Final thesis

Extending browser platforms with native capabilities, enabling additional features in a media streaming context

by

Nicklas Schultz

LIU-IDA/LITH-EX-A–15/027–SE

2015-06-18
Final thesis

Extending browser platforms with native capabilities, enabling additional features in a media streaming context

by

Nicklas Schultz

LIU-IDA/LITH-EX-A–15/027–SE

2015-06-18

Supervisor:  Anders Fröberg
Examiner:   Erik Berglund
Abstract

Web applications are playing an increasingly central role in our everyday life, as many of our computer-based tasks are performed solely within the web browser. Desktop applications requiring installation are becoming part of the past and most applications get a web alternative — turning the web into a target platform for application development. A complication with this is that the web lacks the performance of native applications, motivating a need for something more powerful. This thesis investigates different technologies that aim to solve this problem by allowing developers to create native web applications, that allow web applications to use native capabilities. Two such technologies are Xax and Native Client (NaCl), which allow web applications to run compiled native code in the web browser, while still providing the portability across operating systems. In addition to the investigation a case study of how Portable Native Client (PNaCl) is to use for creating a native web application. A performance analysis is presented in order to determine how large the performance gains are with native web applications compared to traditional. The results show that NaCl performance is far superior to JavaScript, and very close to native speeds. Native Client is a great technology, both for migrating legacy C/C++ applications to the web and for creating high performance native web applications. The biggest challenge for Native Client is to get accepted by other browser manufacturers.
## Contents

1 Introduction .............................................. 1
   1.1 Accedo .................................................. 2
   1.2 Purpose .................................................. 2
   1.3 Limitations .............................................. 2
   1.4 Methodology ........................................... 2
   1.5 Discussion about sources ............................... 3
   1.6 Outline .................................................. 3

2 Problem description ....................................... 5
   2.1 Problem statement ..................................... 5
   2.2 Goals .................................................... 6

3 Background ................................................ 7
   3.1 Accedo VIA .............................................. 7
   3.2 History .................................................. 8
      3.2.1 Browser usage ..................................... 8
   3.3 Plugins and Extensions ................................ 9
      3.3.1 Netscape Plugin API ............................... 10
      3.3.2 Pepper & Pepper Plugin API ..................... 10
      3.3.3 ActiveX Controls ................................ 11
      3.3.4 FireBreath ......................................... 11
      3.3.5 Java Applet ....................................... 12
      3.3.6 Add-ons security ................................. 13
   3.4 Native vs Web Applications ............................ 13

4 Theory ...................................................... 15
   4.1 Native Client .......................................... 15
      4.1.1 NaCl ................................................. 16
         4.1.1.1 Architecture ................................ 16
         4.1.1.2 Implementation ............................... 18
         4.1.1.3 Structure of a web application ............ 21
         4.1.1.4 Toolchain .................................... 21
      4.1.2 PNaCl ............................................... 22
         4.1.2.1 Toolchain .................................... 23
4.2 Xax

4.2.1 Architecture

4.2.2 Picoprocesses

4.2.3 Platform Abstraction Layer

4.2.4 Services via browser mechanisms

4.2.5 Legacy code modification

4.2.6 Browser integration

4.3 Compile native to web

4.3.1 Emscripten

4.3.2 asm.js

4.4 Comparison

5 Case study

5.1 Theory

5.1.1 Requirements

5.1.2 Universal Plug and Play

5.1.3 UPnP A/V

5.1.4 Transcoding

5.1.5 External device

5.1.6 The NaCl IO Library

5.1.7 Mini-XML

5.2 Implementation

5.2.1 Discovery

5.2.2 Control

5.2.3 Accedo VIA

5.2.4 PNaCl

5.3 Notes

6 Results

6.1 Application

6.1.1 NaCl SDK

6.2 Porting legacy code

6.3 Performance

7 Analysis

8 Conclusion

8.1 Future Work

A Further Reading

A.1 NaCl Sandbox Internals

B Examples

B.1 NaCl embed object

B.2 NaCl module components

B.3 nacl-gb debugger
Abbreviations

**ABI** Application Binary Interface

**AOT** Ahead-of-time

**API** Application Programming Interface

**AWT** Abstract Window Toolkit

**BSD** Berkeley Software Distribution

**GCC** GNU Compiler Collection

**IMC** Inter Module Communication

**JIT** Just in Time

**LLVM** Lower Level Virtual Machine

**NaCl** Native Client

**NAS** Network Attached Storage

**NPAPI** Netscape Plugin API

**PAL** Platform Abstraction Layer

**PNaCl** Portable Native Client

**POSIX** Portable Operating System Interface

**PPAPI** Pepper Plugin API

**SDK** Software Developer Kit

**SFI** Software Fault Isolation

**TCB** Trusted Code Base

**UPnP** Universal Plug and Play

**UUID** Universally unique identifier

**SOAP** Simple Object Access Protocol
SRPC Simple Remote Procedure Call

SSDP Simple Service Discovery Protocol
Glossary

**Application Binary Interface (ABI)** An interface at machine code level that determines how two program modules call functions and in which binary format information should be passed between components.

**Application Programming Interface (API)** Describes the problem description and goals that both the practical and theoretical part of this thesis is aimed to solve.

**Universal Plug and Play (UPnP)** A standard that defines a set of protocols that allows network devices such as NAS, PC, printers or any other device that can connect to the network.

**Trusted Code Base (TCB)** The trusted code base contains the software components that is essential to the systems security. If a bug or error exists inside the TCB, it could risk the security of the entire system. Therefore small TCBs are preferable since a deep and excessive examination of the TCB components is needed in order to say that they are safe.

**Abstract Window Toolkit (AWT)** Java's original platform independent toolkit for graphics, windowing and UI.

**syscalls** A system call is how a program requests a service from an OS kernel.

**Same-origin-policy** A policy that allows a script running in a web page to access data in a second web page, but only if the two web pages have the same domain.

**GNU Compiler Collection (GCC)** GCC is a compiler system that supports multiple languages. It is developed by the GNU project and was originally designed to only handle C code, but has later been extended to compile several languages including C++ and Objective-C.
List of Figures

3.1 Approximate browser usage reported by StatCounter, retrieved 2015-04-14 ......................................................... 9

4.1 Native Client application architecture ................................................. 17
4.2 Static analyzer NaCl inner-sandbox .................................................. 19
4.3 Simplified scheme showing steps from source code to running NaCl application ......................................................... 22
4.4 Simplified scheme showing steps from source code to running PNaCl application ......................................................... 23
4.5 Xax Architecture ........................................................................... 24
4.6 Simplified figure over the Emscripten architecture ......................... 27

5.1 UPnP A/V Architecture ................................................................. 33

6.1 Shows ratios against C for different technologies ............................... 42
List of Tables

3.1 Plugin support for the major browsers . . . . . . . . . . . . . 11
4.1 Describing features for the different technologies. . . . . . . . 28
6.1 Results that Webber presents in [43]. . . . . . . . . . . . . 41
Chapter 1

Introduction

The introduction of HTML5 and increased JavaScript performance have enabled the creation of more powerful web applications, allowing web applications to get a similar look and feel as desktop applications.

The web platform is more often becoming the target platform for application development putting further demands on performance. But running applications at near-native speed is no easy thing to achieve in the browser as the web browser was originally designed to display static document back in the early 1990s. Today there exists JavaScript engines that have increased the performance of the web significantly but native speed is still hard to achieve using a dynamic language like JavaScript.

So the solution might not be to speed up JavaScript, but to run native code in the browser instead. Popular technologies that does this has over the years mainly been NPAPI and ActiveX. They allow you to build native applications that can run in the browser as plugins, but these technologies suffer from severe security issues, as the native code runs with the same privileges as the browser and can therefore execute arbitrary code directly against the OS.

To solve the problems of bringing high performance applications to the web without compromising security different technologies for creating native web applications have started to emerge. This paper will look at a few of these technologies, investigate how they work and evaluate their strength and weaknesses compared to each other as well as standard web applications.

A case study will be performed by creating an application that lets the browser discover and communicate directly with an external device without having to install any plugin or software.
1.1 Accedo

Accedo was founded in 2004 by Fredrik Andersson and Michael Lantz, and has grown to thirteen offices around the world with the headquarter located in Stockholm. They deliver the future TV application experience with a range of products and solutions. Accedo has applications running on over 40 different platforms, more than 1000 deployed applications reaching more than 100 million households.

1.2 Purpose

The purpose of this thesis is to explore how PNaCl can be used for creating native web applications. Examine what gains and losses PNaCl has compared to traditional web applications and also investigate what other technologies for creating native web applications that exists besides PNaCl.

1.3 Limitations

There are a few limitations with this thesis:

- Due to time limitations the application will have focus on getting the functionality working first and then on appearance.
- Since I have limited experience from C programming the solution can probably be further optimized.
- Details about how various components are implemented in the different technologies will not be covered in this paper since its not covered by the scope of thesis, e.g. how the trampoline/springboard mechanism in NaCl prevents untrusted code from achieving arbitrary transfer control.

