Institutionen för datavetenskap
Department of Computer and Information Science

Master's Thesis

An Investigation into the Applicability of Node.js as a Platform for Web Services

Erik Eloff & Daniel Torstensson

LIU-IDA/LITH-EX-A–12/024–SE
Linköping 2012

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Institutionen för datavetenskap
Department of Computer and Information Science

Master's Thesis

An Investigation into the Applicability of Node.js as a Platform for Web Services

Erik Eloff & Daniel Torstensson

LIU-IDA/LITH-EX-A–12/024–SE
Linköping 2012

Supervisor: **Anders Fröberg**
IDA, Linköpings universitet
Erik Hallbäck
Motorola Mobility
Fredrik Johansson
Motorola Mobility

Examiner: **Erik Berglund**
IDA, Linköpings universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Sammanfattning

Det här är en utvärdering av hur **node.js** lämpar sig för att utveckla webbtjänster. **Node.js** är en mjukvaruplattform för att utveckla event-drivna nätverksapplikationer i JavaScript. Dessutom avhandlas språket JavaScript, inrikat mot de programspråksstrukturer som underlättar utvecklingen av event-driven programvara.

Node.js påstås vara en lösning på problemet med mycket stora mängder samtidiga nätverksanslutningar. Dessutom försöker **node.js** undvika flera av de skalbarhetsproblem som kan förekomma i stora webbapplikationer. En utvärdering av plattformen har genomförts för att att kontrollera och undersöka om dessa påståenden håller. Detta har genomförts genom att utveckla en HTTP-boot-server för Motorola Mobilitys räkning. Mjukvaran, vid namn Wellington, används för att hantera konfiguration och distribution av programvara till set-top-boxar.

Dessutom har en undersökning av och jämförelse mellan event-driven och trådbaserad samtidighet (concurrency) gjorts. Slutligen diskuteras mognadsgraden av **node.js** och ekosystemet kring plattformen, bestående av bibliotek och ramverk.

Sammanfattningsvis är **node.js** en intressant teknik och lämpade sig som utvecklingsplattform för Wellington. JavaScript är ett kraftfullt språk och fungerar bra till att skriva event-driven serverprogramvara. **Node.js** lämpar sig också för att lära sig att bygga nätverksapplikationer med en event-driven paradigm.

Abstract

This study investigates the applicability of **node.js** for developing web services. **Node.js** is a software platform for developing event-driven networking applications using JavaScript. Moreover, the language JavaScript is discussed regarding features that facilitate development of event-driven software.

Node.js's selling point is to be a solution to the problem of massive amount of concurrent network connections. In addition, it tries to avoid scalability issues that may appear in large web applications. To verify and investigate if this holds, an evaluation of the platform was conducted by developing an HTTP boot server for Motorola Mobility. The boot server, named Wellington, is used to manage configuration and distribution of set-top box software.

Furthermore, an investigation and comparison between event based and threaded concurrency models has been made. Lastly, the maturity of **node.js** and its ecosystem of libraries and frameworks are discussed.

In conclusion, **node.js** is an interesting piece of technology and it was suitable as development platform for Wellington. JavaScript is a powerful language and works well to write event-driven server-side software. When learning to build networking applications, **node.js** is a good start to do so using an event-driven paradigm.

Acknowledgments

We would like to thank our supervisors Fredrik Johansson, Erik Hallbäck and Anders Fröberg. Karin Malmborg for showing an interest and testing Wellington. Our families for support through our five years at LiU.

Last, we would like to thank each other for good cooperation.

Thank you!

Linköping, June 2012
Erik Eloff & Daniel Torstensson

Contents

1 Introduction

This thesis work has been conducted at Motorola Mobility, which is a multinational company active in several different areas of business. Motorola Mobility is a major provider of set-top boxes (STB) and software to deliver IP-TV, both of which is the main focus at their site in Linköping.

The task has been to replace the existing boot configuration system with a new one, which was to be built by us. The new system was given the name Wellington.

1.1 Background

During development and testing of STBs and software many configurations of hardware and software appear. The software under test must be installed on an STB, which is normally done by using network booting. The STBs have capabilities to fetch the new software in a **boot image** package via multicast stream, TFTP or HTTP. However, the steps required to configure network booting are many and involve manually editing of DHCP configuration files. An error made by one user can lead to the breakage of the boot system for everyone.

The purpose of the system is to replace the current solution of managing boot image distribution. The existing system sends a boot image via multicast continuously and thus the rate has to be limited in order to avoid congestion in the network. There are very few receivers (typically one) of the multicast used in the testing environment.

A prototype HTTP boot server already existed, but a better solution was needed to ease configuration and allow more employees to use HTTP boot. The server to be built should take care of configuration and distribution of boot images. This server should make testing and development faster and less prone to error. Moreover, making more testers move from the old multicast boot protocol to HTTP boot would decrease the network bandwidth required.

The system also aims to make the process of changing boot image on an STB faster and easier via a web interface. A user should be able to use the web interface to upload new boot images. In addition to the web interface, a command line program should offer most of the same functionality to allow for automation of the system configuration process.

A list of requirements is presented in Appendix A.

Furthermore, there was an interest in a new server platform named **node.js** and Motorola was interested to know if this was a usable technique.

1.2 Motivation

Web servers must be able to handle multiple users at the same time. The traditional way is to use one thread per request. For example Apache uses the approach in its multi processing module *Worker MPM* (The Apache Software Foundation 2011).

Node.js is a young platform for building event-driven networking applications, e.g. web servers. The developers behind **node.js** proposes a different approach in comparison to traditional web server software. For example, the platform brings the JavaScript language into the server-side; JavaScript is normally run in a client-side web browser. JavaScript is normally executed inside a web browser, and the move to server-side has some properties worth investigating. The language itself has often been criticized for being a "toy language" doing bad things, but it has some "the good parts" that are worth trying out (Crockford 2008).

Server software based on the **node.js** platform uses asynchronous I/O whenever possible, and we would like to see how this paradigm affects software design and performance.

In short, **node.js** is a platform which is still evolving rapidly. It has got a lot of media attention in the software developer community in the last couple of years and we wanted to try it out and see if it lives up to the hype.

To evaluate **node.js** we built a system at Motorola to determine if it was possible/reasonable to build web services based on this technology.

1.3 Problem Formulation

The aim of this study is to investigate the usefulness of the web server platform **node.js**.

Since **node.js** applications are written in JavaScript, it is also important to describe and assess JavaScript's abilities and weaknesses in the context of a server-side language.

Node.js uses an event-driven, non-blocking I/O model (Joyent 2012). Therefore, we investigate the differences between this model and the more traditional model of multiple threads and blocking I/O to handle concurrency.

This leads to some more questions that will be answered in this paper:

- What are the pros and cons of asynchronous programming compared to synchronous programming in the context of event/threaded concurrency?
- What are the pros and cons of event-driven concurrency, compared to threaded concurrency? Does JavaScript solve any problems with event-driven concurrency?
- How does Wellington perform under different scenarios?
- How mature is the **node.js** platform and ecosystem?

