value. The pseudo-code seen in Listing 2 describes how to get the keysym value of the last character in the hidden input field.

```java
// Returns the keysym for the character // at the specified index // @param int index, the index of the // character // @return String, the keysym value String charCodeAt(int index)

String key;
key = hidden_input.charCodeAt(len-1);
```

Listing 2: Finding the keysym value

Once the keysym value was obtained a key event had to be generated, this was done through the usage of an existing function in the RFB module. This function generates the correct key event for the specified keysym value. A pseudo-code signature of the function is found in Listing 3.

```java
// Generates a key event for the // character with the specified keysym // @param key, the keysym of the // character sendKey(String key)
```

Listing 3: Generating the key event
When the two key functionalities above were established the remainder was to implement handling of special cases such as those needed for SPACE, BACKSPACE, and gesture typing. The result with functioning gesture typing is shown in Figure 10. Below follows a brief explanation of how these special cases was resolved.

A SPACE character alone was not considered as input and did not generate any input event because of that. This was solved by always considering the caret position in the hidden input field. If the caret position changed but no input event came, that meant that a SPACE was written. Additionally, a BACKSPACE on an empty field would not do anything since there would not be anything to remove. This was solved by never allowing the hidden input field to be empty, there would always at least be one character in there which a BACKSPACE could remove. Gesture typing was simply supported by considering input of multiple characters at a time and using a loop. The implementation of the keyboard input feature was thus finished and input worked in browsers where input events were produced instead of key events. The pseudo-code in Listing 4 summarizes the logic described above for the special cases.

```java
// Move the caret position in the hidden input field to the end
hidden_input.setCaretPosition(hidden_input.value.length);

void function handleInputEvent(event) {
    String input = event.target.value; // The new input to the field
    int diff; // The change in length of the text in the hidden input field

    // Set the diff to the largest number between
    // the length of the input and the caret position
    if (hidden_input.getCaretPosition() > input.length) {
        diff = hidden_input.getCaretPosition();
    } else {
        diff = input.length;
    }

    // Check if something was removed or if there is new input
    if (diff < 1) {
        // generate key event for BACKSPACE...
    } else if (diff > 1) {
        for (int i = diff - 1; i > 0; i -= 1) {
            // Trailing white spaces are not considered as a
            // part of the string and are therefore undefined.
            if (input[diff - i] !== undefined) {
                // generate key event for the character...
            } else {
                // generate key event for SPACE...
            }
        }
    }

    // There has to always be text in the hidden input element
    hidden_input.value = "_";
}
```

Listing 4: How the special cases for the input events were handled
4.3.2 Modifications to tlwebaccess

As mentioned, modifications to tlwebaccess were required for the audio and local printing features. It was earlier outlined that multiple tlwebaccess processes will be running, one for each type of communication, each with it's own WebSocket. Each tlwebaccess process had a port it needed to fetch data from or send data to. The HTML5 client, at the client computer, was one endpoint of the port. The other endpoint of the port was a tlwebaccess process. One process for the redirection of audio feature and another process for the local printing feature. A utility function available in the ThinLinc server software (present prior to the thesis) could be used to get the port corresponding to specific types of communication. To be able to use that function, each tlwebaccess process was required to be able to establish which type of communication it was handling. To achieve this, the HTTP headers in the received data was parsed for the `WebSocket-Protocol` header. This specific header was interesting due to the fact that when opening a WebSocket it is possible to include an array called `protocols`. This array was thus used to specify which protocol the communication would follow. A pseudo-code signature of the utility function `get_tunnel_port` is found in Listing 5.

```java
// Returns the output port corresponding to the desired SSH tunnel
// @param String tunnel, the tunnel to get the port for
// @return int, the port
get_tunnel_port(String tunnel)
```

Listing 5: Signature for the function that gets the port

The specific types of communication are distinguished by the parameter `tunnel` as seen above. The variable is called as such due to the fact that the native ThinLinc clients use separate SSH tunnels for each type of communication. In the case for redirection of audio the protocol array was set to include the string “audio”. The pseudo-code in Listing 6 shows how the correct tunnel port for communication with the HTML5 client was established in that case.

```java
Array<String> protocols = headers.get("Sec-WebSocket-Protocol").split(",");
if ("audio" in protocols) {
    int audio_port = get_tunnel_port("audio");
    // Redirect audio data to the port...
}
```

Listing 6: Establishing the correct tunnel port

When opening the WebSocket for the local printing feature, the protocol array was set to include the string “printing”. The tunnel port for that feature was established in the same way as seen above, but with the “printing” string instead of the “audio” one. How the tlwebaccess processes determined which protocol they were going to work with and how the tunnel port was found, has thus been established. The remaining work in tlwebaccess consisted of relaying the data to and from the HTML5 client. The function used in the code for relaying the VNC data, `do_proxy`, was used. That code was present prior to the thesis. A pseudo-code signature of the `do_proxy` function is found in Listing 7.

```java
// Relays the received data to the specified network socket
// @param Socket conn, the socket to relay data to
do_proxy(Socket conn)
```

Listing 7: Signature for the relay function

The parameter `conn` seen in the signature above is a network socket. Lubbers et al describe a network socket as an endpoint in communication between multiple threads in one or more processes, across computer networks [8]. Each has an address which consists of an IP address and a
port number. The network socket needed for the relay function do_proxy was created by using a
standard library Python function. Other standard Python functions were used for configuring,
binding, listening to a connection, and for accepting a connection to this socket. PSF specify the
need of these functions [47]. Pseudo-code signatures and usages for these standard Python functions
is seen in Listing 8.

```python
// Creates a new network socket with the specified family and type
// @param int family, the family of the address, IPv4 or IPv6
// @param int type, the socket type
Socket.socket(int family, int type)

// Sets values for configuration options on the socket at the specified level.
// @param int level, the level at which the option is defined
// @param int optname, the name of the option
// @param String optval, the value to set
Socket.setsockopt(int level, int optname, String optval)

// Binds the socket to the specified address.
// @param Pair<address, port> the address and the port to bind to
Socket.bind(Pair<String address, int port> address)

// Listen for connections made to the socket.
// @param int backlog, the maximum number of queued connections
Socket.listen(int backlog)

// Accepts a connection and returns a new socket object usable to send and
// receive data on the connection and an address bound to the socket on the
// other end of the connection.
// @return Pair<conn, address>, the new socket object and the address
Socket.accept()
```

Listing 8: Python socket functions and examples

The parameters used for the functions above which start with Socket were constants found in the
socket class. The first constant, AF_INET, specified that the family of the address to the socket was
IPv4. IPv4 is the fourth version of the Internet Protocol, IP. The second constant, SOCK_STREAM,
specified that the socket type was a TCP socket. A TCP socket is a network socket using TCP. The
third constant, SOL_SOCKET, is the name of a level of options in the socket class. This level
contains the option accessed by the fourth constant, SO_REUSEADDR. This option does, if set,
allow the socket to bind to an address that already is in use. The argument “1” for setsockopt
represents that the option should be set to be active. The variable port represents the tunnel port that
was previously established. First the socket was created and the options set. The socket was then
bound to the established tunnel port and set to listen to incoming connections on that port.