1.4 Methodology

Three major parts can be identified in this thesis. The first part is a thorough literature study in order to gain knowledge about native web applications. Including how web sites have gone from displaying static documents to web applications with similar look and feel as native applications.

The second part of the thesis is the design and implementation phase. The methodology used for this project is scrum, or personal scrum seeing as I am alone working this project. Personal scrum is an Agile methodology that adapts scrum to one person projects. Obviously it differs in some ways, like daily stand up meetings which aren’t very useful by yourself, but many aspects of scrum can be adapted to personal scrum.
In scrum there are 4 different activities. There is sprint planning, daily scrum, sprint review and sprint retrospective. All of these with daily scrum in a parenthesis can be performed in personal scrum.\([27]\)

With my thesis supervisor I have had sprint planning and sprint review every Monday. This is so that both me and him can keep track of my overall progress, to see where I am now and where I am going from here. As we have this every Monday this of course means that I have 1 week sprints, this is to get quick feedback, which I think is even more important when you work alone as you don’t have anyone else to fall back on. Daily scrum meetings, short reflection about last days work. During sprint retrospective I took some time reflecting on how the last sprint went.

The final part of this thesis is to test how the performance of a PNaCl application stands against a native application and how traditional JavaScript compares to the newer technologies. Reflect on the results of the thesis and the technologies examined.

1.5 Discussion about sources

When using sources it’s important to question the credibility of the information. With web sources e.g. blogs and articles, this is even more important seeing as the web is open for anyone and therefore anyone can be the author. To help me with this I have used a checklist that defines several questions in 6 different areas, Authority, Accuracy, Objectivity, Coverage and Intended Audience, Currency and Interaction and Transaction Features\([41]\). The purpose of the questions is to help the user determine the credibility of the web page (Ann Tate 2009). I haven’t answered all questions for every web page, because some questions didn’t feel relevant, questions that I have put extra weight in are questions like:

- **Is the authors name and qualifications presented?** In order to determine how trustworthy the author is.

- **Are the sources presented?** If the sources used is referenced the information can be verified.

- **Is it clear what point of view the author has about the topic?** If the topic of the content is a comparison between two technologies and the author is the lead developer of one of them, you could question his objectivity.

- **Is the date of when the page was last revised available?** In order to be able to determine how up-to date the information is.

1.6 Outline

The rest of the report is structured as follows:
Problem description: Describes the problem description and goals that both the practical and theoretical part of this thesis is aimed to solve.

History of the web: This chapter gives some background to how the web has evolved from viewing static content to Web 2.0 and towards native web applications. Also discusses different plugin interface that has been used over the years and how the security shortcomings in these interfaces have led towards their downfall.

Native Web technologies: A summery of different methods and technologies used to run native code on the web.

Case study: Case study of an application written in PNaCl communicates with a UPnP device. Starts with going through the theory behind the application and ends with implementation and design decisions.

Results: Presentation of the results. Presenting how the application turned out and performance results for the benchmark testing.

Conclusion: Discussion about the different technologies looked at as well as the results.

Analysis: An analysis is performed over how the goals and problem statements setup for this thesis is met.
Chapter 2

Problem description

The main objective is to develop an application demonstrating how PNaCl can be used to create an application, allowing a browser to discover and communicate with media servers located on the local network. The application should be integrated in an existing Accedo product called VIA Web.

Other technologies that could be used instead of PNaCl to run native code in the browser will be investigated and evaluated as well as PNaCl. The investigation will be done by examining how each technology works, what their strength and weaknesses are, what’s feasible with the technology and their portability. Further a performance test will be performed in order to provide a comparison between traditional web applications and native web applications.

2.1 Problem statement

The main problem with this thesis is:

Exploring the use of PNaCl for mixing native code and HTML5 in web browsers.

In order to be able to answer this to the best capacity I will have to actually use PNaCl to create an application. This application should run in the browser and should not have to be installed as a plugin or require any user acceptance like pop-up acceptance.

In addition to this problem I will also investigate different technologies for deploying native code in the browser that could be used in PNaCl’s place. By doing this I hope to get a wider understanding of how PNaCl’s approach differs from technologies that aims to solve the same problems and thereby making me better suited to assess PNaCl.
2.2 Goals

The expected result is:

- A comparison between different technologies, based on their features, portability and feasibility.
- An application using PNaCl that allows communication between an external device and a browser.
- A performance comparison between a native and a web version of an application.

By answering these goals I hope to be able to provide a basic understanding of different native web technologies that exists and also in what areas you gain the most by using such a technology.
Chapter 3

Background

This chapter gives a short background to how web sites have evolved into web applications with look and feel similar to native applications.

A short introduction to a few different browser plugin interfaces including NPAPI and ActiveX which has been around since the middle of the 1990s and is still used today and how security issues with these interfaces led to the creation of newer interfaces like PPAPI.

But first a quick introduction to the Accedo VIA product that my application will be integrated to is given.

3.1 Accedo VIA

Accedo VIA is a multiscreen application solution providing TV operator and media companies with the ability to launch full-featured digital entertainment services quickly across multiple connected devices such as Mobiles, Tablets, Smart TVs and PCs.

Some of the key features of Accedo VIA applications:

- Live TV
- Program Guide
- Video on Demand
- My Account
- Favorites
- Watch History
- Social
3.2 History

When the World Wide Web was created in the early 1990s websites consisted primarily of static information, they had low interactivity and lacked the functionality needed to create rich interactive websites. These weaknesses lead to the development of browser plugins and applets. According to Sir Tim Berners-Lee, the inventor of the World-Wide-Web, the introduction of Java and Java applets opened up new potentials for the web, allowing creation of applications that would be simple and inexpensive.\[16\]

Next big step for the web was Web 2.0. Web 2.0 is a term that has many different definitions, but some key concepts in common. In short Web 2.0 has transformed the Web from a network of information to browse to a network of participation, with key ideas of sharing and user generated content.\[33\] Along with this web browsers evolved and HTML5 opened for new capabilities that allowed for more complex web sites. Interactivity and functionality of the web sites grew and web sites evolved into web applications with similar look and feel as native applications, with one significant difference they run solely in the web browser.

3.2.1 Browser usage

When researching technologies for creating web applications one thing became apparent very quickly and it was that technologies most often only supports a limited set of browsers. From a business perspective you most often look to develop for the greatest number of platforms allowing for the greatest number of users. But since all technologies aren’t supported on all browsers, you might have to exclude some browsers from being supported, or create multiple solutions targeting different platforms. When making that decision it is interesting to know how much a specific browser is used.

Internet Explorer has been the leading browser when it comes to usage for many years but a few years ago Chrome took over that position and is today the dominating browser used.

There are many different numbers regarding this, StatCounter says that Chrome stands for 50% of the usage today, but w3schools says that Chrome has over 63%. I believe that w3schools figure which is based on their users only for the w3schools page is a bit misleading as people visiting their website are developers, developers tend to know more about different browsers then the average user that probably uses the browser installed by default. StatCounter on the other hand uses a web analytics service to analyze the browser usage. Their tracking code is installed on over 3 million web sites located all over the world, leading to billions of recorded page views every month and more unbiased results.

In figure 3.1 the results that StatCounter\[11\] reports is presented.
3.3 Plugins and Extensions

There are different ways of running native code in the browser, historically the use of plugins have been the way to go, and especially the Netscape Plugin API and Microsoft’s ActiveX. But now there exists alternatives, this chapter will present different plugin interfaces and later in the thesis other methods of deploying native code in browsers will be discussed.

First off lets clarify the difference between a plugin and an extension so that we don’t get them mixed up. A plugin is a simple third party library that plug-in to the page, it can be embedded in the page using an `<embed>` tag or an `<object>` tag. Plugins only affect one page, and can either be run as a mime type or they can run in place of a page. Knowledge about tabs and other pages in the same browser process is not available for the plugin, meaning the plugins only know about the page they run in and can thereby don’t affect browser menus or creates toolbars.

Examples of common plugins are:

- Macromedia Flash
- Adobe Reader
3.3. PLUGINS AND EXTENSIONS  

- Apple Quicktime

Extensions affect the browser itself, they can add onto the browser UI and process pages. But they are not limited to that, an extension can affect the page as well. They run with the same privileges as the browser, and can contain plugins.

Examples of extensions are:

- Firebug
- AdBlock

3.3.1 Netscape Plugin API

Netscape plug-in API or NPAPI is a cross-browser API for developing browser plugins. It was first introduced in the Netscape Navigator 2.0 browser in 1995. One popular plugin that came in 1996 using the NPAPI was Adobe Acrobat 3.0 (Amber), which is a pdf reader, that has been released in many updated versions since then and is still used today.

NPAPI has over the years been supported on all major browsers, though some browsers have dropped support for NPAPI at different times, Internet Explorer dropped support for NPAPI with version 5.5 SP2, and has since only directly supported plugins using ActiveX. NPAPI took another hit when Google announced in September 2013 that they will start phasing out support for NPAPI starting January 2014. They state that the reason was that NPAPI plugins had become the leading reason for hangs, crashes and security incidents. In the release of Google Chrome version 42 NPAPI is blocked by default and causes thereby that plugins using NPAPI is blocked, it is still possible to re-enable NPAPI but that alternative will also cease to exist in September 2015 when Google plans on removing that option as well.