1.3.1 Limitations

The STBs already had the functionality to fetch boot image via HTTP implemented in firmware. There was no need for us to design or implement the firmware boot protocol.

We have not evaluated the impact Wellington had on the testing process in this study, neither from a usability nor economic point of view. That is an entirely different thesis topic.

The system we built, Wellington, is not going to handle a massive amount of users simultaneously. Therefore, we have not tried to scale out the system to multiple computers. Part of the system is distributed by design, which decreases the risk of the server hardware being the bottle neck.

Due to the fact that we only implemented one solution (on **node.js**), no comparison with other server software platforms was made in this thesis. Thus, we didn't measure any other implementation.

The testing have only been focused on single core performance and the scaling to multicore systems have only been briefly touched upon in the thesis.

1.3.2 Related work

For the Python programming language there are at least two event-driven frameworks **Twisted** and **Tornado**. There are also similar non-blocking frameworks in other languages such as **eventmachine** in Ruby and **Lift** in Scala.

As for JavaScript, there are other alternatives for using it outside the browser. For example, **Narwhal** is one such project providing a standard library conforming to the **CommonJS** standard. CommonJS lists[1] similar projects that have not yet come as far as **node.js** and Narwhal.

There have been a lot of research concerning concurrency and how to handle multicore systems and scaling in computer science. Welsh, Gribble, Brewer & Culler (2000) present a design framework for highly concurrent systems combining threads and event-loops. von Behren, Condit & Brewer (2003) argue that cooperative threading is the best way to achieve high performance web services. In contrast, von Leitner (2003) argues that event-driven architecture is the best way to build a high performance web server.

[1] `http://www.commonjs.org/impl/`

2 Methodology

This section describes how the work was structured and the methods used.

2.1 Literature Study

A literature study was conducted during this thesis work. Scientific and research papers were studied, but since **node.js** is still a young technology there were few papers available on the subject. Therefore, online documentation and blog posts were also studied.

2.2 Experimental Software Engineering

We have used a method that is based on experimentation, tests and studies of documentation. Basili (1996) claims that software engineering is a *laboratory science* and that it depends on experiments to test and prove theories. By applying a technique in practice it is possible to understand how it works in context and learn from it.

An agile (Beck et al. 2001) approach to software development was used. The list of requirements and planned features was changed and improved over time in collaboration with our supervisors at Motorola Mobility.

2.2.1 Requirements Analysis

We discussed the initial requirements with the customers to be able to build a usable product.

During the initial phase we tried to get a grasp of the problem by talking to our supervisors and other employees. A short survey was performed with the future users of the system-to-be-built. We studied how the existing systems were used and what features the new system should have. Ideas and improvements were collected via informal interviews and meetings.

A list of requirements was developed. From those requirements a list of features and tasks was created and prioritized. The list was flexible, and during the implementation new items were created while others were removed in collaboration with the customers.

2.2.2 Implementation

The implementation of the product was performed by writing code. We used an iterative process and strove to have a working product at most times.

The testers at Motorola, that was the supposed target group, assisted with testing the system.

2.2.3 Testing and Evaluation

Basili (1996) claims that software engineering needs to follow an experimental paradigm like other fields e.g. physics. This thesis is an experimental evaluation of **node.js**, and how well it works as a platform.

To evaluate **node.js**, we built a system using its language, packages and web stack to form our own opinion about it.

A HTTP boot server was built using **node.js**. This boot-server is called Wellington and provides a RESTful API that can be used to configure and integrate the server. To simplify the use we also built a web-based interface, which is a browser-based JavaScript application.

The main back-end-server handles file uploading, image generation and downloading of large files. It also aggregates information from other repository servers via HTTP requests. Therefore, the technical requirements on the system were varied and covered a lot of areas that are of interest when building web services. Consequently, this makes the results hold for many different networking applications.

We used CoffeeScript which is a language that translates to JavaScript via a source-to-source compiler. CoffeeScript gives JavaScript a new syntax that helps to avoid a lot of JavaScript's bad parts and things that often lead to mistakes. (Ashkenas 2012)

Our approach separates the client side very much from the server with client-side-rendered templates and HTML5 local storage for caching of data. This is not the only way to do it in **node.js**, but makes it different from Ruby on Rails or PHP where templates most often are rendered at the server.

2.3 Method criticism

We have only implemented one version of the system due to time constraints. Consequently, we have not evaluated any other language or framework. Therefore, this is not to be regarded as a comparison of different platforms, but rather an evaluation of **node.js**.

The exploratory nature of the experimental method may have weaknesses. Without rigorous documentation of the procedure it is very hard to disprove a hypothesis that claims something to be possible. For example, there may be other possible solutions to the problem that were not considered in the development process. On the other hand, if the result confirms the hypothesis that something is possible that is still a solid result.

Node.js is a young project and little research has been done. Therefore, the documentation and research does not always fully cover all functionality.

2.4 Criticism of sources

Some of the sources used in this thesis are found on the Internet in the form of blog posts and answers on stackoverflow[1] and similar fora. Since **node.js** is such a new phenomenon few sources have been available during this process. Many renowned members of the community have been quoted from websites and they have also committed open source software of quality which make their opinions more trustworthy.

2.5 Disposition

Chapter 3 presents the theory which lay the ground for the result of this thesis. The topics covered in that chapter are JavaScript, **node.js**, multitasking and scaling and REST.

Some JavaScript language features and performance is presented in section 3.1 as a basis for the advantages and disadvantages of **node.js**.

The architecture of **node.js** and its advantages and disadvantages are covered in section 3.2. Section 3.3 presents the topics of multitasking and scaling. That section discusses different paradigms for multitasking, how these work and how they affect scaling of web services using them.

The result of this thesis is covered in chapter 4. The architecture of Wellington and the technical choices made during its development is covered. As are the problems that have occurred during the thesis and secure API design. Last in this chapter some performance measures are reported and presented.

[1] `http://www.stackoverflow.com/`

In the discussion we discuss the results and cover asynchronous programming, HTTP authentication, the maturity of **node.js** as a platform, performance, front end development and future work.

In chapter 6 some conclusions are drawn.

3
Theory

This chapter describes the theoretical base upon which the implementation work is based. It starts of with the language JavaScript, its history and some important parts of the construction of the language. Section 3.2 presents paradigms and models facilitated by **node.js**. Then concurrency in computing is discussed. Event-driven programming is compared to thread-based programming. Lastly REST (section 3.4) and Promises (section 3.3.5) is described for future reference.

3.1 JavaScript

In this section the language JavaScript is discussed, it is highly integrated in **node.js** which is why it is of interest for this thesis.

3.1.1 History of JavaScript

JavaScript was constructed by Brendan Eich at Netscape in 1995. It was conceived as the scripting language of Netscape's browser *Navigator*. Netscape Navigator already had support for Java-applets, but as this was targeted at professional programmers, Netscape wanted a lighter scripting language more like Microsoft's Visual Basic. (Severance 2012)

In 1996 Netscape submitted JavaScript to the Ecma International which resulted in the standardized ECMAScript in 1997 (Andreessen 1998).