Once the socket was created and had accepted an incoming connection, the do_proxy function
could be called. The socket object which was used was the one generated by the accept function,
conn. The tlwebaccess proxy work was thus finished and sent data could be received in the browser
on the client computer.
4.3.3 Implementation for redirection of audio

The redirection of audio feature made use of the tlwebaccess modifications made in the previous section. Firstly it was important to get audio data to send over the tlwebaccess proxy. The PulseAudio middleman server on the server computer had to be configured for this purpose. This server should receive data on a port. Note that this port is different from the tunnel port established in the previous section. The correct port for the PulseAudio middleman server was found with the following command on the server computer.

```
    echo $PULSE_SERVER
```

The argument to the `echo` command is an environment variable. According to freedesktop.org, the variable `PULSE_SERVER` corresponds to the port to which applications using libpulse send the audio data [39]. When configuring the PulseAudio middleman server, this port was used for a module called `module-native-protocol-tcp`. This module is a source for data in the PulseAudio protocol. The PulseAudio middleman server should also be configured to output data in EsounD protocol format. For this purpose, an other module, `module-esound-sink`, was used. This is a sink for data in the EsounD protocol. The server was thus configured by using the following command.

```
    pulseaudio -L "module-native-protocol-tcp listen=127.0.0.1 port=4915"
    -L "module-esound-sink"
```

The -L argument specifies that a module should be loaded [39]. This argument is followed by the name of the module. If the module is a source, an address and a port to listen to is required. In this case, the address that was used was 127.0.0.1 which is the address to the local host. The local host is the same computer as the PulseAudio middleman server is on, the server computer. The PulseAudio middleman server was thus configured to accept input in the PulseAudio protocol from port 4915 on the same computer and to output data in the EsounD protocol. The output from the middleman server was relayed to the HTML5 client via tlwebaccess.

Once the audio data could be received in the HTML5 client the work with implementing a JavaScript EsounD server started. The data was received in the HTML5 client on a WebSocket using a new socket object which was described in the previous section. The code that was present for receiving VNC data on a WebSocket from tlwebaccess used some interface functions from a module called `Websock`. This module provided a receive queue, a send queue, as well as a layer of abstraction upon the standard WebSocket interface. There were functions for both the receive and the send queue. Pseudo-code signatures of the five most important out of these functions are found in Listing 9.

```
    // Checks the number of bytes found in the receive queue
    // @return int, number of bytes in the receive queue
    Websock.rQlen()

    // Check if we must wait for 'num' bytes to be available in the receive queue
    // @param String msg, an optional message to pass for debugging
    // @param int num, the number of bytes to wait for
    // @param int goback,
    // @return Bool wait, true if we need to wait, otherwise false
    Websock.rQwait(String msg, int num, int goback)

    // Shifts the specified number of bytes from the receive queue
    // @return Array<Byte>, a byte array containing the bytes shifted from the queue
    Websock.rQshiftBytes(int bytes)
```
With the use of the two functions called \texttt{rQshiftBytes} and \texttt{rQshiftStr}, the EsounD server in the HTML5 client could receive data from the receive queue. The receive queue consisted of data received from the WebSocket. The \texttt{rQshiftBytes} function took data from the receive queue and returned it in the form of a byte array. The \texttt{rQshiftStr} function worked in a similar way but returned the data in the form of a string. The protocol written by ymnk call the data received in the EsounD protocol packets \cite{37}. The length in number of bytes of a received packet could be found through using the \texttt{rQlen} function. Each packet starts with a 4 byte long code representing the operation that should be executed. This code is called \textit{opcode}. After successfully receiving each packet the EsounD server should send a “1” in response to the sound client using the \texttt{send} function. The sound client is, as mentioned, the application playing the sound on the remote computer.

If an error occurs while receiving a packet, a “0” should be sent in response. The error that was handled was if the process had to wait more than one second for enough data to arrive. The \texttt{rQwait} function was used to check if we had to wait for more data. The threshold was set to one second due to the time sensitive nature of sound. If a sound was delayed by more than a second it was assumed that it would often be more confusing than useful to eventually play it. The action of the sound client following receiving an error response depends on the sound client in question \cite{37}. The PulseAudio middleman server does not perform any action when receiving an error response after sending a packet \cite{28}.

While playing sound in the remote session, the first received packet from the sound client had an opcode zero corresponding to the operation \textit{init}. This packet consisted of 20 bytes of data excluding the opcode. The structure of the \textit{init} packet can be found in Table 6.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|p{7cm}|}
\hline
\textbf{Part} & \textbf{Bytes} & \textbf{Information} \\
\hline
opcode & 4 & The opcode for \textit{init} is 0 \\
\hline
esdkey & 16 & The authentication key that should be compared with ~/.esd\_auth \\
\hline
endian & 4 & ENDN or NDNE represents the endianness, big endian or little endian respectively \\
\hline
\end{tabular}
\caption{The structure of the \textit{init} packet}
\end{table}

The first 16 bytes after the opcode was the \textit{esdkey}. This key should be compared with the contents of the local file ~/.esd\_auth. This comparison could however not be done since W3C specified that a browser does not have access to local files \cite{42}. The purpose of this authentication mechanism is that no client should be able to play sound on an EsounD server without having the proper \textit{esdkey}. However, in the context of this thesis, it was assumed that no other parties would get access to the socket that leads to the implemented EsounD server. Thus, an authentication mechanism was skipped.

The last four bytes in the \textit{init} packet indicated the incoming audio data's endianness. Blanc and Maaraoui write that endianness refers to the order of storing bytes in the computer's memory \cite{48}. The two options are little endian and big endian. Little endian means that the least significant byte is stored in the smallest address in the memory. Big endian means that the most significant byte is stored in the smallest address. The four endianness bytes represented as a string would be “ENDN” for big endian and “NDNE” for little endian.
The response sent to the sound client in case of success is a byte array in different order depending on the endianness, representing a “1”. The endianness specified in the init packet was required in order to know in what order to read the data of any future packets. Once a packet with the opcode for the init packet was found, the data in the packet was parsed. Pseudo-code describing the parsing of the data in the init packet is found in Listing 10.

```java
if (Websock.rQwait("init", 20, 0)) {
    // Wait for more data...
}
Array<Byte> esdkey = Websock.rQshiftBytes(16);
String endian = Websock.rQshiftStr(4);
Bool little_endian = (endian == "NDNE");
if (little_endian) {
    Websock.send([1,0,0,0]); // Least significant byte last
} else {
    Websock.send([0,0,0,1]); // Most significant byte last
}
```

Listing 10: Parsing the init packet

After the init packet came a packet called stream-play. This packet consisted of, so called, header information, as well as the actual audio data. This header information described the incoming audio data. The information included bit format, number of sound channels, type, action, bit rate, and a name for the connection. The structure of the stream-play packet can be found in Table 7. The bit format, number of sound channels and audio format are specified together in the first four bytes of the packet after the opcode. Each of these four bytes correspond to it’s own piece of information. The bit rate is found in the following four bytes. After the bit rate comes 128 bytes of data on which the name of the connection is found. The total size of the header information is thus 136 bytes. The size of the actual audio data is variable depending on the sound being played.

<table>
<thead>
<tr>
<th>Part</th>
<th>Bytes</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>4</td>
<td>The opcode for stream-play is 3</td>
</tr>
<tr>
<td>format</td>
<td>1</td>
<td>0 or 1 represents 8 bit or 16 bit data format respectively</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of sound channels</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0, 1, or 2 represents the type which can be stream, sample, or ADPCM respectively</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0, 1, or 2 represents the action which can be play, monitor, or record respectively</td>
</tr>
<tr>
<td>rate</td>
<td>4</td>
<td>The bit rate for the audio data</td>
</tr>
<tr>
<td>name</td>
<td>128</td>
<td>The name of the connection</td>
</tr>
<tr>
<td>data</td>
<td>?</td>
<td>The actual audio data</td>
</tr>
</tbody>
</table>

The information included in the stream-play packet was necessary for playing the audio data that followed. Once a packet with an opcode representing the stream-play packet was found, the data in the packet was parsed. The first part of this consisted of parsing the header information. Pseudo-code describing the parsing of the header information in the stream-play packet is seen in Listing 11. Note that this code snippet assumes little endian to increase readability.
Once the necessary information regarding the audio data was gathered the Web Audio API was setup. Web Audio API was used, as described in section 4.1.2, to play this audio data on the speakers. The AudioContext was created with a special case check to ensure support in webkit-based browsers such as Safari and Google Chrome as well. This was necessary due to the fact that the Web Audio API was implemented by Mozilla in parallel with the Web Audio API specification [49]. Some of the early implementation pieces were kept to ensure backwards compatibility. This special case check is required due to one of these pieces. This meant calling the constructor with a different name for those browsers where the original was not found.