Mozilla thought in similar paths when they implemented the feature click-to-play plugins in Firefox. The click-to-play feature means essentially that Firefox will not activate plugins automatically and instead ask the user if they want to run the plugin and then load it. The only plugin that isn’t affected by this is Flash.

3.3.2 Pepper & Pepper Plugin API

In 2009 Google introduced a new project called Pepper, with the associated Pepper Plugin API or PPAPI. The Pepper project was started in order to address portability, performance and security issues with NPAPI, and was at first based on NPAPI to allow easy adoption by plugin developers, but was later rewritten and deviated substantially from NPAPI.

At present time the only browsers that support the PPAPI is Chrome, Chromium and Opera 24 and later. Pepper applications can though run on Firefox, Safari, Internet Explorer and more through the use of pepper.js,
which is a JavaScript library that by the help of Emscripten which is described in section 4.3.1 compiles Pepper applications to JavaScript.\[17\]

### 3.3.3 ActiveX Controls

ActiveX is not as many people think a technology for creating plugin like applications. ActiveX is a framework created by Microsoft containing a collection of technologies, protocols and APIs. One technology that implements ActiveX technologies is ActiveX controls. ActiveX controls are similar to NPAPI plugins in many ways. An ActiveX control can be automatically downloaded and executed by browsers, I say browsers but in theory the only browser that fully supports ActiveX controls is Internet Explorer. ActiveX controls can be embedded in a HTML document by the `<object>` tag and can be developed using multiple languages including C++ and Java. Unlike Java applets which is described in 3.3.5 ActiveX controls have full access to the underlaying OS. This means that they are more powerful but also increase the security risks. To deal with the security risks Microsoft developed a system for signing and authenticating ActiveX controls. Allowing users to see who signed it and validate that the control have not been modified since the signing. \[39\] Unfortunately this method for security in non-technical, it does not stop you from downloading and executing malicious controls, it only in best case points you to the developer of the control.

<table>
<thead>
<tr>
<th>Browser</th>
<th>PPAPI</th>
<th>NPAPI</th>
<th>ActiveX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safari</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Chrome</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Firefox</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Internet Explorer</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 3.1: Plugin support for the major browsers

Table 3.1 shows what plugin interfaces the 4 major browsers support. A note about the Chrome support for NPAPI, it is blocked by default but still possible to enable manually, thats why the table shows support for NPAPI.

### 3.3.4 FireBreath

FireBreath is a framework that allows the creation of cross-browser plugins by using both NPAPI and ActiveX. For browsers running on Windows, Mac or Linux, FireBreath will work as NPAPI plugin if the client uses any of the browsers:

- Firefox 3.0 and later
- Google Chrome 2.0 and later
• Apple Safari
• Opera

and as a ActiveX plugin on Microsoft Internet Explorer 6 and later.\[15\]

### 3.3.5 Java Applet

As stated earlier the introduction of Java Applets in web browsers changed the way content was thought of in web sites. Applets provided a dynamic and active content in contrast to the existing passive content. But with these dynamic self-executing programs came two challenges, security and portability.

Both these issues was solved using the same solution, which was the Java Virtual Machine(JVM). A Java program isn’t compiled to executable code but to bytecode which is a set of instructions that is then executed by the JVM. This means that the JVM control the execution and can thereby prevent unwanted side effects. Over the years further security measures have been taken and something worth mentioning is the introduction of signed and unsigned applets. A signed applets contains a signature that the browser verifies against an independent certificate authority server, that ensures that the author of the applet is known and can be trusted. Unsigned applets have very restricted access and is blocked by default from Java 7 update 51.

Portability is also solved with this approach if the JVM is implemented on each platform then the applet can be executed on every platform. The portability aspect was an important goal for the Java designers and one term that they coined was ”Write-once-run-anywhere”.\[37\]

There exists two types of applets, the first one is based on the Applet class which uses AWT to provide a GUI and the second one is based on the class JApplet which inherits the Applet class but uses swing instead as GUI provider. Applets are not stand-alone program, they run inside a web browser or an applet viewer. We are interested in the web browser runtime since its more suited for the topic of this thesis. When a user navigates to a web site with a applet embedded the applet will automatically download and run. This is why its important to keep applets small to keep the download time low. Java applets are embedded in the HTML document by either the \(<\text{applet}>\) tag which is obsolete so instead the \(<\text{object}>\) tag should be used.\[29\]

Java applets requires that browser has the Java plugin, without this plugin the applets will not work. This has become a problem since the Java plugin relies on NPAPI, NPAPI has as described in chapter 3.2.1, started to loose support among the major browsers, meaning that applets will for example not work by default in Google Chrome 42.
3.3.6 Add-ons security

Below add-on will use as an umbrella term for extensions and plugins, this might not be entirely correct as add-on usually refers to extensions.

Add-ons suffer from quite severe security weaknesses since there is no technical measure to ensure security, meaning that the user has the choice of either not using the add-on or trusting the developer. There are two ways that this trust can be exploited for malicious intent.

First is if the add-on is malicious in itself. For example if an attacker write an add-on that logs all content written in forms, and sends back the data, the attacker could get over all kinds of information. In order to get users to download the extension the attacker might advertise it as an update to a trusted add-on. The second way for an attacker to attack add-ons is to exploit security vulnerabilities in add-ons. Here we have a benign add-on developed by someone with no malicious intention’s, but might not be an expert at security, which leads to an add-on with security vulnerabilities that can be exploited by an attacker. \[13\]

In the paper presenting Hulk\[28\] they perform an analysis of browser extensions found in the chrome web store. They identified 130 malicious and 4712 suspicious extensions out of 48,332 extensions. Meaning that about 10% of the extensions found in the chrome web store is malicious or at the very least suspicious. This gives a clear perspective of why we need new technologies that addresses these security issues.

According to Grier\[21\] plugins is the single largest source of vulnerabilities in web browsers.

3.4 Native vs Web Applications

First lets clarify what a web application is because most people might not realize that web sites and web applications are not the same thing. Web applications extends web sites with added business functionality.\[24\] This means in simple terms that web applications allows user input to affect the state of the business. Web sites are primarily used to display information.

Native applications are called native since they are written to work on a specific platform or device. Writing applications for a specific platform opens up the ability to use platform specific capabilities. Meaning that its possible to take advantage of OS features and software that is installed on the platform. Native applications are often written in a language that is a C dialect, that is then compiled.\[20\] This approach with a compiled language has two major advantages, speed and efficiency.

Web applications are not written for a specific platform, and can be used on any platform with a modern browser. They can be written in many different languages, but JavaScript and HTML5 is widely used. HTML5 and CSS is used to create user interface meaning that JavaScript is pitted against the compiled languages used for native applications. JavaScript is
getting faster at a rapid rate and is starting to approach near native speeds but for now at least native code is faster.

Web applications are less expensive to develop, seeing as you only have to develop one application and not one application for each platform your targeting, they are delivered as services and doesn’t require any installation process as native applications does. Native applications have the advantages of speed and efficiency that comes from access to platform specific functionality. However the performance advantage might be negligible if the application doesn’t perform any computation demanding tasks as e.g. 3D-rendering.

To get the performance of native applications and the portability of web applications, technologies that uses a combination of web and native capabilities have started to emerge. In chapter 4 a couple technologies that allow web applications to use native capabilities in order to combine the strengths of both approaches is discussed.
Chapter 4

Theory

Earlier in this thesis a comparison between native and web applications where discussed, and in this chapter we look at technologies that aims to combine the advantages of both approaches in order to create Native Web Applications. Native web applications are web applications that is extended with native capabilities. The first two technologies discussed, Native Client and Xax achieve this by allowing web applications to execute compiled native code in the browser. Finally we look at Emscripten which allows native code to be compiled to JavaScript.

The main goals that native web applications strive to achieve is:

- Performance
- Portability across browsers and OS
- Security

4.1 Native Client

Native Client is a open source Google project that started as an experiment in 2008, developed to let developers create applications running native code safely in the browser. The problem with NPAPI and ActiveX plugins described in [31] is that they circumvent the security mechanisms otherwise applied to web content, while at the same time allowing access to full native capabilities. Given this behavior browser applications using native code must rely on non-technical measures of security, like for example pop-ups.

To solve these issues Native Client separated the problem of safe native execution from that of extending trust. Meaning that native client is organized in two parts, an environment for executing native code to prevent unintended side effects, and a runtime for hosting extensions through which allowable side effects may occur safely.
Today there are two different modules for developing Native Client applications and those are NaCl and PNaCl.

### 4.1.1 NaCl

NaCl stands for Native Client and is Google’s solution to the problem of running native code in the browser securely. It was developed to provide safe execution of native code in the browser across operating systems in order to not sacrifice portability.