JavaScript have since then become *the* client-side scripting language of the web and is one of the most wide spread languages, all modern browsers come with a JavaScript-engine and almost all PCs have at least one browser installed.

JavaScript was designed with the early web browsers in mind for adding interactive elements to web pages. Common tasks included adding dynamics and interaction to web pages represented by a HTML tree. A web page is represented as a Document Object Model Interface (DOM) and is not part of the JavaScript specification. The DOM is defined by W3C, but different browsers' implementations differ. These are the reasons why JavaScript has no standard library for things common in most other languages, such as file handling, network communication or database connections.

JavaScript's syntax inspired by C and Java, but the inner workings of the language are inspired by Scheme (Eich 2008). It features a prototype-based inheritance model and functions are first class objects. It also features lambdas i.e. anonymous functions.

The different types available in JavaScript are Number, String, Boolean, Object, Array, Null and Undefined. An Object is a collection of properties, where a property can be either a *data property* or an *accessor property*. Objects may be created with the object initializer syntax (figure 3.1) and is used as flexible data structures. (Ecma International 2011).

```
var my_object = {
  hello: "world",
  x: 3,
  square: function (a) {
    return a*a;
  }
};
my_object.hello // "world"
my_object.x // 3
my_object.square(7); // 49
```

Listing 3.1: *Object initializer*

JavaScript have often been accused for being a bad language and it has a lot of bad parts. However there are good parts too. It is an often misunderstood language (Crockford 2001), but with the rise of the web as a platform for an ever increasing amount of things. JavaScript often has to be used when building web systems. JavaScript has become the "Language of the web" (Crockford 2008).

3.1.2 The evolution of the JavaScript ecosystem

In recent years, with the rise of Rich Internet Applications (RIA), JavaScript have been used in increasingly more complex and sophisticated products. This evolution have lead to the creation of many JavaScript frameworks (e.g. jQuery, prototype, ExtJS, Backbone). This has lead to a lot of code being available to developers, but since JavaScript has no module system, many libraries and frameworks have been hard to use together. The CommonJS module specification aims to help solve this problem (*Modules/1.1 - CommonJS Spec Wiki* 2012).

JavaScript has a very limited standard library. However, browsers supply many API:s to allow manipulation of the DOM and other browser specific features. Using JavaScript as a general-purpose programming language, outside the browser, requires various additions to the standard library. For example, file system manipulation, sockets and standard interface to databases are needed.

CommonJS is an initiative to bring JavaScript up to par with other scripting languages like Python and Ruby. In order to achieve this, work has begun to specify a more extensive standard API, which have been a big step in order to move JavaScript to the server-side (Kowal 2009). CommonJS started with the post *What Server Side JavaScript needs* by Dangoor (2009). At the time of writing, CommonJS has five current specifications and 32 proposals in development (Dangoor et al. 2012).

3.1.3 Closures

Functions may exist inside of functions and be returned as first class objects to some variable outside the function it was first constructed in. When this happens a new *closure* is created. A closure is a data structure containing a function and a referencing environment for the non-local variables for that function. (Sussman & Steele Jr. 1975)

Listing 3.2 shows the function `create_adder` which creates an inner (anonymous) function and returns it. The inner function can be invoked later, when `create_adder` has returned, and it will still have access to `x`.

```
function create_adder(x) {
  return function (y) {
    return x + y;
  }
}

add_two = create_adder(2);
add_four = create_adder(4);

add_two(7); // 9
add_four(10); // 14
```

***Listing 3.2:** Closure*

Closures come at a cost; closures is the most common source of memory leaks. It is also slower to create a closure than an inner function without a closure. (Baker & Arvidsson 2012)

3.2 Node.js

Node.js is a platform for building event-driven networking programs, e.g. web servers. It provides the developer with a JavaScript runtime and libraries to write applications. The JavaScript-runtime is V8; the same engine that is used by Google's web browser Chrome. V8 uses dynamic machine code generation to achieve high performance (Google 2012).

Node.js is more of a standard library to JavaScript than a web framework and should be compared to PHP[1] or Ruby[2] rather than CakePHP[3] or RubyOnRails[4]. It exposes an event-loop which executes as long as there is some event to listen to for example socket connections or non-blocking I/O. Listing 3.3 shows an example web server that responds with "Hello World" to every HTTP-request.

```
var http = require("http");
var server = http.createServer(function (req, res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.end("Hello World\n");
});
server.listen(1337, "127.0.0.1");
console.log("Server running at http://127.0.0.1:1337/");
```

Listing 3.3: *Hello World web server*

One of the proposed advantages with **node.js** is that it should be easy to write scalable network applications (Joyent Inc. 2012). This is accomplished by using one event-loop. To avoid blocking the main loop all API-calls that involve I/O are asynchronous, i.e. non-blocking. Thus, instead of waiting for the function to return, the event-loop can execute the next event in the queue, and when the API-call is finished it calls a callback function specified when the call was made. Later in this paper we discuss the pros and cons with event-driven compared to threaded systems with respect to scaling.

3.2.1 Architecture

As can be seen in figure 3.1, **node.js** is composed of a number of components. V8 is the JavaScript engine. **libeio** handles an internal thread pool which is used to make synchronous POSIX-calls asynchronous to the event loop. Asynchronous file system is not used, instead blocking I/O is performed in its own thread, so it is non-blocking to the event loop in **node.js**. **libev** is the event loop. Node bindings are some thin C++ bindings that expose the API of the underlying components to JavaScript.

Node.js's standard library exposes operating system feaures such as handling file system, sockets, processes and timers. The standard library also has functionality for events, buffers and simple DNS and HTTP requests.

[1]http://php.net
[2]http://ruby-lang.org
[3]http://cakephp.org/
[4]http://rubyonrails.org/

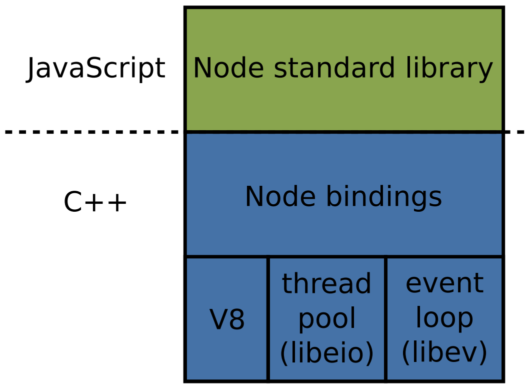

***Figure 3.1:** Node architecture*

Node.js uses techniques like `kqueue`, `select`, `epoll` or `/dev/poll` to get notifications from the operating system when new connections are incoming and then dispatch new events to the programmer to handle with handlers. The technique used depends on the operating system.

Node.js aims to give the programmers the tools needed to solves the 10,000 concurrent connections problem outlined in Kegel (2011). **Node.js** is single-threaded by default. The way to scale out is to spawn new threads, via the cluster module. Each new thread has its own event loop and is preferably run on separate cores.