The audio destination, the speakers, were connected to a special node called \texttt{scriptProcessor}. The \texttt{scriptProcessor} node is a reference to an AudioNode and has an event handler for an event called \texttt{onaudioprocess}. This event indicates when the node is ready to process more data and provides a pointer to an output buffer. This buffer comes from the destination node and is thus the path to playing the sound on the speakers. When the \texttt{onaudioprocess} event was detected the audio data processing was made. The audio data was in Pulse Code Modulation (PCM) format. PCM was designed by Reidel to be a way for analogue sample signals to be represented digitally [50].

\begin{verbatim}
Array<Byte> opcode = Websock.rQshiftBytes(4);
switch (opcode) {
    case 3: // stream-play
        if (Websock.rQwait("stream-play", 136, 4)) {
            // Wait for more data...
        }

        String format = Websock.rQshiftBytes(4);

        // Get the information from each byte of the format data
        Bool sixteen_bits = format[0] == 1;
        int nrOfChannels = format[1];
        int type = format[2];
        int action = format[3];

        // Set the bit_format variable depending on the sixteen_bits variable
        int bit_format = sixteen_bits ? 16 : 8;

        int rate;
        Array<Byte> rate_data = Websock.rQshiftBytes(4);
        // Convert the four bytes of data to the corresponding integer
        for (int i = rate_data.length -1; i >= 0; i--) {
            rate += rate_data[i] * (2^(8*i));
        }

        // Get the connection name data from the receive queue as a String
        String connectionName = Websock.rQshiftStr(128);

        // Setup Web Audio API...
        break Web Audio API...
    }
\end{verbatim}

\textit{Listing 11: Parsing the header information in the stream-play packet}

Once the necessary information regarding the audio data was gathered the Web Audio API was setup. Web Audio API was used, as described in section 4.1.2, to play this audio data on the speakers. The AudioContext was created with a special case check to ensure support in webkit-based browsers such as Safari and Google Chrome as well. This was necessary due to the fact that the Web Audio API was implemented by Mozilla in parallel with the Web Audio API specification [49]. Some of the early implementation pieces were kept to ensure backwards compatibility. This special case check is required due to one of these pieces. This meant calling the constructor with a different name for those browsers where the original was not found.

The audio destination, the speakers, where connected to a special node called \texttt{scriptProcessor}. The \texttt{scriptProcessor} node is a reference to an AudioNode and has an event handler for an event called \texttt{onaudioprocess}. This event indicates when the node is ready to process more data and provides a pointer to an output buffer. This buffer comes from the destination node and is thus the path to playing the sound on the speakers. When the \texttt{onaudioprocess} event was detected the audio data processing was made. The audio data was in Pulse Code Modulation (PCM) format. PCM was designed by Reidel to be a way for analogue sample signals to be represented digitally [50]. In computers, PCM is the standard for digital audio. The data was converted into the desired format through four parsing phases. The first phase was to set the sample rate and convert the data to the
correct bit format. The sample rate and the bit format used were the ones established earlier in this section from the header information. The audio data from the receive queue was represented as a byte array. This lead to require a conversion if the stated bit format was 16 bits. No conversion was needed if the bit format was 8 bits, since a byte is 8 bits. The second phase was to adjust the data according to the first bit in each byte of data. The first bit in each byte represented the sign of the value. Thus the third phase consisted of assigning the correct sign for the value. The third phase was to convert each value in the data into a number between -1 and 1. This was done because the Web Audio API requires the audio data to be in this format in order to play properly [34]. The last and fourth phase was to put the data on the sound channels of the output buffer. The number of sound channels were established earlier in this section. Pseudo-code for the parsing mentioned in the paragraph above is found in Listing 12. Note that this code snippet assumes little endian to increase readability.

```java
Array<Byte> data = Websock.rQshiftBytes(Websock.rQlen());
int audio;
onaudioprocess.outputBuffer.setRate(rate);

// If it is 16 bit format we need to take two bytes at a time instead of one
int step = sixteen_bits ? 2 : 1;
for (int i = 0; i < data.length; i += step) {
    // Convert to 16 bit format if needed
    if (sixteen_bits) {
        // Add the two bytes together with one left-shifted 8 times
        audio = data[i] + (data[i+1] << 8);
    }

    // Check if the value is larger than the max value and make it negative if so, since the first bit corresponds to the sign.
    int max_value = 2^(bit_format-1);
    if (audio > max_value) {
        audio -= 2 * max_value;
    }

    // Convert the value to a number between -1 and 1
    audio = audio / max_value;

    for (int c = 0; c < nrOfChannels; c++) {
        // Check which channel to put the audio on through checking the index
        // in the loop modulus the number of channels. Take bit format into
        // consideration when calculating the modulus by dividing with the step
        if (((i / step).mod(nrOfChannels) / step) == c) {
            (onaudioprocess.outputBuffer[c]).put(audio);
        }
    }
}
```

Listing 12: Parsing the audio data

Once the data was put on the output buffer the sound started playing on the local speakers. This was true in all cases except for when running the HTML5 client in Safari on an iOS device. Apple report
that on these devices the sound would not play unless it was started in a user-initiated event [51]. Examples of user-initiated events are keyPress events or touch events. A touch event is triggered, on devices with touch screens, when the user touches the screen. The HTML5 specification by W3C have a specified event for the very first touch event that is detected in the window [52]. This event is called touchstart. This event was used for the special case handling required for iOS. Once the event was triggered a BufferSource was created and its' buffer was filled with zeroes which represent silent sound. The BufferSource was then connected with the destination, the speakers. Thus, when the first touch event occurs on iOS devices, the HTML5 client starts playing silence. When actual sound is played the AudioContext and destination are reassigned to the proper entities. Pseudo-code of the special case for iOS is found in Listing 13. The implementation of the redirection of audio feature was thus finished.

```javascript
// Call a function when the touchstart event is produced in the browser window
Window.addEventListener('touchstart', function() {
    AudioBufferSourceNode source = context.createBufferSource();
    // Create a buffer with silence
    Array<int> silence[1024];
    for (int i = 0; i < 1024; i++) {
        silence[i] = 0;
    }
    source.buffer = silence; // Put the silence on the BufferSource
    source.connect(context.destination); // Connect with the speakers
    source.start(0); // Start playing the silence
});
```

Listing 13: Special case for iOS devices

### 4.3.4 Implementation for local printing support

The local printing feature made use of the tlwebaccess modifications made in section 4.3.2. The server side implementation consisted of the CUPS back-end for thinlocal as well as tlwebaccess. Since thinlocal did not require any modifications, tlwebaccess already had data to relay to the HTML5 client. The implementation phase of the local printing feature could thus start directly with implementing a JavaScript LPD server. The data was received in the HTML5 client on a WebSocket using a new socket object as described in section 4.3.2. The data on the WebSocket was retrieved using the same Websock interface functions that were described in the previous section. The data from thinlocal were in the form of messages in the LPD protocol format. As specified by IETF, each received message was a command and begun with a one byte long code [29]. This code was a binary number identifying the command. Following the code were different arguments. Unlike the ESD protocol, the LPD protocol did not have specified lengths for each part of a message. The only part with a specified length in any message was the one byte code at the start of the message. The LPD protocol instead relied on different delimiters to indicate when an argument or message ended. Arguments were delimited by a space. Messages were delimited by a newline character, 0x0A. This led to that retrieving the desired amount of data from the receive queue in Websock could not be done, since the desired amount was unknown. When data was received on the WebSocket, the LPD server could not know whether the messages in the data were properly ended or cut off. A boolean variable called delimited, was created to track this. This variable was be set to false while in the process of parsing a message and true when the last parsed byte represented a delimiter. A utility function, get_data, which took the correct delimiters into account and set the line_clear variable was implemented. The get_data function would retrieve correct amount of data from the receive
queue and return it. Pseudo-code of this function is found in Listing 14.