#### 4.1.1.1 Architecture

A NaCl application consists of trusted and untrusted components. User interface and NaCl modules (.nexe files) are considered untrusted and the NaCl runtime is considered trusted. As seen in figure 4.1.

When a user navigates to a web site using a NaCl module. The NaCl module will automatically be loaded completely transparently for the user, meaning no pop-ups or any other method of asking the user for permission will occur. This means that Native Client is responsible for ensuring that the NaCl module, which is untrusted doesn’t cause any security issues.

To prevent the NaCl module from doing harmful things NaCl uses a dual sandbox design. Before we get into the details about the double sandbox a basic understanding of the Chrome browser is needed.

### Chrome

Chrome is a multi-process browser and the main process is often simply called browser. The browser process runs the UI and manages things like tabs and plugin processes. With the same permissions as the current user the main process have access to network and file system calls.

Each tab is allocated in a process called renderer. Renderer runs in a sandbox and handles the interpretation of the HTML and other aspects of displaying the page. The sandbox introduces limitations on the access permissions, restricting the renderer from making network and file system calls and it can only respond to calls from the browser. Since each tab is in a separate process this means that they are isolated from each other, meaning that if one tab crashes or runs malicious software no other tab or part of the system is affected. This sandbox will be called the outer sandbox through the rest of this paper.

Since the outer sandbox restricts the use of network and file system calls which is often used by plugins, plugins cannot be run inside that process. Therefore Chrome supports out of process plugins. These processes doesn’t have the same restrictions and can communicate with the renderer and browser process via Inter-Process communication (IPC) techniques.
4.1. NATIVE CLIENT

Chapter 4. Theory

Double sandbox design

Native Client uses a double sandbox to ensure that the system can only be accessed through safe APIs by code. And also to ensure that the code operates within its limits, without attempting to infer with other code running either within the browser or outside it.

The inner sandbox creates a security sub-domain within a native operating system process. To detect security defects in the native operating system process the inner sandbox uses static analysis. To increase the effectiveness of the static analysis Native Client introduces a set of alignment and structural rules, enforcing that the module can be disassembled reliably. The rules has mainly to do with self-modifying code and overlapping instructions. With reliable disassembly the validator can ensure that the NaCl module only contains safe instructions.

As the inner sandbox creates a sub domain within a native operating system process you can place untrusted application modules within the same process as trusted service runtime subsystem, and use a trampoline/springboard mechanism to transfer control between untrusted and trusted code safely. The inner sandbox not only isolates the system from the NaCl module, it also isolates the NaCl module from the system, which is good...
4.1. NATIVE CLIENT

since the OS isn’t in general defect free.

The outer sandbox as described earlier is the Chrome sandbox.

Runtime Services

To provide a way for untrusted and trusted components to communicate with each other Native Client provide Inter-Module Communication or IMC. IMC can be used to share files, share memory objects, communication channels and more, across process boundaries.

The runtime service further acts as a container for NaCl modules, it is a operating system process that provides a set of system calls and other features like POSIX threading.

4.1.1.2 Implementation

This section will go into more details about the various components in NaCl.

Inner Sandbox

The inner sandbox is the main security measure in NaCl and it consists of three components. A set of rules for reliably disassembly, a modified compilation toolchain based on GNU gcc/g++ that follow these rules, and a static analyzer that ensures that the rules was followed. The toolchain is further discussed in section 4.1.1.4. By having the compilation tool outside of the trusted code base (TCB) and a validator implementation that requires less than 600 C statements\[31\], the Google team managed to decrease the size of the TCB significantly. The trusted code base contains the code significant for ensuring security of the system and is preferably kept small to keep costs of review and testing down.

The validator guarantees four properties essential for a secure NaCl application:

- Data integrity
- Reliable disassembly
- No unsafe instructions
- Control flow integrity

Data integrity is guaranteed by use of software fault isolation (SFI). What SFI does is to create a sandbox that restricts the NaCl modules access to memory. Native Client currently have sandboxes for ARM 32-bit, x86-32, x86-64 and MIPS. The specific implementation details vary depending on the architecture and I will not go into the details, but the main idea is to isolate the untrusted NaCl module from accessing and altering the trusted runtime. For more details about the sandbox internals see Appendix A1.
Reliable disassembly is achieved by enforcing that binaries loaded into memory is not writable during execution and that all valid instructions must be reachable by a fall-through disassembly starting at the load address. Once you have reliable disassembly the prevention of unsafe instructions is straightforward, some examples of disallowed instructions are instructions that invoke operations on the operating system directly, like syscall, use of ret instructions, all returns are implemented with special sequence that ends up with an indirect jump.

The final problem is control flow integrity which is ensured by checking that control transfers in the program only target instructions identified during the disassembly.

The time it takes to perform the validation is insignificant compared to the time it takes to download the NaCl application. According to Google [31] the validator can check code at 30MB/second which means that the validation time will be very small compared to the download time, since most users cant download content near 30MB/second. According to Akamai the average connection speed is around 3.9MB/second.\[10\]

The whole process of the static analyzer is shown below in figure 4.2.

![Figure 4.2: Static analyzer NaCl inner-sandbox](image)

### Outer sandbox

The outer sandbox in NaCl is used as a second level of security in Native Client. If the inner sandbox should fail the attacker would gain access to the service runtime but the outer sandbox prevents access to the rest of the system. The outer sandbox mediates all system calls from the NaCl application to the underlaying operating system and validates them against a whitelist of allowed calls. If a disallowed system call is used the NaCl module will terminate immediately.

### Service Runtime

The service runtime is a native executable that runs in the same process as the untrusted NaCl module and is maintained by the OS. A certain NaCl browser plugin invokes the service runtime by asking the browser to
create a process for the service runtime. The NaCl plugin also controls the interactions between the browser and service runtime.

The runtime creates the inner sandbox and controls what calls the NaCl module can make out from the sandbox. For example it allows calls that draw graphics but prevent direct system calls. The call from a NaCl module to a resource outside the NaCl module is called a trampoline. In reverse a call from trusted to untrusted code is called a springboard. This trampoline/springboard mechanism is considered trusted as its installed by the service runtime, to prevent untrusted code to use the springboard mechanism certain alignment rules are implemented.

Messaging System

To provide bi-directional communication into and out of NaCl modules IMC is used. IMC uses a NaCl-socket, which provides reliable in-order datagram service. When a NaCl module is created it receives a NaCl-socket which is accessible from JavaScript via the Document-Object Model (DOM) used to create it. This socket is used by JavaScript to send messages to the NaCl module.

NaCl’s Simple Remote Procedure Call (SRPC) abstraction is implemented as untrusted code, and provides a way to declare interfaces between JavaScript and NaCl modules, or between two NaCl modules. It supports a few simple types (int, char, float) and arrays.

On the NaCl module side of the communication the methods used to handle incoming and outgoing messages are:

```c
void pp::Instance::PostMessage(const Var &message)
virtual void pp::Instance::HandleMessage(const Var &message)
```

PostMessage sends messages to JavaScript and HandleMessage receives from JavaScript.

In order to not freeze the web page on the JavaScript side while waiting for a response from the NaCl module the messaging system is asynchronous. This means that as soon as JavaScript is done with sending the message it returns, instead of waiting for a response. An event listener is setup on the JavaScript side to handle messages sent by the NaCl module that will, when it has finished processing the request return a message.

Exceptions

Native Client supports C++ exceptions but not hardware exceptions like segmentation faults and floating point exceptions. Forcing the developers to prevent all possible occurrences of hardware exceptions since the application will not be able to recover from e.g. a segmentation fault, which will cause an abrupt termination of the application.
4.1.1.3 Structure of a web application

A basic Native Client application consists of three things. The first thing is HTML and JavaScript files, an embed element needs to be inserted in the HTML either directly in the HTML file or through JavaScript, this element is where the NaCl module will be loaded, in appendix B1 an example embed element is presented to demonstrate how it might look. This embed element tells the browser where to find the manifest file which is the second thing needed. The NaCl manifest file (.nmf) identifies which module to load, based on the client machine’s architecture. The last thing needed is a native client module, a .nexe file. The .nexe contains the compiled C/C++ code, and uses PPAPI to provide a bridge to JavaScript and other browser resources.

A NaCl module written in C++ must include three components, a factory function called CreateModule, a Module class and an Instance class.

NaCl modules don’t have main functions as regular C++ programs instead the factory method acts as the entry point into the module. When the browser loads a module the CreateModule function is called and thereby becomes the binding point between the browser and the module. Each module includes a CreateInstance method that is called for each <embed> element that references the same module. Appendix B2 shows how a simple NaCl module could look.

4.1.1.4 Toolchain

The NaCl toolchain uses a GCC based toolchain that produces a set of architecture dependent modules, so called .nexe files, which are packaged into an application. At runtime the browser determines which .nexe to load based on the client machine.
As we see in Figure 4.3 we have a problem if a new architecture is introduced, as it require us to compile the source code again to target the new architecture, this is not convenient nor portable and causing NaCl to be ill-suited for the web.