3.2.2 Why JavaScript for event-driven web servers?

The JavaScript-engines that run in browsers are single-threaded processes and use an event loops to handle concurrency. Because of this, binding event handlers and triggering events are easy things to do in the language. Developers who are used to work with client-side JavaScript will find the concepts of **node.js** to be familiar.

Event loop based servers scale very well with many concurrent connections as the throughput is the same even under high load. The bottleneck is the CPU, but even with more connections per second than the system can handle the throughput is the same (as the peak) but the delay of requests/connections gets higher (Welsh et al. 2000).

Node.js implements the CommonJS specification *Modules/1.1*.

3.2.3 Advantages

JavaScript

JSON is the native representation of objects in JavaScript and one of the most common formats when sending data via Ajax.

A programmer working on both front and back end will not have to switch between two languages. Switching between different tasks or languages is considered a high cost for a programmer and may decrease productivity (Sparsky 2001).

Code reuse with the same code running both on the server-side and on the client side, in the browser is possible when the language on both sides is the same.

JavaScript comes with some language features that makes it suitable to write event-driven applications. The first is closures which automates the saving of state on the stack. This is a problem in other languages that is a non-issue in JavaScript. JavaScript is also fitted for making callbacks because it has anonymous functions.

Node.js

One of the advantages of **node.js** over many other event-driven frameworks is that **node.js** has a built-in event loop. All available modules and libraries in the ecosystem use it. This makes the "gluing" together multiple parts to one asynchronous program a lot easier. (Elhage 2012)

When using **node.js** the event-driven paradigm is the natural way to do things. As long as the programmer does not do anything blocking the performance and scaling will come from the platform. **Node.js** is less complicated to program than following the same paradigm in for example C.

3.2.4 Disadvantages

JavaScript

JavaScript requires semicolons at the end of statements but has a feature for inserting missing semicolons in the source. This was intended to make it easier to write JavaScript but unfortunately often makes the code behave differently than intended, especially if the programmer is not aware of this feature.

JavaScript's comparison operators come in two flavors; both strict and type-converting, where the strict operator return true only if the two compared objects are of the same type (Network 2012). The problem is that the operators == and != does not behave in JavaScript as in many other languages.

Another disadvantage of JavaScript is the lack of native modules. This makes it harder to write large projects in a structured manner. However, there are implementations of modules by CommonJS.

Node.js

Asynchronous event-driven programming is a new concept to many developers and take time to get used to and get fully productive with. The paradigm solves some problems from the pre-emptive threaded paradigm but instead has its own problems.

In **node.js** there is no loose coupling between the web server and the web application. Loosely coupled architecture has proved itself to be maintainable. (Dziuba 2011)

As with many areas of **node.js** development, deployment is an area where there are many alternatives, but none of them are well tested. This can be a factor for consideration when choosing **node.js** as development framework. The fact that it is new and untested in large environments can also be a risk factor for businesses.

3.3 Multitasking and scaling

This section presents how multitasking can be achieved in different ways and the pros and cons with the discussed paradigms. Furthermore, how the multitasking paradigms affect the ability to scale out over many cores or machines is discussed.

Multicore processors have existed for some time, but there is still a need for a single core to be able to execute more than one process at a time. This is most often accomplished via time-division multiplexing. A scheduler decides which process should run and for how long. (Ousterhout 1995)

3.3.1 Event-loop vs Threads

When writing a program in a single process, there are a number of ways to handle concurrency within this process. One way is by spawning lightweight child processes called threads, which share the same address space as the main thread of execution. Each thread allocates its own stack, registers and program counter. Threading needs a scheduler that can be either pre-emptive or cooperative. Another way is to create an event-loop with event handlers. When an event is fired the associated handler is queued up for execution. Each event handler is allowed to finish and return control of the CPU when finished.

3.3.2 Pre-emptive concurrency

Pre-emptive concurrency is scheduled time divided multitasking where a scheduler decide which process or thread that is to run and for how long before it gets interrupted.

Threading is a widely adopted technique for multitasking and the implementation of threads differ between operating systems.

A thread can be interrupted at any time, and at that time all CPU registers must be saved. The registers must then be restored when resuming the thread. Because the thread can be interrupted at any time, one must program with care when sharing resources between threads. However, as long as a thread only use local resources this is not a problem.

Code that may be accessed from different threads has to be *thread safe*. There are several theories on how to write thread safe code. This involves, for example, locks, transations, and semaphores. However, it introduces a number of new problems, e.g. deadlocks and starvation. (Ousterhout 1995)

In addition, the context switching of threads infer some overhead costs. Each thread has its own stack that has to be kept in memory so that the thread can continue executing from the exact same point. With a massive amount of threads the overhead of scheduling might increase drastically (von Leitner 2003).

The use of threads often leads to an easier flow pattern in the program and this makes it easier to maintain and develop. However, there is a cost attached to switching process in that the scheduler has to store away the state of each paused process/thread which is cheaper in **node.js** with closures and events.

3.3.3 Cooperative concurrency

Cooperative concurrency is based on the premise that all code cooperates and voluntarily returns control to the system when a task is done. This means that task interleaving can only happen at certain points. Cooperative task management is faster and easier in highly concurrent systems than pre-emptive concurrency handling. This applies whether they are event based or threaded (von Behren et al. 2003).

The considerations that have to be made, when developing a cooperative concurrent system, concern the responsiveness of such a system. Since any task can block the main-loop for however long time it desires. If a responsive system is wanted all tasks have to be kept short. If the tasks are too small the switching may cause too much overhead. On the other hand if the interleaving are too coarse grained the system will have less overhead but also become less responsive.

CPU intensive tasks are blocking, and if responsiveness is a goal then a long task can be divided into smaller tasks. The division allows other tasks to execute interleaved with the long task to improve responsiveness.(Levis 2006)

Normally I/O operations are blocking, meaning that the calling thread will be blocked until the I/O operation completes. During this time nothing else in the thread can be executed. The solution is to use non-blocking I/O, i.e. the calling thread will not be blocked. It can for example queue up file system reads but continue executing. When the read operation completes the thread will be notified. There are different ways of signaling in Linux, other unices and Windows, e.g. `select()`, `epoll()` and IO Completion Ports (IOCP) [5].

A disadvantage of cooperative concurrency is that the technique requires everything to be "friendly" and cooperative. Misbehaving code in a small part can block the whole system, e.g. a infinite loop that never returns control. This makes the whole paradigm an all-or-nothing approach.

Cooperative threads

Normally threads are wrongly assumed to be subject to pre-emptive scheduling. In fact, that is not really a property of threads, but rather of the implementation. According to von Behren et al. (2003), cooperative threading can achieve the same (or higher) level of performance as event based systems. He also claims that the thread model is easier for programmers to use and can be supported by compiler optimizations.