The messages in the received data was, as mentioned, commands. Some commands required that a one byte acknowledgement was sent back to the client [29]. The client, in this context, was thinlocal. This byte was sent using the Websock function send which was described in the previous section. Positive acknowledgements were marked by zeroes being sent. Negative acknowledgements were marked by any other pattern being sent.

Many of the commands had following subcommands in several layers. After receiving a first command, the following received commands could be subcommands to the first command. This encouraged the use of the state design on the server. States was used to keep track of which commands to expect. A state structure to handle these different types of commands was thus required. The states were implemented in such a way that the LPD server would only be in one state at a time. Depending on the current state the LPD server would handle each received message differently. Five states were implemented.

The states can be put in three layers. The first layer, the top layer, consisted of the idle state and the command state. The second layer consisted of the subcommand state. The third layer consisted of the states controlfile and datafile. The name of the five states in the layers are found, along with information about when each state is entered, in Table 8.

Table 8: The five printing states in three layers and when each state is entered

<table>
<thead>
<tr>
<th>Layer</th>
<th>State name</th>
<th>Enters state when</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>idle</td>
<td>Idle</td>
</tr>
<tr>
<td></td>
<td>command</td>
<td>Receiving general printing commands, such as “receive a printer job”</td>
</tr>
<tr>
<td>2</td>
<td>subcommand</td>
<td>Receiving printing job subcommands such as “receive data file”</td>
</tr>
<tr>
<td>3</td>
<td>controlfile</td>
<td>Parsing control file lines</td>
</tr>
<tr>
<td></td>
<td>datafile</td>
<td>Parsing data file</td>
</tr>
</tbody>
</table>

Listing 14: Getting the right amount of data from the receive queue

```java
String function get_data(String delimiter) {
    // Find the position of the delimiter in the receive queue
    int position = get_rQ.indexOf(delimiter);

    if (position == -1) {
        // The delimiter was not found
        delimited = false;
        return "";
    } else {
        // The delimiter was found at the position
        delimited = true;
        String arg = Websock.rQshiftStr(position+1);

        // Remove the last character from the string, the delimiter
        // before returning it
        return arg.remove(arg.length - 1);
    }
}
```

String function get_data(String delimiter) {
    // Find the position of the delimiter in the receive queue
    int position = get_rQ.indexOf(delimiter);

    if (position == -1) {
        // The delimiter was not found
        delimited = false;
        return "";
    } else {
        // The delimiter was found at the position
        delimited = true;
        String arg = Websock.rQshiftStr(position+1);

        // Remove the last character from the string, the delimiter
        // before returning it
        return arg.remove(arg.length - 1);
    }
};

Listing 14: Getting the right amount of data from the receive queue

String function get_data(String delimiter) {
    // Find the position of the delimiter in the receive queue
    int position = get_rQ.indexOf(delimiter);

    if (position == -1) {
        // The delimiter was not found
        delimited = false;
        return "";
    } else {
        // The delimiter was found at the position
        delimited = true;
        String arg = Websock.rQshiftStr(position+1);

        // Remove the last character from the string, the delimiter
        // before returning it
        return arg.remove(arg.length - 1);
    }
};

Listing 14: Getting the right amount of data from the receive queue

String function get_data(String delimiter) {
    // Find the position of the delimiter in the receive queue
    int position = get_rQ.indexOf(delimiter);

    if (position == -1) {
        // The delimiter was not found
        delimited = false;
        return "";
    } else {
        // The delimiter was found at the position
        delimited = true;
        String arg = Websock.rQshiftStr(position+1);

        // Remove the last character from the string, the delimiter
        // before returning it
        return arg.remove(arg.length - 1);
    }
};

Listing 14: Getting the right amount of data from the receive queue

String function get_data(String delimiter) {
    // Find the position of the delimiter in the receive queue
    int position = get_rQ.indexOf(delimiter);

    if (position == -1) {
        // The delimiter was not found
        delimited = false;
        return "";
    } else {
        // The delimiter was found at the position
        delimited = true;
        String arg = Websock.rQshiftStr(position+1);

        // Remove the last character from the string, the delimiter
        // before returning it
        return arg.remove(arg.length - 1);
    }
};

Listing 14: Getting the right amount of data from the receive queue

String function get_data(String delimiter) {
    // Find the position of the delimiter in the receive queue
    int position = get_rQ.indexOf(delimiter);

    if (position == -1) {
        // The delimiter was not found
        delimited = false;
        return "";
    } else {
        // The delimiter was found at the position
        delimited = true;
        String arg = Websock.rQshiftStr(position+1);

        // Remove the last character from the string, the delimiter
        // before returning it
        return arg.remove(arg.length - 1);
    }
};
The LPD server was implemented to always start in the idle state. However, when data was received the state was changed to command. The structured usage of non-idle states when receiving a message in the `handle_message` function is found in the pseudo-code in Listing 15. The variable state corresponds to the current state of the LPD server. The variable `line_clear` was explained earlier. The functions `parse_command`, `parse_subcommand`, `parse_controlfile`, and `parse_datafile` are described later.

```c
void function handle_message() {
    while (Websock.rQlen() > 0) {
        switch (state) {
            case COMMAND: parse_command(); break;
            case SUBCOMMAND: parse_subcommand(); break;
            case CONTROLFILE: parse_controlfile(); break;
            case DATAFILE: parse_datafile(); break;
        }
        // If the data is incomplete we need to wait for more data.
        if (!delimited) { return; }
    }
}
```

Listing 15: Handling messages differently depending on the state

When parsing a message it was determined if the current state should transition to another state. Transitions between states are depicted and each transition is marked with a number in Figure 11. That the server entered the command state when a message was received is indicated by transition number one in the picture. When in the command state, messages with codes corresponding to commands such as “print any waiting jobs”, “receive a printer job” and “remove jobs” were handled. The parsing of the commands in the command state was done in the `parse_command` function. The only command which was relevant to this thesis was the “receive a printer job”-command. This command came with one argument called `queue`, which corresponded to the name of the printing queue the job came from [29]. This command required an acknowledgement to be sent to the client. The other commands would be used if a printing queue was implemented. No such queue was required for the sought functionality in this thesis. The queue argument was thus left unused. Pseudo-code of the relevant functionality in the `parse_command` function is found in Listing 16. The `get_data` function is the utility function that was described earlier.

Figure 11: The numbered transitions between the five states
void function parse_command() {
    Array<Byte> code = Websock.rQshiftBytes(1);
    switch (code[0]) {
        case 0x02: // Receive a printer job
            String queue = get_data(" "); // Unused argument
            state = SUBCOMMAND; // Change to subcommand state
            Websock.send([0,0]); // Send positive acknowledgement
            break;
        //...
    }
}

Listing 16: Parsing in the command state

Once receiving the “receive a printer job”-command, the LPD server entered the subcommand state, transition number two in Figure 11. Three commands were handled in this state. All of these commands required that an acknowledgement was sent. The commands were: “abort job”, “receive control file” and “receive data file”. The command “abort job” returned the LPD server to the command state, transition number three in the picture. The other two commands indicated that files were to be received. The first command, “receive control file”, indicated that a control file was coming. This file contained information about the file to be printed. The command had two arguments following the binary code [29]. The first argument, count, consisted of the number of bytes in the control file which were to be received. The second argument, name, consisted of the name of the control file. The second command, “receive data file”, indicated that a data file was coming. This was the file to be printed. This command had the same corresponding arguments as the first command. The count for the data file was saved to a variable called file_size in the LPD server. Pseudo-code of the parse_subcommand function is found in Listing 17.