### 4.1.2 PNaCl

Portable Native Client (PNaCl) is a project that aims to provide the same level of security and performance as NaCl but also address the portability problem that NaCl has. The problem is that NaCl modules are tied to a specific architecture, and require thus that applications needs to be compiled once for each architecture it’s deployed on. PNaCl resolves this issue by using an alternate toolchain that produce a .pexe file and is further described in section 4.1.2.1.

On the browser side something that Google call a translator which is built into the browser, translates the .pexe into native code for the specific instruction set architecture used by the client.

Now does this mean that if you open an arbitrarily number of tabs and all of them load the same PNaCl module do you have to compile the same .pexe multiple times?

No, in order to avoid this re-compilation step each time you load a PNaCl module, the translator has a cache. So if the .pexe has been compiled before on the client browser, no compilation will occur. [7]
4.1.2.1 Toolchain

The PNaCl toolchain uses LLVM, which stands for Low Level Virtual Machine and is an infrastructure for compilation. PNaCl uses LLVM to first compile the source code, that can be written in any language that can be compiled to bitcode. This compilation produces LLVM bitcode that then passes through an optional optimization and linking step before being shipped as a portable executable, a .pexe file.[14]

Figure 4.4: Simplified scheme showing steps from source code to running PNaCl application

As seen in figure 4.4 the problem that NaCl had when a new architecture was introduced no longer exists since the .pexe is architecture independent.

4.2 Xax

Xax is a Microsoft research project defining a browser plugin model with the goals of providing security, OS-independence, performance and legacy support. To achieve these goals Xax uses four mechanisms, picoprocesses, a platform abstraction layer, browser based services and lightweight code modification.
4.2.1 Architecture

The architecture of Xax could be said to have three parts. The first part is a web browser. The web browser communicates with an Xax Monitor which is the second part and is a part of the browsers trusted code base. The Xax Monitor is in charge of creating and managing picoprocesses, which is the third and untrusted part of the Xax architecture. Figure 4.5 shows the basic architecture of Xax.

The first thing the Xax Monitor does is to allocate a set of shared memory used as a communication channel between it and a picoprocess. Picoprocesses run as a child process of the Xax Monitor and starts by executing a boot block which is OS-specific. The boot block maps the shared memory into the child process and then issues a kernel call that revokes the child process privileges to make kernel calls and thereby isolating the child process. The last thing it does is to pass execution to PAL that loads and passes execution to the Xax application and thereby finishing the creation of a picoprocess. The boot blocker is part of the TCB as the child process isn’t really a picoprocess until the boot blocker has passed execution on.
The Xax Monitor communicates with picoprocesses through xaxcalls.

4.2.2 Picoprocesses

A picoprocess can be thought of as an OS process that acts as a container for executing native code. In order to assure security and make it easier to allow operating system independence a picoprocess has a very narrow interface and is restricted from making kernel calls. A picoprocess communicates by making xaxcalls to a Xax Monitor. Xaxcalls are analogous to syscalls. Xax Monitor is part of the browsers trusted code base, and uses OS services to create and manage picoprocesses an is thereby OS specific.

4.2.3 Platform Abstraction Layer

Xax defines an application binary interface (ABI) to allow for OS independence. The ABI is OS independent as it only defines the interface not the implementation. The interface is the same for every OS but the implementation varies depending of the underlaying OS. To translate the OS independent ABI into OS-specific xaxcalls of the xax monitor, xax has a platform abstraction layer.

4.2.4 Services via browser mechanisms

"A key Xax principle is that there is sufficient functionality within the browser to support the system services needed by web applications." Xax reuses the existing security policy already employed by the browser, meaning that xax doesn’t introduce any new security vulnerabilities other than bugs in the Xax monitor. The Xax Monitor provides services indicated by xaxcalls, which includes easy calls as memory allocation and deallocation, but also a communication path to the browser.

4.2.5 Legacy code modification

To port a legacy application to Xax, there is a five step design pattern you need to follow.

1. First step is to disable dependencies on irrelevant components, all libraries aren’t necessary for use within the web application framework. Disabling dependencies reduces the size of the web app and the amount of code that needs to be ported. To disable dependencies certain compiler flags can be used.

2. Second step is to restrict the interfaces used by the application. Favor interfaces that needs less support from the system.

3. The third step is to identify sysscalls made by the application that can be handled trivially, like sending back an error message when rejecting sysscalls.
4. The fourth step is to handle the syscalls that cannot be handled trivially. These calls need to be handled by emulating the syscall functionality internally.

5. The final step is to create xaxcalls for the remaining syscalls that cannot otherwise be handled.

4.2.6 Browser integration

Xax is integrated in the browser as a HTTP proxy and will therefore work on any make of browser. From the browsers point of view the Xax application appears to be part of the origin server, which ensures that picoprocesses is governed by the same origin policy. Most of the HTTP requests will be passed to the specified host, completely transparent, however if the URL begins with xax the proxy will direct the request to an existing picoprocess or create a new picoprocess.

4.3 Compile native to web

So far we have looked at technologies that takes native code as C++ and then embed or link it in some way so that the browser can understand it. But you can also compile native code to JavaScript. There are a few frameworks that does this including Mandreel and Cheerp, but I will not get into details about those and instead focus on Emscripten which also compiles C/C++ code to JavaScript and is further described below.

4.3.1 Emscripten

Emscripten is a Open Source project that compiles LLVM to JavaScript. LLVM is a project with primary focus on C, C++ and Objective-C, that compiles those languages through a compiler into a LLVM-intermediate representation (LLVM IR). The Emscripten compiler (emcc) uses clang which is an extremely fast C, C++ and Objective-C compiler. This LLVM IR is passed to a backend that generates machine code for the particular architecture. Emscripten has a component named Fastcomp which acts as the backend and compiles bytecode to JavaScript. Fastcomp has a focus on asm.js code generation, asm.js is a subset of JavaScript with significantly higher performance and is further discussed in chapter 4.3.2. The steps from C++ code to JavaScript is shown in figure 4.6.

This method allows for two ways of native languages to be run on the web, either by compiling code or compiling the runtime of a language. Both approaches uses a two step compilation, in the first you have code written in a language that can be compiled into an LLVM IR and then compile that into JavaScript. In the second alternative you have code written in a language that cannot be compiled to LLVM but the languages runtime is
written in a language that can. This means that you compile the runtime to JavaScript and then you can run the unsupported code in the runtime.[44]

4.3.2 asm.js

Asm.js is a strict subset of JavaScript that uses ahead-of-time (AOT) compilation to allow an optimized compilation strategy. JavaScript has historically been implemented as an interpreted language, meaning that a program executes without previous compilation to machine language. But recently JavaScript engines have started to perform just-in-time (JIT) compilation, which is simply put compiling during execution. The introduction of JIT compilers in JavaScript increased the performance drastically, when Google first launched Chrome with the V8 JavaScript engine the execution speed of JavaScript increased with 150%. [25]

A developer is not intended to write asm.js code directly, the intended
use is to use a compiler like Emscripten to generate asm.js code. The code generated is valid JavaScript but may look very different than handwritten JavaScript and may even in some cases be unreadable.

Asm.js uses both static and dynamic validation to validate the asm.js code, if the validation succeeds the JavaScript engine can use an AOT optimizing compilation strategy, if however the validation fails JIT compilation or interpretation must be used. [26]

4.4 Comparison

In this paper we have looked at four methods of running native code in the browser, plugins, NaCl, Xax and with Emscripten. In many aspects there are similarities in their goals and methods of achieving them, but in other aspects they differ. Focus of the comparison will be on NaCl and Xax, since their approach is most similar and also the most different from the methods used previously.

<table>
<thead>
<tr>
<th>Browser integration</th>
<th>plugin</th>
<th>NaCl</th>
<th>Xax</th>
<th>Emscripten</th>
</tr>
</thead>
<tbody>
<tr>
<td>New tool chain</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>OS-independence</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Allows socket and file access</td>
<td>yes</td>
<td>no*</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Supported programming languages</td>
<td>multiple</td>
<td>C/C++</td>
<td>C/C++,Java,Python</td>
<td>C,C++</td>
</tr>
<tr>
<td>Support multi-threading</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Security method</td>
<td>user acceptance</td>
<td>dual sandbox</td>
<td>process isolation</td>
<td>JS restrictions</td>
</tr>
<tr>
<td>Works on all major browsers</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Requires modification to legacy code</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 4.1: Describing features for the different technologies.

Xax and NaCl use different browser integration methods, Xax has gone with a proxy based approach while NaCl is implemented as a plugin. The advantage with a proxy based approach is that it works on all major browsers, while NaCIs plugin approach is for now at least limited to Google Chrome. The plugin approach however has the advantage that it allows for a greater control over native modules and a deeper integration with the browser.

NaCl introduces a set of structural and alignment rules on its applications, requiring a new tool chain that follow these rules. In order to provide the means of compiling and linking NaCl applications, they modified the standard GNU tool chain. Thereby causing legacy applications to need porting in order to be applied in NaCl applications. Xax meanwhile have only done very lightweight modifications to existing tool chains, requiring no porting.