Events

Event-based systems have one event loop running in one thread. The event loop normally has a queue of event handlers that it executes in order. The overhead

[5] `http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx`

when switching from one event handler to the next one is much lower than with pre-emptied threads (especially when compared to kernel threads). (von Behren et al. 2003)

Claims have been made that the flow of a event-driven program tends to be very complex. Event based programs can have very flexible and complex patterns while threads are more sequential. von Behren et al. (2003), on the other hand, argue that almost all flows can be described as *call/return*, *parallel calls*, and *pipelines*, and that all other are too complicated to use.

When threads switch jobs on the processor the running task/functions stack is automatically saved so that it can be restored when the task is resumed. According to von Behren et al. (2003), event based systems have to manage the stack manually i.e. all variables that will be needed by an event handler have to be manually made available in that scope. This is particularly hard when the handler will run at some unknown time later. Adya, Howell, Theimer, Bolosky & Douceur (2002) briefly mentions that languages with closures, like JavaScript, come with a solution to manual stack management. By creating a closure with the event handler, the variables will be bound to the referencing environment.

3.3.4 Event-driven programming

Event driven programming is a paradigm often used for interactive application interfaces. User interaction creates events and the main loop executes the relevant event handlers.

Listing 3.4 shows an example of event based programming in Java Swing. The method `buttonHandler` is executed by the event loop and is meant to return fast, in order to avoid blocking the main (GUI) thread. However `buttonHandler` needs to do a blocking call, `lookup`. Therefore, it spawns a background task in a new thread to handle the lookup. When the lookup is complete a GUI component is to be updated. In Java Swing, GUI operations must be run from the main thread so the background task must create an event handler (anonymous inner class extending from `Runnable`) and post it to the main thread via `invokeLater`.

```java
void buttonHandler() {
    // This is executing in the Swing UI thread.
    // We can access UI widgets here to get query parameters.
    final int parameter = getField();

    new Thread(new Runnable() {
        public void run() {
            // This code runs in a separate thread.
            // We can do things like hit a database or access
            // a blocking resource like the network to get data.
            final int result = lookup(parameter);
            final Runnable handler = new Runnable() {
                public void run() {
                    // This code runs in the UI thread and can use
                    // the fetched data to fill in UI widgets.
                    setField(result);
                }
            };

            javax.swing.SwingUtilities.invokeLater(handler);
        }
    }).start();
}
```

Listing 3.4: *Example callback in Java, source Wikipedia*

In JavaScript the syntax becomes less verbose since normal functions can be used as event handlers, without creating anonymous inner classes. It is because functions are first class objects and can easily be passed as parameters. The event handlers in **node.js** are ordinary functions used as callbacks.

The **node.js** API methods often take callback functions that eventually are executed when the non-blocking operation completes (such as file system/socket read/write). The event loop in **node.js** receives the completion signals later and executes the callback.

Listing 3.5 shows how a blocking file read is performed. Note that the thread waits until the file is read and `do-something(data)` returns before calling `foo`. In contrast to listing 3.6 where `foo` is called immediately.

```javascript
var data = fs.readFileSync("/etc/passwd"); // blocking operation
do_something(data);
foo(); // Foo will be called when do_something returns
```

Listing 3.5: *Synchronous file read*

```javascript
fs.readFile("/etc/passwd", function(err, data) {
    do_something(data);
});
foo(); // Foo will be called before do_something
```

Listing 3.6: *Asynchronous file read*

3.3.5 Asynchronous libraries

Ousterhout (1995) argues that threads are much harder to program than events. von Behren et al. (2003) on the other hand argue that events are too hard for complex programs, and means that threads should to be used. Regarding whether threads or events are easier, there seem to be no consensus among scientists and software engineers. However, most agree on that concurrency is hard (Ousterhout 1995).

To simplify the readability and semantics of asynchronous code there are several techniques and libraries to help. This is true also for the **node.js** ecosystem, e.g. node-fibers, promised-io, futures etc.

Fibers are a low level concept which is essentially the same thing as coroutines, or cooperative threads of execution. **node-fibers** is a library that adds support for fibers to node. A fiber has its own stack and it can *yield* in order to give control back to the event loop and save its stack at the same time. The execution can be resumed later by making the fiber's stack active again. Herman (2011) argues that "Coroutines are almost as pre-emptive as threads" in the perspective that a function or module may call `yield` which is not explicitly shown to the programmer. This has the implication that the main loop may execute before the original function execution is resumed. This is similar to pre-emptive concurrency and breaks the run-to-completion assumption that usually is true in JavaScript (Mozilla Developer Network 2012).

There are many higher level program flow abstractions that try to make asynchronous more readable (*nodejs wiki - Control flow / Async goodies* 2012). One such design pattern is called a promise (sometimes also referred to as future, deferred).

A **promise** represents a value that will be resolved eventually and is an object with three possible states: *unfulfilled*, *fulfilled* and *failed*. There are only two allowed state transitions:

- *unfulfilled* $\rightarrow$ *fulfilled*
- *unfulfilled* $\rightarrow$ *failed*

According to the proposal *Promises/A* by CommonJS, it is possible to attach a handler that will be executed when the promise is fulfilled. If the promise already is fulfilled when the handler is attached the handler will be executed at once. It is also possible to attach an error handler, which is called when the promise enters the failed state. (*Promises/A - CommonJS Spec Wiki* 2012)

As can be seen in listing 3.7, the use of a promise library can help reducing the nesting depth and the sequential semantics are more visible. However, it requires the involved functions to return promise objects. jQuery has an implementation of promise semantics and many of its functions return promises (The jQuery Foundation 2012).

```javascript
// Original "Pyramid of doom" of callbacks
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Do something with value4
            });
        });
    });
});

// Modified code using a promise library
step1()
.then(step2)
.then(step3)
.then(step4)
.then(function (value4) {
    // Do something with value4
}, function (error) {
    // Handle any error from step1 through step4
})
```

***Listing 3.7:** Use of promises*

3.3.6 Web servers

This section discusses how the use of different multitasking paradigms affect the scalability of a program.

A correctly implemented, pre-emptively threaded application has the advantage that each thread can run on different cores of a multicore CPU without any changes to the program. This, however, comes with the overhead attached to pre-emptive threads which may degrade performance, if a massive amount of threads are used.

Apache is a common web server on the Internet and widely used as a default web server. Apache has different MPMs (multi-processing module); two of which are pre-forking and worker. *Pre-fork* as the name implies forks a pool of threads before accepting connections and then do not accept more connections than there are threads in the pool. The other module, *Worker*, consists of one control process, that spawns new threads which in turn spawn threads that serve connections. This is the traditional threaded model for web servers. (The Apache Software Foundation 2011)

von Leitner (2003) and Kegel (2011) propose the use of event-driven web servers to handle a massive amount of concurrent connections (the 10,000 clients problem).