void function parse_subcommand() {
    Array<Byte> code = Websock.rQshiftBytes(1);
    switch (code[0]) {
        case 0x01: // Abort job
            state = COMMAND; // Change to command state
            break;
        case 0x02: // Receive control file
            int count = get_data(" "); // Number of bytes in the control file
            String name = get_data("\n"); // Name of control file
            state = CONTROLFILE; // Change to controlfile state
            break;
        case 0x03: // Receive data file
            file_size = get_data(" "); // Number of bytes in the data file
            String name = get_data("\n"); // Name of data file
            state = DATAFILE; // Change to datafile state
            break;
    }
    Websock.send([0,0]); // Send positive acknowledgement
}

Listing 17: Parsing in the subcommand state
The “receive control file”-command put the LPD server in the controlfile state, transition number four in Figure 11. The data in the control file consisted of rows of information which could start with 27 different codes [29]. These codes were single characters, either letters (both lower case and upper case), or numbers. Each code represented a piece of information about the file to be printed. Two important codes were “N” and “T”. The code “N” indicated that the line contained information about the name of the source file, the file that should be printed. The code “T” indicated that the line contained information about the type of the data file to be printed. PDF was the type in this case, since thinlocal always sends the files to be printed as PDF no matter the format of the source file. The different codes in the control file were parsed and the line was read to access the information after each code. The information about the file name and the file type were saved in variables called 
\texttt{file\_name} \textbf{and} \texttt{file\_type} respectively in the LPD server. Once the parsing of the control file was finished the state of the LPD server was changed back to subcommand, transition number five in the picture. An acknowledgement was also sent to the client when the parsing was finished. The parsing of the control file, in the parse_controlfile function, is shown in the pseudo-code in Listing 18. The code snippet is limited to only show the parsing of the two important codes mentioned above.

\begin{verbatim}
void function parse_controlfile() {
    String code = Websock.rQshiftStr(1);
    switch (code) {
        //...  
        case 'N': // Name of source file
                file\_name = get\_data("\\n");
                break;
        case 'T': // File type of file to print
                file\_type = get\_data("\\n");
                break;
        //... 
    }
    // Check if control file parsing is done
    if (Websock.rQlen() == 0 && delimited) {
        Websock.send([0,0]); // Send positive acknowledgement
        state = SUBCOMMAND; // Change back to subcommand state
        return;
    }
}
\end{verbatim}

\textbf{Listing 18: Parsing of the control file}

The “receive data file”-command put the LPD server in the datafile state, transition number six in Figure 11. The parsing of the data file was done in the parse_datafile function. This function made use of the information gathered from the control file. Since the size of each message is unknown the size of the data in the receive queue had to be monitored. Once the data in the receive queue had reached the same as the size stored in the file\_size variable, all the data for the file to be printed was available. The chosen solution consisted of using a data URL to store the file. The data URL string consists of two parts [41]. The first part is the type of the data, which in this case is stored in the file\_type variable. The type of the data is preceded by the string “data:” and followed by a semicolon. The second part is the data. The data is preceded by a string containing the encoding of the data. Once the data URL string was finished the idea was to present the print file which had
been received for the user. If the user's browser had a built-in PDF reader, like Google Chrome has, this print file would be presented as depicted in Figure 13. The print file was opened in a new tab in the browser. If the browser did not have a built-in PDF reader, the user would be asked if he or she wanted to download the file and open it in a PDF reader. The same code that would open the file in a new tab would generate this behaviour.

Browsers which had pop-up windows blocked by default, like Google Chrome and Apple's Safari, would require some special handling [53,54]. Different ways of detecting if the pop-up was blocked had to be used since different browsers handled this differently. What differs is the order in which different browsers change attributes of HTML elements when destroying them. In order to be able to open a new tab if pop-ups where blocked, the code for that would have to be called within a onclick event. For this purpose a new panel with two buttons was created. The panel was called printPanel and contained the two buttons openPrintButton and closePrintButton. The panel was designed as depicted in Figure 12. By clicking the openPrintButton the browser allowed the pop-up to be opened. When the print file was presented to the user, the user could choose to print the file from the PDF reader.

Once the pop-up window was opened an acknowledgement was sent to the client. The state of the LPD server was also changed back to idle as the print job was finished. This last state transition is marked as number seven in Figure 11. The functionality described in the sections above is gathered in the pseudo-code for the parse_datafile function in Listing 19. The implementation for the local printing feature was thus finished.

```javascript
void function parse_datafile() {
  // Check if data file buffering is done
  if (Websock.rQlen() >= file_size) {

    // Create the data url
    String dataurl1 = "data:" + file_type + ";base64," + Base64.encode(Websock.rQshiftBytes(Websock.rQlen()));

    // Open a new tab containing the dataurl
    Window popup = new Window(dataurl1, "popup");

    // Check if the popup was blocked
    if (((!popup || popup.closed || (typeof popup.closed=='undefined') || popup.outerHeight == 0 || popup.outerWidth == 0)) {

      printPanel.show();
      openPrintButton.onclick = function () {
        popup = new Window(dataurl1, "popup");
        printPanel.hide();
      };
      closePrintButton.onclick = printPanel.hide();
    }
    Websock.send([0,0]); // Send positive acknowledgement
    state = IDLE; // Change back to idle state
  }
}
```

*Listing 19: Parsing of the data file*
Figure 12: The notification that a print file has been received

Figure 13: The PDF file that was received
Chapter 5: Evaluation

Chapter five starts by explaining the tests that were done for the main features as well as the results of these tests. The results are discussed and evaluated. Lastly, the chapter covers an evaluation of future development. This development concerns the additional features referred to in chapter 1.

5.1 Testing of implementation

The testing process embraced Cendio's continuous peer review of the code but also consisted of feature specific tests. The continuous peer review means that the developers at Cendio read through and comment on each other's code changes. The testing of each feature was conducted after the implementation of a solution was finished. The testing phase for each feature contained acceptance tests as well as platform tests. The purpose of these tests were to find if the expectations and requirements of the features were met. Lawrence and Atlee's writings support the fact that such insights can be achieved through such tests [55]. The acceptance tests were conducted before the platform tests. All the acceptance tests were performed with the ThinLinc server and the client on the same computer in order to eliminate network delay from the results.

As depicted in Figure 14, both unit testing and integration testing was conducted prior to the acceptance and platform tests. The unit testing was done before the integration testing. These two testing phases were done in the development process. The last part of the testing which included acceptance and platform tests was done after the development process was finished. This chapter will only cover the acceptance and platform testing. The goal of the testing process was to measure whether the resulting features in the HTML5 client has reached the same level of functionality as the native ThinLinc clients. Both which functionality was present and how well each function performed in terms of data usage and delay was looked at. This section will start by looking at the tool used to measure data usage amount and delay. The process of the acceptance and platform testing is then described. Lastly, the testing setup for the tests of each feature is explained.

---

**Figure 14: The testing phases with parts presented in the thesis highlighted**
5.1.1 Measurement tool

In order to be able to measure data usage amount and delay of different features of the HTML5 client the application Wireshark was used. Wireshark is a tool by the Wireshark team for monitoring and analyzing network traffic [56]. When looking to monitor traffic from the local machine to the local machine, the real network interfaces will not be used [57]. Wireshark allows access to a special interface for this purpose, this interface is called the loopback interface. Wireshark also offers the possibility to filter the traffic via filters. Thus, a capture on the loopback interface was done and filters were made for each feature. The filters which were used for measurements in the HTML5 client are seen in Table 9.