Both NaCl and Xax achieve OS-independence with fairly similar methods. Both restrict the use of OS-specific system calls and have created an inter-
face that reuses the browsers functionality. NaCl uses its service runtime to achieve this and the corresponding component in Xax is the Xax Monitor.

NaCl does not permit socket and file system access by default. It can be achieved using the nacl_io library but not with NaCl itself. Xax on the other hand does permit network and file system access.

Threading is not supported in Xax, which is a significant setback compared to NaCl that supports C11/C++11s threading libraries and POSIX threads. Multi-threading allows you to distribute the workload for demanding computations and has several advantages e.g. faster execution and responsiveness.

When it comes to security Xax and NaCl have very different approaches, Xax relies on process isolation, using picoprocesses that are restricted from making OS system calls. Google however took it further than restricting OS system calls, they even organized a security contest to test the security of Native Client[6]. NaCl employs a dual sandbox method where the inner sandbox perform static analysis on untrusted code and the outer sandbox checks system calls against a whitelist, calls not whitelisted causes immediate termination of the application.

As a rule both Xax and NaCl needs modification to legacy code, although legacy code may not require modifications. With Xax you follow a five step guide to port your legacy code, where the main difficulty is how to handle OS system calls. If your application does not perform OS system calls then porting your application is straightforward. Same thing goes with NaCl, if you have no OS system calls or only OS system calls that are whitelisted, porting legacy code is straightforward and may require no modification.

Emscripten compared to NaCl is slower since it runs JavaScript code in a JavaScript engine which will not achieve near native speeds, as seen in the performance results shown in table 6.1. However since it is JavaScript code that runs in the browser it works on any browser and have the same security as JavaScript.
Chapter 5

Case study

There are many different areas where technologies like NaCl would be extremely useful. Native code brings especially performance to the web but also the ability to access OS level information and functionality. Giving you a number of areas where this would be useful

- 3D rendering - Create cool animations or games using 3D engines
- Newtonian physics - Simulate heavy computation physics in the browser
- External devices - Access to external devices, like USB, NAS
- Security - Gives you superior security compared to traditional plugins and extensions.
- UUID - You could create a way to recognize a user even if the user uses multiple different browsers on the same computer. As you can with Android or iOS for example.
- File system access - Access files on the system

5.1 Theory

The application I have created gives a browser the ability to discover and communicate directly with a Media Server.

5.1.1 Requirements

The requirements for this application was

- Be able to discover a Media Server
- Be able to list discovered Media Servers
- Be able to trigger actions on a Media Server
5.1. Theory

• Be able to browse content of Media Server
• Be able to display Media Server file structure
• Be able to render video contents of Media Server
• Be able to render picture contents of Media Server
• Application should run in browser
• Application should use PNaCl

5.1.2 Universal Plug and Play

Universal Plug and Play or UPnP is an architecture defining a set of standards enabling the communication of network devices. These devices can either be a PC, NAS or an intelligent toaster it doesn’t matter, as long as the device implements the set of standardized protocols that UPnP uses. The first version of the UPnP specification was developed by Microsoft and approved in 2000.

When talking about UPnP you often talk about devices, services and control points which is the core abstractions of a UPnP architecture.

• A device is an entity implementing the specified protocols required by the UPnP architecture.

• A service has a set of methods that are called actions. Each action has a set of optional input and output parameters and also an optional return value. A service is implemented in the device and each device can have multiple services or none.

• A control point is an entity on the network that works with the functionality provided by the device, similar to a client in a client/server architecture. Control points can invoke actions on devices.

These three different components follow a pattern in their interactions with each other. Control points discover devices and invoke actions on a devices services. Control points can also listen to state changes in the device.

Phases

To give some more structure on how the components communicate you can say that there is six phases in UPnP:

• **Addressing** is the process of how a device that just joined the network acquires an IP address, so that it can start communicating with other devices.
5.1. THEORY

- **Description** is an XML-based summary of what functionality a device and its services provide. It contains information about things like make and manufacturer of the device and a detailed description of its services and their actions.

- **Discovery** is the process that allows control points to find devices and retrieve information about them. Devices include their description in discovery response messages, allowing control points to find out all the information about the device and its services.

- **Control** is what actions invoked on services by the control points is called. A service receives a control message and acts according to the information in the message.

- **Eventing** is a way for the device to notify the control point that its state has changed. For example, if a control message causes the device's state to change, the device's service sends out a notification to all control points that subscribe to the service.

- **Presentation** is a process for the device to present information about itself in a browser-based user interface. Some devices provide a web page where you can see information about the device, or even control the device. The information available varies depending on the device.

5.1.3 UPnP A/V

For audio and video UPnP have a slightly different architecture normally called UPnP A/V. The architecture is based on the traditional UPnP architecture but introduces two new components, a media server and a media renderer. As seen in figure 5.1.

The media server is where the audio and video content is stored, it can be either stored locally in the media server or you can access externally stored content through the media server. The main purpose with the media server is to allow the control point to browse the content stored on the server. The content is transmitted to the media renderer using a transfer protocol understood by both the media server and the media renderer. The process of selecting a language that both parties understand is handled by a Transcoder, which is further explained in 5.1.4.

The media renderer has the purpose of rendering content transmitted by the media server. It also allows the control point to control properties of the rendering, like brightness and volume. Depending on the transport protocol used, the media renderer can also let the user control the flow of rendering, like play, pause, etc.

The control point in the UPnP A/V handles the coordination between the media server and media renderer. Meaning that the user accesses the control point and the control point in turn handles the discovery and communication with the media server and media renderer. The control points
main tasks is to discover A/V devices, locate desired content, select a transport protocol that both media server and media renderer understand, select the desired content and start the playback of it. The control point doesn’t transfer the content from media server to media render, it only configures the devices and triggers the transfer, the actual flow of content is handled using an out-of-band protocol (non-UPnP) between media server and renderer.

Further the control point handle user actions like change of brightness or selection of new content, the final thing the control points does is to cleanup by closing connections to the media server, this can be done as soon as the transferring of content starts since the control point isn’t involved in the flow of content.\[9\]

Figure 5.1: UPnP A/V Architecture

5.1.4 Transcoding

Transcoding is needed in an UPnP A/V context for situations where the media renderer cannot render server content. There could be many reasons for this and one example is if the media server has content using a format that the renderer cannot decode. Instead of leaving the renderer with content it cannot understand, a transcoder is used to translate the foreign content. The transcoding process most often takes place on the media server.

There are many techniques for this and I will not go into details, just give a broad understanding of what it is and why it is important to think of when it comes to UPnP A/V.

A video format is defined by a set of properties like bit rate, frame
rate, spatial resolution and content etc. A simple use case for transcoding is to adapt the bit rate between two components with different bit rates. In this example, it’s relatively easy to perform the transcoding, but in the context of UPnP A/V where you have entities communicating over the Internet, it quickly gets more complex. You have to take into account things like bandwidths, bit error rates, packet loss, and you also have to factor in computational power, display capabilities, and more. 

5.1.5 External device

In order to be able to test the application, an external device was needed to run it against. Here, there exists several alternatives, you could use a NAS, SmartTV, XBox, Playstation or basically any device that supports UPnP. We decided on a media server.

There are many alternatives for setting up a media server, some popular choices are Plex and Kodi. Both these have a powerful and visually appealing user interface for managing your content and support lots of features. Since I was just building a prototype application, I didn’t need a heavy media server that supports every feature, only needed the basics.

I decided to use a software called MediaTomb. MediaTomb is an open-source UPnP MediaServer with a web user interface for easy management of content. Transcoding is supported and can be configured with a bunch of settings. I setup the media server on my computer, it was very straightforward and required very little configuration to get it up and running.

5.1.6 The NaCl IO Library

Earlier in this paper, I have described the Pepper API. NaCl IO is a utility library built on top of Pepper API and is included in the main NaCl SDK. The purpose of NaCl IO is to allow implementations that use standard C APIs like POSIX I/O and BSD sockets to be compiled and used within Native Client modules.

Using NaCl IO allowed me to use Socket API in the standard way, the only thing that you have to do is to link it with the NaCl IO library, initialize it at startup using a special function and to run all my socket calls on a background thread. This is required since the main Pepper thread doesn’t support the blocking behavior needed by POSIX. This also means that use of PPAPI calls is out of the question and that your code will look like traditional C code.

If you create an application that needs to make file system calls, you would have to mount the desired file system using the mount function, I didn’t need this in my application.
5.1.7 Mini-XML

Since I would have to parse XML files and SOAP responses which also is XML I included a library called mini-XML. I decided on mini-XML since it had a small code base which I thought would be easy to port to PNaCl. I only needed to do simple operations on the XML so I didn’t need a heavy XML parser.

5.2 Implementation

I have created an application that can playback content stored on a media server. Content like pictures and videos. The application was created externally from the VIA Web product and then included it in VIA Web when the application was done.