3.4 REST

Conforming to the Representational State Transfer (REST) constraints are often called being RESTful. REST is a way to design interfaces between clients and servers e.g. web services. (Fielding 2000)

The fundamental abstraction of information is a **resource**. A resource can be a collection of other resources, temporal information or any other information that can be targeted by a hypertext reference. These resources are mapped to hypertext references. Actions can be performed on resources via these references. All REST interaction is stateless similar to HTTP. (Fielding 2000)

According to Fielding (2000), there are three fundamental options of presenting the data (resource) to the user

1. Render the data where it is located and send a fixed-format image and send to the recipient (traditional client-server-style)
2. Encapsulate the date with a rendering engine and send to the recipient
3. Send the raw data (together with metadata) to the recipient and let the recipient render with some rendering engine

REST is not a specification, nor does it specify transport. For example, it is possible to use REST over HTTP or any other transport. It is suitable to use the HTTP request methods, also known as verbs, GET, POST, PUT and DELETE to manipulate the resources.

3.5 HTTP authentication

REST does not specify any means of security; hence it is up to the specific system to handle security implementation. When using HTTP as transport protocol it would be possible to make use of the authentication methods available.

When using REST over HTTP it would be possible to use one of the authentication schemes defined in RFC 2617 (Franks et al. 1999). HTTP Basic is insecure in itself and requires a secure channel (such as SSL encryption). The basic scheme essentially transmits the username and password in plain text to the server. It is fairly easy to sniff the combination to steal a users identity. This method exposes a users password, and if the user chooses their passwords this may lead to compromising other services where the same password is reused. Users often reuse the same password several times (Florencio & Herley 2007).

HTTP digest is considered the secure alternative to HTTP basic. But it does still not provide strong authentication when compared with public key methods. The password is never transmitted directly, but instead a digest message is sent. Digest feature neither confidentiality nor full integrity protection. The quality of protection (qop) specifies what data should be used as input to the signature. (Franks et al. 1999)

A digest authentication is initialized when a client requests a URL that requires authentication. The server responds with 401 forbidden and the header WWW-Authenticate. This header contains realm and a random nonce. Optionally it may obtain an opaque field. The first version of digest authentication did not specify qop and the digest was created from the values username, password, realm, HTTP-method, request-uri and nonce. The technique is based on challenge-response and calculating hash values (signatures) of the HTTP request. (Franks et al. 1999)

3.5.1 Cross site requests

One security issue in web applications has been so called cross-site request forgeries (CSRF or XSRF), meaning that unsafe JavaScript-code could be included on web pages and execute malicious code. A policy was invented by browser vendors to mitigate this security issue. This policy is called *same-origin policy*. In short a web page can only do XMLHttpRequests (XHR) to the same origin/domain that the page came from. If a script would try to issue an XHR-request to a server that is not the origin of the script the browser should be block it.

Since developers want to have the possibility to talk to third-party servers something had to be done , while still preserving the security by default. *Cross-origin resource sharing* (CORS) is a way to enable sharing of resources from a server to web pages from other domains.

To facilitate the same origin policy the browser makes a *preflight request* to the requested URL, but with the OPTIONS method. The server responds with headers that specify if the actual request would be allowed. With CORS a server can specify which resources should be allowed by clients to access. The CORS preflight response may contains rules describing which HTTP-methods and headers are allowed.

4 Results

This chapter presents the solution and implementation of Wellington. How the system perform under different conditions and what design decisions that were made will also be covered.

4.1 Wellington

This section describe the construction and implementation of the final product of this thesis work; the HTTP boot server Wellington.

4.1.1 Architecture

The implemented system consists of four loosely coupled applications, which can be seen in figure 4.1. These are:

- Main server
- Repository server(s)
- Web interface (WebUI)
- Command line interface (CLI)

The **main server** is center piece of the whole system. It contains a database and controls the configurations of all connected STBs. This server exposes a RESTful API that should be accessible to clients, both using the WebUI or CLI, and by STBs. All components communicate over HTTP.

The fact that all actions of the main server were to be available both via WebUI and CLI solidified the decision that the front and back end were to be completely separated.

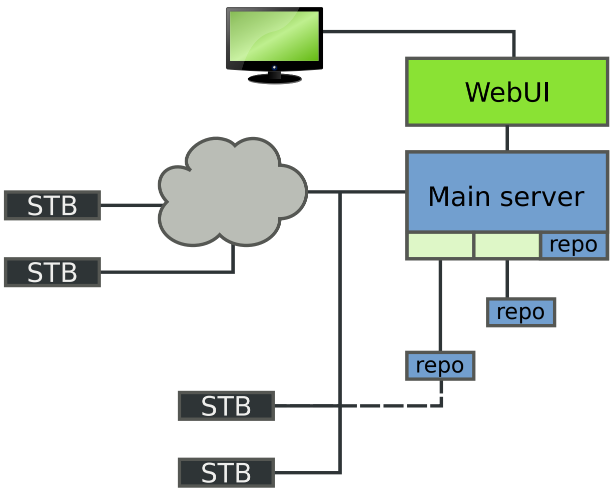

Figure 4.1: *Architecture of Wellington*

This meant that the web interface had to be a *Rich Internet Application* (RIA) with client side state and functionality in JavaScript. All communication with the main back end server were to use asynchronous requests i.e. *XmlHttpRequest*. This made it easier to host the web interface client as it can be served using any web server capable of serving static files (e.g. Apache, LightHTTP, IIS or nginx).

All software was written in CoffeeScript and compiled into JavaScript which executes in **node.js**. The express framework[1] was used to structure the code and to design the API. The server contains the functionality to handle users, STBs and boot images. It also has the responsibility to communicate with other repositories and generate custom splash images.

A **repository** allows a user to host and serve boot images from a local folder without moving the files to the main server. The repository component is a small web server that can serve static files to a booting STB and also allows the main server to list all available files. When a repository server is started it announces its existence to the main back end server which then asks for a list of all available boot images.

This architecture was chosen because there are a lot of teams of both testers and developers, the potential users of the system, which all have their own servers and workstations where they keep boot images. It also helps to lower the load on the main server as can be seen in figure 4.2. The STBs boot directly from the repositories without going through the main server when configured to boot an image that is located on a repository server.

[1] `http://expressjs.com/`

The STBs already had the ability to boot over HTTP and thus there were no immediate need to change anything in their firmware. This meant that the server had to accept the kind of requests the STBs send out.

The deployment of a product is also of interest when choosing platform. There were not very much trouble getting a **node.js** application to run as a system daemon. There are several libraries that accomplish this.

4.1.2 Technical choices and frameworks

In this section the most notable frameworks and their use is described.

Front end

The Web Interface was built using an MVC-framework called **Spine**[2]. This was chosen over backbone.js[3] and knockout.js[4] because it seemed simple to use and get started with. Also contributing to the choice was the fact that Spine was written in CoffeeScript. It was later revealed that the model for persisting objects and state to the server that Spine imposes was not entirely suited for this application. The problem with the model for server communication was that Spine assumes that all state is kept and handled in the client. Therefore, Spine assumes that all requests that are sent to the server are going to be accepted. The back end is just used to persist state from the client and if the state is approved by the client the server should too. This model was not suitable to this system since requests to the server could come from either the CLI or the WebGUI making the back end server responsible for the state. The model class of Spine had to be extended and modified to better fit. This helped to reduce the amount of resources that were preloaded at startup.