**Table 9: Wireshark filters used for measurements of each feature in the HTML5 client**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Wireshark filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Keyboard input</td>
<td>tcp.dstport == 5901 &amp;&amp; vnc.client_message_type == 4</td>
</tr>
<tr>
<td>2. Redirection of audio</td>
<td>tcp.srcport == 4910 &amp;&amp; esd</td>
</tr>
<tr>
<td>3. Local printing support</td>
<td>tcp.srcport == 4913 &amp;&amp; lpd</td>
</tr>
</tbody>
</table>

The filters which were used for measurements in the native ThinLinc client are the same except for the filter for the audio feature. The filter for redirection of audio for the native ThinLinc client was instead as follows.

```
tcp.srcport == 4910 && pulseaudio
```

The traffic was presented as series of packets. The packets were listed along with information about the packets. The information was presented in multiple columns, each column for its own type of information. The two columns that was of interest for our measurements were the **Length** and the **Time** columns. The Length column displayed the size of the packet in bytes. The Time column displayed the time passed since the capture started in milliseconds. To measure data usage amount the value in the **Length** column of each relevant packet was added together. The data used is the total number of bytes used in the communication. The difference in the **Time** column for a single packet between when it appears on the input port on the middleman sound server and when it comes from the output port of the EsounD server, is the delay. The measured delay is the experienced delay by a single packet.

5.1.2 Acceptance testing

The acceptance tests required testing of a native ThinLinc client to form a benchmark to compare against. The latest stable version of the Linux client was chosen for this purpose. The ThinLinc Linux client was ran on a PC with the operating system Fedora 18. To avoid device and operating system peculiarities the acceptance tests of the HTML5 client was conducted on the same PC with the operating system Fedora 18 and the browser Google Chrome 32. Table 10 shows the acceptance test criteria which was tested for each main feature. Each criteria was in the context of comparison to the native ThinLinc clients. The second criteria for redirection of audio, “latency can be compensated for”, refers to the synchronization that was mentioned in section 4.1.2. Latency is here defined as, the time interval between audio data being sent and the same data being received. This can also be seen as for how long the data is en route. The most important part of synchronizing audio is to compensate for latency. This compensation would mean that, if latency is present, skipping to play some pieces of data to be able to play the most recent data as quickly as possible.
Table 10: Acceptance tests with both functional and non-functional criteria

<table>
<thead>
<tr>
<th>Feature</th>
<th>Criteria with functional criteria in bold</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Keyboard input</td>
<td>• All keyboard input works</td>
</tr>
<tr>
<td></td>
<td>• Multiple different language layouts work</td>
</tr>
<tr>
<td></td>
<td>• The data usage amount is the same or lower</td>
</tr>
<tr>
<td></td>
<td>• There is no added delay</td>
</tr>
<tr>
<td>2. Redirection of audio</td>
<td>• Basic audio output works</td>
</tr>
<tr>
<td></td>
<td>• Latency can be compensated for</td>
</tr>
<tr>
<td></td>
<td>• The data usage amount is the same or lower</td>
</tr>
<tr>
<td></td>
<td>• There is no added delay</td>
</tr>
<tr>
<td>3. Local printing support</td>
<td>• Basic printing works</td>
</tr>
<tr>
<td></td>
<td>• The data usage amount is the same or lower</td>
</tr>
<tr>
<td></td>
<td>• There is no added delay</td>
</tr>
</tbody>
</table>

The non functional criteria which were data usage amount and end-to-end delay were measured using the same approach in the Linux ThinLinc client and in the HTML5 client. Each measurement was conducted 10 times and the mean value of the measurements was what was counted for the acceptance tests. The conclusion of the acceptance test criteria for each feature reflected whether the criteria was fulfilled or not fulfilled in the HTML5 client compared to what was seen in the Linux ThinLinc client. The results of the acceptance had direct impact on the conclusions regarding each feature. If the result of an acceptance test criteria was not fulfilled, the conclusion was that the solution was limited. However, if a criteria was very close to being fulfilled some margin tolerance were applied.

5.1.3 Platform testing

The platform testing of the HTML5 client consisted of doing all the functional criteria tests as seen marked in bold in Table 10 which was done for the acceptance testing in all of the supported browsers on a number of different devices with different operating systems. The results of the platform testing had direct impact on the conclusions regarding each feature. If the result of a platform test criteria was not fulfilled the conclusion was that the solution was limited. The platforms which were tested on are presented in Table 11. Note that no mobile device with Windows 8 was available for testing purposes.

Table 11: Tested browsers and platforms

<table>
<thead>
<tr>
<th>Browser</th>
<th>Devices with operating system in parenthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE 10 or newer</td>
<td>PC (Windows 8)</td>
</tr>
<tr>
<td>Firefox 11 or newer</td>
<td>PC (Windows 8, Fedora 18), Nexus 7 (Android 4.4)</td>
</tr>
<tr>
<td>Chrome 16 or newer</td>
<td>PC (Windows 8, Fedora 18), Nexus 7 (Android 4.4), iPad Mini (iOS 7)</td>
</tr>
<tr>
<td>Safari 6 or newer</td>
<td>Mac (OS X 10.8), iPad Mini (iOS 7)</td>
</tr>
</tbody>
</table>
5.1.4 Testing setup

The measuring for keyboard input started from the input of the letter “A” into a terminal on the client until the client key event containing the written letter arrived at the client. Delay was measured as the time between the keyPress event was triggered on the client and the time the client key event arrived at the server. Data usage amount was measured as the total volume in bytes of the messages sent to the server. Note that, due to that the solution chosen for the keyboard input feature only affected Google Chrome on Android, this feature’s acceptance tests were conducted on a Nexus 7 with Android 4.4 in Google Chrome. The Nexus 7 device was connected via WiFi to the local area network where the ThinLinc server was located.

The measuring for redirection of audio started from the start of playback of a 5 second long soundtrack until the sound of the playback finished playing on the client. Delay was measured as the time between packet which included the start of playback and the time the sound of the playback finished playing on the client. Data usage amount was measured as the total volume in bytes of the messages containing the audio data sent from the server to the client.

The measuring for local printing started from the print click of a page containing the word “test” until the printing job appeared in the printers queue. Delay was measured as the time between the keyPress event which triggered the print function and the time the printing job appeared in the printers queue. Keyboard input delay was subtracted in the results. Data usage amount was measured as the volume in bytes of the message containing the printing data sent from the server to the client.

5.2 Results of the tests on the main features

This section covers the test results of the acceptance and platform tests for the three main features. The three main features were keyboard input, redirection of audio, and local printing support. Tables with the test results can be found in Appendix A.

5.2.1 Results for keyboard input

All acceptance tests was fulfilled and all of the platform tests were fulfilled for keyboard input, as seen in Table A.1 and Table A.2. The data usage amount for keyboard input, 148 bytes per character, is small and no big deviations were thus not expected in the results, this was also the case. The data usage amount was exactly the same in the HTML5 client as seen in the native ThinLinc client. This leads to the conclusion that WebSocket does not increase the bandwidth requirements compared to SSH tunnels.

The acceptance test criteria for delay was noted as fulfilled, as seen in Table A.1 even though there was a little added delay. There was 0.12 ms added delay in the HTML5 client compared to what was seen in the native ThinLinc client. However, it should be considered that the keyboard input acceptance test was ran on a different device and the delay is probably caused by this. The implemented solution does likely not have any impact. Note that the HTML5 client and the native client could not be tested on the same platform in this case since the implemented solution was specific to the Android platform. Due to the fact that the difference in delay was only 0.12 ms and that all the platform tests were successful the solution of the keyboard input feature was considered successful and not limited.
5.2.2 Results for redirection of audio

Two out of the four acceptance tests was not fulfilled for redirection of audio, as seen in Table A.3. One of the two functional criteria for the platform tests was not fulfilled on each platform, as seen in Table A.4. Due to the limited nature of the EsounD protocol there are no synchronization options. This meant that there is no good way to compensate for latency without introducing higher than acceptable jitter. The bandwidth results are positive and even improved compared to what was seen in the native ThinLinc client. The total data amount in the test was reduced by 2024kb. This is most likely because of the fact that the EsounD protocol offers less functionality compared to the PulseAudio protocol. The additional features in the PulseAudio protocol are likely to add more overhead.