As described earlier there are several steps involved in UPnP, the discovery and control phase is most significant to my application so below I explain in greater detail how I handled those two phases.

5.2.1 Discovery

The discovery process is done with the use of the SSDP protocol which is Simple Service Discovery Protocol. A protocol that is based on HTTPU which in turn is an extension of HTTP/1.1 protocol but instead of using TCP as normal HTTP, HTTPU uses UDP as transport protocol.

There are two ways for a control point to discover devices. First way is by intercepting alive messages. When a device is added to the network it announces its services by sending an alive message that control points can intercept. Since I have created a control point and not a device I have used the second alternative which is to send a message to a specific multicast address and port. The devices that receive this message will send a response with limited information about the device for example, type, identifier and a location to where a more detailed description about the device can be found. Using this location I can retrieve the description XML to get information about what services the device has and also how these services are used.

After retrieving the description XML I ran it through a parser to get the information that I needed. The parsing of the description XML meant that the discovery of devices was done. I now know which devices that are available, there name, type and location etc.

5.2.2 Control

Included in the description XML of a device are the fields SCPD and controlURL. The SCPD stands for Service Control Protocol Document and the value of this field is a location to where the XML that describes the various services can be found. For example on a media server one service could be
a content directory, in the SCPD for the content directory various actions and states are described. So if I for instance want to invoke a browse action on the content directory, I can see in the SCPD what arguments I have to include in my request. Then send the request which is a SOAP message defining what action I want to invoke and with which arguments. This request is sent to the controlURL. As response I will get a XML file that shows all files and folders in the directory I requested. As initial request you use the id 0 which will give you the root directory.

5.2.3 Accedo VIA

Accedo VIA is implemented in JavaScript using AngularJS as frontend, since my application only deals in frontend code the backend used doesn’t matter that much. AngularJS is a frontend MVC framework that uses data binding and dependency injection to reduce the code you have to write, and defines concepts like Controllers and Services.[12] Controllers are view-dependent and is used to handle the business logic behind views. Controllers are usually kept thin and often encapsulate work that doesn’t belong in the controller into services.

Services are mainly used to share code across the application and contains reusable business logic that is independent of the view. If you want to use the functionality that a service provide you, add it to your controller (or any other component) using dependency injection.

I defined a NaCl service in my project that handled all the NaCl specific tasks like communicating with the NaCl module. By having the NaCl specifics in a service this allowed me to use it from multiple views. When I needed to use any NaCl specific functionality I included the NaCl service as a dependency in the views controller and thereby got access to the functionality of the NaCl module.

Since I built my application externally to the VIA product, I had to integrate it into the Accedo VIA product afterwards. This meant in turn that I had to do some rewriting. I already had all the communication with the NaCl module in a separate JS file and the view specific logic in another file, this meant that I pretty much had the service done already, I only wrote some AngularJS specific code to define the service. So the main part of the rewriting was to split up the view-dependent business logic between controllers, this was not a time consuming task. The integration of my external PNaCl application into the Accedo VIA product was very easy. This had a lot to do with the fact that I only had JavaScript code that I needed to include since the NaCl module already was done and ready to be used.

5.2.4 PNaCl

As said before I used PNaCl for my implementation, what I mean with is that I used the PNaCl tool chain to compile and link my application. The
version of the pepper SDK that I used was 42. The development itself turned out to be roughly the same as for a standard C application since I had to use the NaCl IO library, only difference was the compilation process.

Obstacles

During the development I have encountered many obstacles. All of them are not relevant for this report, so I will discuss a few of them that stems from the architecture and implementation of PNaCl.

At first when I started this project in order to save time I wanted to use an existing library for UPnP communication, but this turned out to be more difficult than I expected. I looked at three different libraries, libupnp, gupnp and PlatinumUPnP. I had problems with all of these, it was either that they where dependent on libraries that are unsupported by PNaCl, like glibc \[ 4 \] or it was that the library required a lot of porting to be complaint with PNaCl. Especially parts regarding sockets required porting. Since a major part of the UPnP communication happens over sockets which is heavily restricted by PNaCl, I deemed it more time efficient to use the NaCl IO library and to write the socket communication parts from scratch.

Another downside with PNaCl is that, since I used the Socket API and NaCl IO you can no longer distribute your application the standard way as you would with a normal PNaCl application, seeing as you have included sockets in your project you have to distribute your application as a packaged app. This technically mean that you loose the advantages that PNaCl has over NaCl.

5.3 Notes

Since I started the project, Google has released new versions of the pepper SDK, and has in pepper version 43 started a preview to add UDP Socket Multicast API which could have been helpful in my application.
Chapter 6

Results

The results of this thesis is a working prototype of an application created with PNaCl. The application can discover and communicate with a media server and runs in Chrome.

The basics of different technologies have been presented and performance testing have been performed. Below I will discuss the results regarding the application in greater detail as well as the performance results.

6.1 Application

The application created was a working prototype built using the PNaCl toolchain, NaCl IO library and the mini-XML library. It successfully allows the application, that runs in Chrome to discover and communicate with media servers on the network and fulfills all the requirements that was setup for this project. The use of NaCl IO and the Socket API restricted the distribution options for the application, since usage of the Socket API is restricted to packaged applications distributed in the Chrome Web Store. During development you can workaround this issue by setting the "–allow-nacl-socket-api=<host>” flag when launching Chrome, but this was not the wanted behavior for the integration with VIA Web.

6.1.1 NaCl SDK

Short notes about my experience working with the NaCl SDK. It is perfectly possible to port and create applications using the NaCl SDK, but I have experienced a few areas that I think need some special attention.

Debugging

NaCl supports different kinds of debugging, if you develop on a Windows platform you can use Visual Studio and the Native Client Visual Studio
add-in, that allows you to write and debug Pepper plugins and Native Client modules. If you however sit on a different platform like I did during this project, I used Ubuntu you cannot use the Visual Studio add-in.

Included in the SDK is a command line debugger based on the GNU gdb debugger. The method involved to use this debugger I experienced as slightly complex and I had some problem getting it to work properly. In order to use the nacl-gdb debugger you have to compile your native client module with the flag -g, this tells the compiler to include debugging information in the results, next you launch the web server and chrome, Chrome needs to be launched with a series of flags, among them a flag that turns off the Chrome sandbox. After this the debugger can finally be started, then you have to run a series of commands in the debugger in order to set the target to Chrome. Now the debugger is setup and standard gdb commands can be used.

In Appendix B3 the required steps to use the nacl-gdb debugger is shown.

**Documentation**

I found, rather surprisingly, that the documentation about PPAPI was quite poor in some cases. The documentation that exists didn’t give me sufficient information about how to use functions in PPAPI, and I often went and had a look in the source code to understand how to use a certain function.

The SDK does however contain a lot of very good examples of how various components of the PPAPI could be used.

### 6.2 Porting legacy code

Google themselves state that porting legacy code to Native Client is an effortless operation. During this project I have attempted to port several external libraries, to some degree I can agree with Google’s statement. What I have found is that libraries that don’t use socket or file I/O calls require little or no modification, but however if you are forced to use the Socket API as I was it gets significantly more challenging. NaCl blocks these APIs for security reasons and points you towards the NaCl IO library that allows you to use these APIs but restricts you to distributing your application as a packaged chrome app.

### 6.3 Performance

The performance of JavaScript depends heavily on the JavaScript engine used by the browser, we have touched this subject earlier in this thesis but never really explained what it is. A JavaScript engine is a virtual machine that interpret and executes JavaScript, as mentioned earlier they evolved during 2008 when the JIT compilers came into the world of JavaScript and
suddenly JavaScript became significantly faster. Today there exist many
different engines and the major browsers use different engines that can dif-
fer significantly, this it is important to run the performance tests on different
browsers.

The main question that I want to answer is:

   How does the performance of native web applications compare to native
   applications and to JavaScript?

Test Program

I did not compare a native version and a web version of the application
created in the case study for a few reasons. First since that application
uses sockets, and especially blocking socket that wait for a response, thus
leading to unpredictable time consumption. Even if the socket part of the
application was predictable the application is lightweight and therefore not
suitable for benchmark testing. So instead a library called Box2D have been
used, which is a physics library that is used frequently in games to simulate
realistic movement of objects.[19] This allowed me to test a more compute
intense task which is good in order to get good benchmark results. What
the test does is to create a pyramid with a base of 40 boxes, resulting in a
total of 820 boxes stack on top of each other in a pyramid.

I haven’t written the test cases myself I have used test programs written
by Webber.[42] Webber tests more VMs than we have covered in this thesis,
but those that are relevant to this thesis is included except Xax which is
unfortunately a closed research project and I have thereby not been able to
run tests for it. Its a shame since it would have been very interesting to see
how Xax stands against NaCl.