Spine includes **jQuery**[5] and the template system **Eco**[6]. jQuery also contains an implementation of CommonJS Promises/A, which are used heavily throughout the implementation.

Since graphical user interface design was not the focus of this thesis the CSS and JavaScript framework **Bootstrap**[7] was used as a baseline for design. It also contributed to minify the time spent on writing stylesheets and layout handling. Bootstrap's JavaScript components are built as jQuery extensions and depend on jQuery.

jQuery was also used to handle some of the DOM-manipulation and preprocessing of AJAX-requests on the front end as it is a versatile helper library.

[2]http://spinejs.com/
[3]http://documentcloud.github.com/backbone/
[4]http://knockoutjs.com/
[5]http://jquery.com/
[6]https://github.com/sstephenson/eco/
[7]http://twitter.github.com/bootstrap/

Less[8] is a preprocessor/superset of CSS which introduce variables and mixins in CSS. Bootstrap uses Less to make it easy to tweak the look and feel of the framework. Less was used in the rest of the project as well for its easier nesting syntax (mostly) and to get variable support in CSS.

Back end

In this section follow descriptions of the most influential frameworks used on the back end, the web service, and notes about their use.

The API of **node.js** is fairly low-level. Therefore it is convenient to use some framework on top of it with some more abstraction such as routing. Express is a web framework that builds on top of **Connect**[9]. Connect uses middleware that handles and parses request before they get to the rest of the application. This pattern is used in the back end with middlewares for handling uploading and multi-part forms, authentication and convenience methods.

Promised I/O[10] contains implementations of CommonJS Promises/A for use server-side. See section 3.3.5.

PersistenceJS[11] was used as database wrapper. It is an abstraction layer on top of either SQLite3 or MySQL. It can also be used in the browser interfacing LocalStorage. PersistenceJS have a module for database migrations, which was intended for use in the browser but could be patched to work on the server-side as well.

4.2 Problems during development

Over the course of the implementation problems have occurred. Some originate in either the event-driven paradigm or in CoffeeScript syntax. In this chapter a selection of these are described and, where appropriate, solutions to the problems are presented.

There is one problem that has occurred several times when refactoring or extending a snippet of code. The problem occur when a non-asynchronous function must call an asynchronous function. Let us call that function `foo()` which used to return a number. Now it has to change to read that number from disc asynchronously instead. This has the implication that the function must be transformed and can no longer *return* a value.

When using synchronous functions the function is called and the code is resumed when the function is done.

The problem is that if the function, `foo()`, is changed to be asynchronous the flow of the program is altered and the caller also has to be asynchronous. So every call to `foo()` must be changed. If `foo()` is called deep down in a chain

[8] `http://lesscss.org/`
[9] `http://www.senchalabs.org/connect/`
[10] `https://github.com/kriszyp/promised-io`
[11] `http://persistencejs.org/`

of function calls, the whole chain must be transformed. This makes it harder to change existing code and is also a process that may introduce errors.

```
    function foo() {
        return 3;
    }

    function bar() {
        var x = foo();
        console.log("x is " + x);
        return x + 7;
    }
```

Listing 4.1: *Version 1, synchronous*

```
    function foo(callback) {
        fs.readFile(function (err, contents) {
            callback(contents);
        });
    }

    function bar(callback) {
        foo(function (x) {
            console.log("x is " + x);
            callback(x + 7);
        });
    }
```

Listing 4.2: *Version 2, asynchronous*

The use of promises can help with the problem mentioned above but they have to be applied in places where they are not yet needed but where things may change in the future.

One example of a hard to spot problem with asynchronous calls is when they are situated inside loops. This is illustrated in Listing 4.3. The output will be 4, 4, 4 not 1, 2, 3 because the variable `i` is captured in a closure and the callback functions will run after the loop has returned, when i = 4. However, the files `test1`, `test2` and `test3` will be read as expected. The for-loop will run to completion before the callback functions be able to execute. The order in which the read operations will be done is not certain in this case which is different from synchronous I/O where the order is preserved.

```
fs = require("fs");

for (var i = 1; i < 4; i++) {
  fs.readFile("test"+i, function(err, data) {
    console.log(i);
  });
}
```

Listing 4.3: *Asynchronous problem*

```
fs = require("fs");

for (var i = 1; i < 4; i++) {
  (function(_i) {
    fs.readFile("test"+i, function(err, data) {
      console.log(_i);
    });
  })(i);
}
```

***Listing 4.4:** Asynchronous solution*

Listing 4.4 shows a solution to the problem in Listing 4.3. However, the order of the output in Listing 4.4 depends on which `fileread` operation is completed first. The solution creates an extra closure which holds the correct value in the variable `_i`.

It is very important when implementing an asynchronous function to call the callback function on each endpoint. This is similar to returning from an non-asynchronous function. Forgetting to call a callback often leads to software that just stops working silently, which is hard to debug.

CoffeeScript was used throughout this thesis work to make it easier to write JavaScript. However there are some pitfalls. Among which some are highlighted here. One of the biggest advantages of CoffeeScript is that indentation is used instead of curly braces. This makes the code easy to read, but also makes it very hard to spot the difference between working code and non-working code in some cases.

Listing 4.5 shows an iteration over an array of repositories. When the first match is found, it is meant to add the found repository and break the iteration by doing an early return. However, the example does not behave as intended. The return statement has been wrongly placed in a callback function, which isn't executed until the for-loop has run to completion. This has the consequence that several matches may be found and added, but only the first match was supposed to happen. Listing 4.6 shows how the code is supposed to be indented, to work as intended.

```
  for repo in repos
    if repo.url.match("^file://")
      user.repositories.add(repo)
      session.flush ->
        session.close()
        return
```

***Listing 4.5:** CoffeeScript problems*

```
for repo in repos
  if repo.url.match("^file://")
    user.repositories.add(repo)
    session.flush ->
      session.close()
    return
```

Listing 4.6: *CoffeeScript problems*

4.3 Secure API design

If the service is exposed on a public network or to the Internet it is also exposed to a lot of threats. The system needs to prevent unauthorized users from accessing the system. Some of the threats are replay attacks, identity theft and also information theft or altering.

REST does not specify any mechanisms for security, authentication or encryption. HTTP authentication was not used in Wellington. HTTP Basic is not considered secure and HTTP Digest requires a challenge response algorithm which is not implemented by web browsers XmlHttpRequest API. Wellington's authentication mechanism is inspired by OAuth without implementing the whole standard. In short, each request is signed with a SHA1-HMAC checksum.

4.4 Performance

JavaScript has been criticized for its bad performance, which was true back in 2005 when AJAX was introduced. However, the development of JavaScript engines since then has improved the performance substantially and modern browsers use sophisticated JavaScript engines which are much faster (Smedberg 2010).

Some of the slowness that JavaScript in the browser is accused of is not actually the JavaScript-engine performing slow, but rather the object models of the browser, e.g. browser elements like window and the DOM are to blame (Resig 2008). These APIs are not present in **node.js**.