The audio delay was increased from 9.0 ms to 140.3 ms with the solution for the HTML5 client. During the WireShark measurements it was noted that the module in the middleman PulseAudio server which converts the audio data from the applications to the ESD protocol is the cause of this delay. This means that if a PulseAudio server would have been implemented instead, the conversion would not have been needed and thus no delay would have been added. Additionally, the delay results are likely perceived worse due to the sensitivity of sound for the human mind. Just small stutters or synchronization errors will instantly be noticed and lower the quality of the experience.

The conclusion for the solution of the redirection of audio feature is that it was limited due to the unfulfilled acceptance and platform tests. A possible alternative for future work, to remove these limitations, would be to implement a PulseAudio server in JavaScript.

5.2.3 Results for local printing support

One out of the three acceptance tests was fulfilled and all of the platform tests were not fulfilled for local printing support, as seen in Table A.5 and Table A.6. That the data amount remains unchanged is no surprise since the network architecture used in the solution is identical to the one found in the native ThinLinc clients. The server sends exactly the same data over the network to the native ThinLinc clients and the HTML5 client for the printing feature.

Implementation of the automatic printing feature was outside of the scope of this thesis, as mentioned earlier. In the future the server-side could be developed to modify the PDF files to automatically print once being opened. Additionally, the architecture could be considered to be changed in the future. If the service tlwebaccess in the ThinLinc server would present the PDF file for download instead of sending the file directly to the browser, there would not be any size limitations. This would require the LPD server to be implemented in Python for tlwebaccess instead of in JavaScript for the client-side. This would result in a printing feature for the HTML5 client able to reach the same level as the native ThinLinc clients.

It can be concluded that the chosen implementation did not reach the same level as seen in the native ThinLinc clients. Thus the conclusion for the solution of the local printing support feature is that it was limited.
5.3 Evaluation of the additional features

This section covers the evaluation that has been made regarding the four additional features which were considered for future development. These four features were access to local drives, Kerberos authentication, access to smart card readers and access to serial ports.

Access to local drives would, as mentioned in section 1.2.2, require access to the local file system for the HTML5 client. Lyle et al write that web browsers are sandboxed environments that normally can not give a website access to local file systems [58]. This is a fact which is not specified directly by the browser vendors but is derived from the possibilities within the browser scope available to web developers. However, due to this it can be concluded directly that a solution working the same way as the native ThinLinc clients would be impossible to implement. Different approaches to the functionality could however be explored, with for example the File System API. Wright writes that the gap between native and web applications is closing [59]. The file access aspect is currently one of the most interesting parts of this gap. As mentioned, the File System API is a means to access sandboxed sections of the users local file system [42]. This means that other application’s files or arbitrary files on the file system can not be accessed. The File System API was only implemented in Google Chrome and Opera at the time of writing (fall of 2013) but it was expected that other big browsers such as Mozilla Firefox would implement it in the future [43]. The state of the tracking bug for the File System API in Mozilla Firefox indicate that this is not close to being finished [60].

Kerberos authentication would, as mentioned, require access to the underlying system's Kerberos APIs [12]. Due to the fact that web browsers can not access the underlying system’s APIs directly it can be concluded right away that it would be impossible to implement unrestricted Kerberos authentication.

However, if limitations are acceptable, other techniques can be employed to overcome these difficulties. Akhawe et al show through case studies how a Kerberos system could be used from HTML5 [61]. These studies do not fulfill the initial requirements in this thesis but illustrate that there are relevant things that can still be done. The Kerberos protocol could also be implemented in JavaScript together with a dedicated Kerberos server for the HTML5 client and thus a type of Kerberos authentication would be possible. Bella and Paulson showed through analysis back in 1998 that this could be done [62]. Such a solution would be exclusively accessible by the HTML5 client. Local resources such as Kerberos tickets could not be shared between native ThinLinc client-logins and HTML5 client-logins and thus the purpose of having a Kerberos authentication would be defeated. This could be achieved if the browser would be able to access the local filesystem.

A smart card reader is a piece of local hardware and browsers do not have access to interface directly with local hardware. Some hardware have dedicated APIs in the browsers allowing them to be used. Unless browser vendors implement an API for Personal Computer / Smart Card, PC/SC, communication it would be impossible to implement access to smart card readers in the HTML5 client. There are browser plug-ins and signed Java Applets allowing access to smart card readers, for example one by Gemalto [63]. But that fell outside the scope of the thesis.

Serial ports are local hardware and browsers do not have access to interface directly with local hardware. As with smart card readers, unless browsers specifically implement such support directly it would be impossible to implement access to serial ports in the HTML5 client. There are Java Applets which allow access to serial ports, one such applet was implemented by scream3r.org [64]. However, that also fell outside the scope of the thesis.
Chapter 6: Conclusions

The sixth and final chapter presents the conclusions regarding whether an HTML5 based client can reach the desired level of functionality. In the first chapter, section 1.3, the goal was defined as answering the question whether an HTML5 client for remote desktops could replace the native ThinLinc clients. The same section also outlined an hypothesis that included that the main features would be possible to implement to a satisfactory result while the additional features would be subject to limitations in the platform. This chapter will present conclusions and discussions relating to this goal and hypothesis. Lastly, this chapter explores possible directions for future work and discusses the applicability of HTML5 in general.

6.1 The desired level of functionality

The desired level of functionality was initially defined as the level seen in the native ThinLinc clients. The conclusion was that an HTML5 based remote desktop client can not reach that level of functionality. Six out of seven of the important features which were discussed in this thesis were derived as not satisfactorily implementable. The hypothesis in section 1.3 was thus true for all of the additional features but only true for one of the three main features. However, based on the investigation of possible technical solutions, it was found that the level one could reach was satisfactory for its purpose.

It could be argued that asking whether an HTML5 client can replace native clients was an unfair question which brought unreasonable expectations on a platform which is designed to be limited. This thesis explores the degree of these limitations. That is an important contribution for further development of the HTML5 client. An HTML5 client for remote desktops will not be replacing the native ThinLinc clients at Cendio but will still serve its purpose. The purpose is to be an easily accessible remote desktop client available on any platform. The goal in section 1.3 to answer the question is thus fulfilled and that clarifies that the intent for the HTML5 client is for it to be the best choice when there are no native ThinLinc clients available. An example of a case were no native ThinLinc clients are available is when the user wants to use an Android or iOS device.

6.2 Discussions and future work

An interesting question to ask is whether the HTML5 development is moving in a pragmatic direction. Many are interested in getting more features and possibilities while few are prepared to sacrifice security in order to get it. It is likely that this is a closed circuit, that you can not both have a universal computing platform and a complete sandboxed and safe environment at the same time. Some think that this development is in certain areas forced by the war between native mobile applications and HTML5 applications [16]. Such incentives can sometimes be counter-productive since the focus might drift away from the real strengths of the platform. That the web platform is developing is itself a good tendency however, there was a long stagnant period between 1999 and 2008 where HTML was widely considered to be a dead-end [8]. As Anthes writes, the recent developments have opened up a lot of possibilities and enabled new web applications which is undoubtedly a positive development [65].
There are several directions for future work for the ThinLinc HTML5 client. First, the implementation of a proper JavaScript PulseAudio server can be attempted for the audio feature. This will require a lot of time but the result would be a big improvement compared to the solution implemented in this thesis. Secondly, the solutions in place to handle multiple different protocols in tlwebaccess would probably have to be made robust and a security audit would be needed. Note that the work required for these two suggestions would not be connected to HTML5.