Webber also documents results, the latest results he presents are from
May 2014 and can be seen in table 6.1. The table have four columns showing
the mean, 5th and 95th %ile and a ratio to C. I used his program to run my
own tests and got similar ratios, although they where most often slightly
higher, which could be due to that my results where consistently slower
than Webber’s, this is most likely because of differences in the test platform.
Webber used a MacBook Pro 2,3GHz i7, 16G memory, running Mac OS X
10.9.2 and Windows 7. Since the results he presents covers more browsers
than me, I used a 2.67GHz i5 4G memory machine running Ubuntu 14.10,
limiting the browser alternatives, the results presented below are Webber’s.
Table 6.1: Results that Webber presents in [43].

Emscripten is included under asm.js since the code generated by Emscripten is asm.js.

The test results show that PNaCl achieves within 20% of native speed which is what Google themselves state.

Emscripten and asm.js on Firefox show very impressing results with a ratio of just 1.5x, which put it very close to PNaCl.

Interesting to see is that Firefox performs better than the other browsers on every test, indicating that their JavaScript engine is the fastest at processing these kinds of calculations. If we look at results presented at the web page arewefastyet [2] we can also see that Firefox performs best on a number of benchmark tests.

Safari displayed severe issues with asm.js during Webber’s tests, Safari performed so bad that is has been excluded from Figure 6.1 in order to make the results readable.
Analysis

It is clear that browsers are getting faster JavaScript engines, just by running the tests on Firefox 35 and Firefox Nightly(40) you could see that nightly was 10% faster then version 35. This trend can bee seen if we look at Webber’s results again, from his results from 2013 and the results from 2014, we can see that everything is getting faster. During just a year JavaScript on Firefox went from a ratio of 12.0x to 5.1x.

I mentioned earlier that my results had similar ratios but most often slightly slower ms/frame and slightly higher ratios than Webber’s. Something worth mentioning here is that my results for asm.js on Chrome stood out in my tests, since my results gave a lower ratio compared to Webber, the ratio I got was 2.2x ,which is 0.3x lower then Webber’s results, which was by far the biggest improvement that my results showed compared to Webber’s, this could indicate that Chrome have worked on optimizing asm.js performance, since I used Chrome version 40 and Webber version 33.

One important observation that you can make Webber’s results [43] is that if you need every bit of performance you possible could get, its clear that you need to go with native code. In Figure 6.1 we can see that there is

![Figure 6.1: Shows ratios against C for different technologies](image)
a cluster of three technologies that performs better than JavaScript and in Webbers results \cite{43} we see that the same three performs much better than other technologies as well. These are C, PNaCl and asm.js, all of these take C/C++ as input, which implies that for high-performance applications this is the way to go.
Chapter 7

Analysis

As stated in chapter 2 there where a few goals setup for this thesis.

A comparison between different technologies, based on their features, portability and feasibility.

Several technologies have been examined and even though many technologies have been left out of this thesis a comparison between some of the most promising or widely used technologies have been performed. At least 3 different methods for deploying native code in browsers have been discussed. Native Client with its new toolchain and plugin based approach, allowing for secure and high performance execution of native code in the Chrome web browser.

An application using PNaCl that allows communication between an external device and a browser.

An application was created that achieved this, restricted to be distributed as a Chrome packaged application the integration with Accedo VIA wasn’t exactly what we hoped for but for demonstration purpose of the technology, the application fulfills all the requirements.

A performance comparison between a native and a web version of an application.

Performance benchmark test results was presented, the tests where performed using C, JavaScript, Emscripten and PNaCl, unfortunately Xax is a closed research project and was thereby excluded from the performance testing.
Chapter 8

Conclusion

The goals for this thesis was to investigate different technologies that can be used to run native applications in the web browser and to develop an application using PNaCl in order to get an understanding for the technology.

I think I have covered much of the basics for a few different technologies, I haven’t introduced all technologies out there but I think I have touched upon the most provident technologies. Me and my supervisor at Accedo decided before the start that PNaCl should be used to develop an application applying the UPnP architecture, looking back PNaCl wasn’t a perfect fit for the project, because of the socket restrictions. So either a different technology would have been better suited or an alternative application not using sockets.

Despite the socket restrictions I managed to develop an application applying the UPnP architecture, that did make use of the PNaCl toolchain but with the restriction that it cannot be distributed as nothing else than a packaged app through the Chrome Web Store. The application was successfully integrated with the Accedo VIA product, and successfully demonstrates how an native application written in C can be used in a web browser by the use of PNaCl.

Google state that its possible to port legacy code to Native Client with minimal effort \cite{31}, I can see that this would be possible if you are familiar with the NaCl SDK and that the code in question don’t make use of a lot of network and file system calls. If however you are unfamiliar with the SDK it could take you a significant amount of time to just get the program to compile.

Native Client is a great technology that tick many boxes regarding the goals for a native web application. It implements a dual-sandbox model to provide a compact security framework and is portable across multiple architectures. NaCl also manages to bring high performance to the web, with a performance around 20% slower than native speed, it opens up for a range of new functions for the web, that are to computation heavy for
8.1 Future Work

Some improvement suggestions that I got from working with NaCl is first off all the documentation for PPAPI, the SDK includes some great examples that shows how the various functions could be used, but I would want better information on the documentation pages.

Access to external devices such as serial ports, camera devices and microphones is currently listed as coming soon on the native client developer page. Native Client can today only use native resources that the browser can access.

The biggest challenge for NaCl is to get accepted by other browsers. Today the only browser besides Chrome that has accepted to work with the NaCl project is Chromium. If NaCl fails to get accepted by other browsers it will remain a specialized technology for developing special tasks that cannot be achieved by JavaScript, for the Chrome platform. This would be a shame since NaCl has the potential to bring a new generation of web applications to the world.
Appendix A

Further Reading

A.1 NaCl Sandbox Internals

The main security measure in Native Client is the inner sandbox, used by both NaCl and PNaCl. Details about how its implemented on different architectures can be found here:

- x86-64 [https://developer.chrome.com/native-client/reference/sandbox_internals/x86-64-sandbox#x86-64-sandbox](https://developer.chrome.com/native-client/reference/sandbox_internals/x86-64-sandbox#x86-64-sandbox)
- MIPS [https://code.google.com/p/nativeclient/issues/attachmentText?id=2275&aid=22750018000&name=native-client-mips-0.4.txt](https://code.google.com/p/nativeclient/issues/attachmentText?id=2275&aid=22750018000&name=native-client-mips-0.4.txt)
Appendix B

Examples

B.1 NaCl embed object

```html
<div id="listener">
  <embed id="hello_tutorial"
    width=0 height=0
    src="hello_tutorial.mnf"
    type="application/x-pnacl" />
</div>
```

This is the embed object from an example application included in the NaCl SDK, its a simple hello world program.

First we have a div container around the embed element, this is not required but it is the recommended way to do it. There is one good reason for doing it this way, most often you want to attach some sort of event listener for instance a handleMessage listener, if you attach the listener to the embed object the listener would not be initialized before the NaCl module has loaded. If we call the JavaScript postMessage method on this embed element the message will be sent to C/C++ method HandleMessage.

The src attributes value refers to the manifest file that determines which nacl module to load based on the users computer, as seen below.

The type can either be x-nacl or x-pnacl depending on if you creating a architecture specific module or not.
Here we see one example for NaCl and one for PNaCl of a manifest file for the same program. In the left one that shows how a NaCl manifest file looks like we see that for each target architecture we have defined a different .nexe file to load, this is not needed in PNaCl where you point to the .pexe and then let the translator decide which .nexe to load.
B.2 NaCl module components

Listing B.3: NaCl module components

```cpp
class HelloTutorialInstance : public pp::Instance {
public:
  explicit HelloTutorialInstance(PP_Instance instance) :
      pp::Instance(instance) {}
  virtual ~HelloTutorialInstance() {};

  virtual void HandleMessage(const pp::Var& var_message) {};
};

class HelloTutorialModule : public pp::Module {
public:
  HelloTutorialModule() : pp::Module() {}
  virtual ~HelloTutorialModule() {};

  virtual pp::Instance* CreateInstance(PP_Instance instance) {
    return new HelloTutorialInstance(instance);
  }
};

Module* CreateModule() {
  return new HelloTutorialModule();
}
```
B.3 nacl-gb debugger

Step 1: Compile your nacl module with the -g flag
Step 2: Launch a local web server
Step 3: Launch Chrome with the flags: --enable-nacl --enable-nacl-debug --no-sandbox
Step 4: Navigate to your application in Chrome
Step 5: Go to source code directory and run the nacl-gdb command
Step 6: Run these commands:

(gdb) target remote localhost:4014
(gdb) remote get nexe <path-to-save-translated-nexe-with-debug-info>
(gdb) file <path-to-save-translated-nexe-with-debug-info>
(gdb) remote get irt <path-to-save-NaCl-integrated-runtime>
(gdb) nacl-irt <path-to-saved-NaCl-integrated-runtime>

These commands are for a PNaCl application, commands differ slightly for NaCl
Step 7: Run standard gdb debugger commands.
Bibliography


[27] Derek Davidson. Can personal scrum be used for a team of one? Webgate, 2014.


På svenska

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 nömd som upphovsmann 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/

In English

The publishers will keep this document online on the Internet – or its possible replacement – for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

©Nicklas Schultz