Thirdly, a realisation of the additional features, namely, access to local drives, Kerberos authentication, access to smart card readers, and access to serial ports, is also an interesting direction for future work. However, this requires that browsers will be able to provide specific support interfaces for these features. Finally, the HTML5 client could be extended with other features that were not considered in this thesis. Features not present in the native ThinLinc clients could be included. One such option would be a file transfer function. This could act as a replacement for the access to local drives feature. Another feature which might be interesting in the future is support for webcams. Future work may increase the potentials of the HTML5 client to excel in its specific area of use.
References


[34] Web audio API: specification. W3C; 2013 [cited 2013 Nov 5]. Available at: http://www.w3.org/TR/webaudio/


[38] GNOME now esound-free: blog post. Poettering L; 2009 [cited 2013 Nov 5]. Available at: http://0pointer.de/blog/projects/esound-free.html


Appendix A: Test results

Table A.1: Acceptance test results for keyboard input

<table>
<thead>
<tr>
<th>Acceptance criteria</th>
<th>Comment</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>All keyboard input works</td>
<td>No comment</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Multiple different language layouts work</td>
<td>No comment</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>The data usage amount is the same or lower</td>
<td>HTML5: 148 bytes, Native: 148 bytes</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>There is no added delay</td>
<td>HTML5: 0.2 ms, Native: 0.08ms</td>
<td>Fulfilled</td>
</tr>
</tbody>
</table>

Table A.2: Platform test results for keyboard input

<table>
<thead>
<tr>
<th>Browser and Platform</th>
<th>Functional criteria</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE 10, PC (Windows 8)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Firefox 24, PC (Windows 8)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Firefox 24, PC (Fedora 18)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Firefox 24, Nexus 7 (Android 4.4)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Chrome 32, PC (Windows 8)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Chrome 32, PC (Fedora 18)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Chrome 32, Nexus 7 (Android 4.4)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Chrome 32, iPad Mini (iOS 7)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Safari 7, Mac (OS x 10.8)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Safari 7, iPad Mini (iOS 7)</td>
<td>All keyboard input works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Multiple different language layouts work</td>
<td>Fulfilled</td>
</tr>
</tbody>
</table>
Table A.3: Acceptance test results for redirection of audio

<table>
<thead>
<tr>
<th>Acceptance criteria</th>
<th>Comment</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic audio output works</td>
<td>No comment</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>Latency can be compensated for</td>
<td>No good such option for EsounD</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>The data usage amount is the same or lower</td>
<td>HTML5: 7216 kb, Native: 9240 kb</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>There is no added delay</td>
<td>HTML5: 140.3 ms, Native: 9.0 ms</td>
<td>Not fulfilled</td>
</tr>
</tbody>
</table>

Table A.4: Platform test results for redirection of audio

<table>
<thead>
<tr>
<th>Browser and Platform</th>
<th>Functional criteria</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE 10, PC (Windows 8)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Firefox 24, PC (Windows 8)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Firefox 24, PC (Fedora 18)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Firefox 24, Nexus 7 (Android 4.4)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Chrome 32, PC (Windows 8)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Chrome 32, PC (Fedora 18)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Chrome 32, Nexus 7 (Android 4.4)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Chrome 32, iPad Mini (iOS 7)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Safari 7, Mac (OS x 10.8)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Safari 7, iPad Mini (iOS 7)</td>
<td>Basic audio output works</td>
<td>Fulfilled</td>
</tr>
<tr>
<td></td>
<td>Latency can be compensated for</td>
<td>Not fulfilled</td>
</tr>
</tbody>
</table>
### Table A.5: Acceptance test results for local printing support

<table>
<thead>
<tr>
<th>Acceptance criteria</th>
<th>Comment</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic printing works</td>
<td>No automatic printing</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>The data usage amount is the same or lower</td>
<td>HTML5: 11 kb, Native: 11 kb</td>
<td>Fulfilled</td>
</tr>
<tr>
<td>There is no added delay</td>
<td>Inapplicable due to no auto-printing</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

### Table A.6: Platform test results for local printing support

<table>
<thead>
<tr>
<th>Browser and Platform</th>
<th>Functional criteria</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE 10, PC (Windows 8)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Firefox 24, PC (Windows 8)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Firefox 24, PC (Fedora 18)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Firefox 24, Nexus 7 (Android 4.4)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Chrome 32, PC (Windows 8)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Chrome 32, PC (Fedora 18)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Chrome 32, Nexus 7 (Android 4.4)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Chrome 32, iPad Mini (iOS 7)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Safari 7, Mac (OS x 10.8)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
<tr>
<td>Safari 7, iPad Mini (iOS 7)</td>
<td>Basic printing works</td>
<td>Not fulfilled</td>
</tr>
</tbody>
</table>
Appendix B: Interview with Joel Martin

Joel Martin is Principal Software Engineer at ViaSat, Computer Science PhD at University of Texas, Arlington, web developer since 1996, creator and owner of the noVNC and the Websockify projects, contributor to W3C HTML5 specification (WebSockets and Canvas).

Q: Would you agree on that one of the fundamental ideas of web development is that the browsers should be limited "sandboxed" environments?

A: “I agree that a browser should be a sandboxed environment and technically a sandbox environment implies some level of limitations. However, I also think the browser should be a complete computing platform and that one of the purposes of the W3C is to safely guide the web platform towards a universal computing platform without compromising the safety of the users of that platform. In other words, I think that the browser sandbox should be just enough to fully protect the user but no more limited than that.”

Q: What do you think of the direction the web is developing in? I am mainly talking about the constant additions to the HTML5 specification. Some would argue that opening up for more features in the browser will widen the vector of risk and open up for more security holes.

A: “I think the direction that the web is heading is generally a very good one. I do have some areas of concern however. I think it’s important to distinguish the W3C HTML5 specification itself from the more common usage of the "HTML5" term. The W3C HTML5 specification (http://www.w3.org/TR/html5/) is a actually a fairly narrow spec that codified the sane and consistent parts of the web platform that were common to most advanced browser circa 2008. For example, it was generally considered good practice to separate content from presentation and so the HTML5 spec eliminated, redefined, or discouraged content tags that primarily served as presentation tags (big, center, font, hr, b, i, etc.).

Many parts of the HTML5 spec that represent brand new functionality (WebSockets) were separated into separate specs but still under the HTML5 umbrella. I think the W3C HTML5 spec and the W3C HTML5 complement specs are quite good and I have very few problems with them. The HTML5 spec and most of the complementary specs are under candidate recommendation status and have been fairly stable for quite a while.

Once you move outside of the core W3C specifications, there is a lot of churn happening in “HTML5+”. The web platform is moving and adapting very quickly (especially compared to the web’s stagnant period from about 1998 to 2008). However, there is currently a mind-share war going on between the mobile native app platforms and the web platform. In some areas I think this is forcing the web platform to innovate and change at such a fast pace that sometimes security, simplicity and consistency is being sacrificed in order to achieve a short time to market.”

Q: Do you think any of the additions have gone too far? For example, do you oppose the idea that websites should expect to be able to use bigger amounts of the users local storage with the File System API?

A: “I consider many of the recent additions and proposals (File System, WebRTC) as necessary and import to fill gaps in the web platform. The concerns I have with these are not whether the web should have them, but whether browsers vendors will come up with easy ways for end-users to monitor, control and understand these new features. I actually have deeper concerns about the fact
that cookie management has become less accessible to normal users than I do about disk space usage due to the File System API. But in both cases I think it needs to be clear to users how their resources are being used by web applications and how to control and limit them.

I do think that the web standardization committees should focus on developing safe low-level APIs rather than high level APIs. Brendan Eich goes into this strategy in detail in a very recent infoq video: http://www.infoq.com/presentations/web-evolution-trends My opinion pretty much aligns with his on this issue.”

Q: I am also thinking of a response you wrote 6 months ago in the noVNC google group: https://groups.google.com/forum/#!topic/novnc/61JQ_A7AOkY

“It turns out that in most cases the performance of the standard tight encoding is actually quite good. The advantage of tightPNG is that it requires less CPU for the client (because it can mostly just blit the data to the canvas without much decoding). You won’t get native performance with noVNC (at least not until a full conversion to binary data + asm.js code happens).”

Can I quote you on this?

A: “Yes.”