4.4.1 Long running tasks

Node.js and its non-blocking I/O model is intended for data intensive work. However, blocking CPU intensive tasks can be a problem because the CPU is the limiting factor. The responsiveness of the web server decreases drastically when bound by a long running task; it cannot respond to new requests. This is because the event loop will not get control back until the task is done, hence it cannot handle other events. A method to counter this behavior is to modify the long running task to be split into several smaller tasks and thereby letting the event loop get control between those parts.

The authentication mechanism was affected by this limitation. The authentication signature is computed in a blocking function, and if the request body is very large this will block the event loop. This was particular noticeable during file upload. The normal use case involved uploading a boot image of approximately 20–30 MB, which would require several seconds of signature computations and was assessed unacceptable. Therefore the content of the file uploaded was not used in the signature.

Security implications: By removing the request body for file uploads, there is no integrity check of uploaded files. This means that an attacker may intercept and change the file content that gets uploaded to the server. This could be classified as a major security flaw, but according to our supervisors every uploaded file (boot image) is signed in itself. Even if an attacker may upload a malicious boot image to the server, an STB would not be able to boot it since the signature of the boot image is invalid. (Unless the attacker has the ability to correctly build and sign boot images, but that is a problem on another level.)

Another threat that has not been mitigated is replay attacks where a malicious user could record the message another user sends and then resend it to the server which will accept it.

4.4.2 Massive connection concurrency

Node.js allows massive amounts of concurrent open connections at any given time. This is good for handling spikes in incoming connections but is not a solution if the number of incoming connections are constantly more than the server can handle.

During development SQLite was used as Database Engine. When starting testing it became clear that this was not suitable for deployment since SQLite can only write sequentially and thus the database locks when concurrent writes occur. Therefore SQLite was switched out for MySQL which can handle concurrent connections much better.

However when testing multiple concurrent connections we found that MySQL limits the number of concurrent connections. Similarly, the number of file descriptors that are allowed to be open at the same time on Unix systems also impose limitations.

4.4.3 Testing

To measure the performance of different use cases the program Apache JMeter 2.7[12] was used. The component under test was the main back end server.

JMeter works by simulating multiple users making multiple requests to the server. Every user is run in a separate thread. To simulate normal conditions JMeter allows a “ramp-up” time to be specified. The ramp-up is the time it takes

[12]http://jmeter.apache.org/

to start all users/threads. Every user is specified to make a given number of requests. In the following tests different configurations were used.

Two identical computers were used with Intel Celeron G530 @ 2.4 GHz processors and 4 GB ram, one running the Wellington server and the other running JMeter.

Manifest, splash- and boot images

This test was conducted to investigate how the behavior of the server was affected when STBs connect to download a new boot image.

When an STB is booting it makes three HTTP requests to the main server. The first requests fetches a manifest, which is an XML file. The manifest contains URLs where the STBs can find the splash and boot image. The boot image is located on either the main server or in any of the repositories, but the splash image is always located on the main server.

We measured how manifest and splash generation was affected when downloading boot images from the same server. This can be seen in figures 4.2, 4.3 and 4.4. For the following tests 10 users were simulated.

The first test **(1a (Manifest + Splash))** assumed all boot images were from repositories, and therefore only the manifest and the splash image were downloaded from the main server. This was performed to get baseline metrics to compare consecutive tests against. 100 iterations per user were performed, totaling 2000 requests.

In the next phase **(1b (Manifest + Splash + 10% Boot Image))** all boot image requests were directed to the main server. Every manifest downloaded was followed by one splash download and one boot image download. For this test fewer requests (50) were made because the test would otherwise take too long to complete.

The third test **(1c (Manifest + Splash + Boot Image))** was chosen to be more representative of a real world use case. Since the storage of boot images is distributed we foresee that a lot of requests for boot images will be directed to repositories, which will result in fewer requests to the main back end. The boot images are quite large and the downloading of them will constitute a substantial part of the bandwidth. In this test we wanted to simulate that 90% of the boot images were downloaded from other servers, thus leaving only 10% to be downloaded from the main server. JMeter was set up to request ten manifests and ten splash screens for each boot image.

As can be seen in figure 4.2, the response time of the manifest is impacted severely by the download of boot images. The response time is almost 50 times slower. In test 1b the bandwidth is saturated, meaning the server cannot send responses any faster.

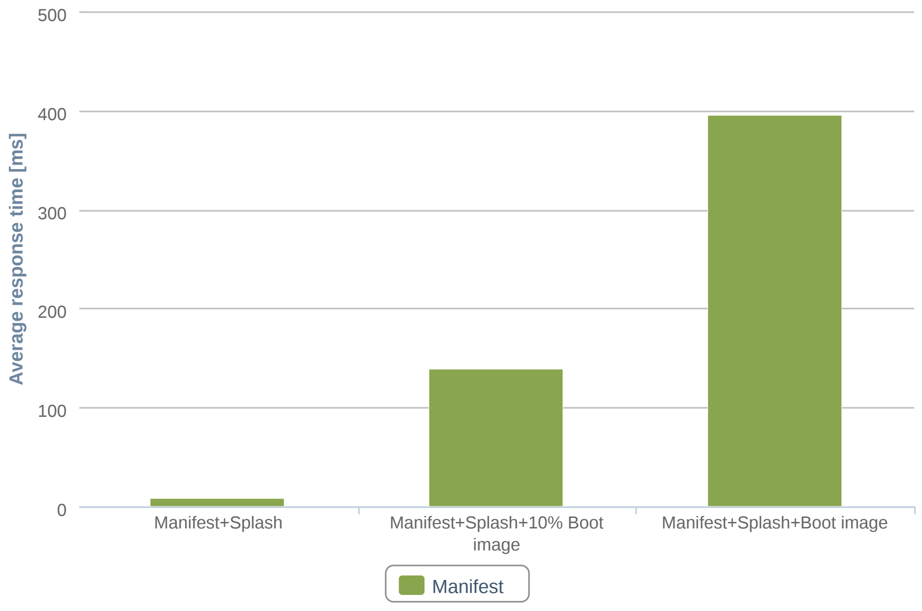

Figure 4.2: *Test 1: Manifest average response time*

Authentication

The next test (2) examines how the performance of the service is affected by the authentication mechanism that is used. The authentication applies SHA1-HMAC to the request which is a computationally intensive task. It is of interest to observe what degree the performance degrades to when the authentication mechanism is active.

The tests involved requests with and without authentication layer. Different message lengths were used to show what difference this makes. The extra payload in the body was not used by the server in any way. It was only used to increase the message to sign.

The signature computation is a blocking computation, so the tests were performed to see how other requests behaved when using authentication. Uploading of a boot image was chosen to exemplify this.

Upload

The third test (figure 4.6) illustrates how the performance of the server is impacted by a CPU-intensive task. This was simulated by making a request every 1/10th of a second. After a short time the server started to calculate the hash sum of an uploaded file. As can be seen in figure 4.6 the response time for requests arriving while this task is running is much higher (9 seconds in worst case) than when the task is not running. This is because of the cooperative concurrency

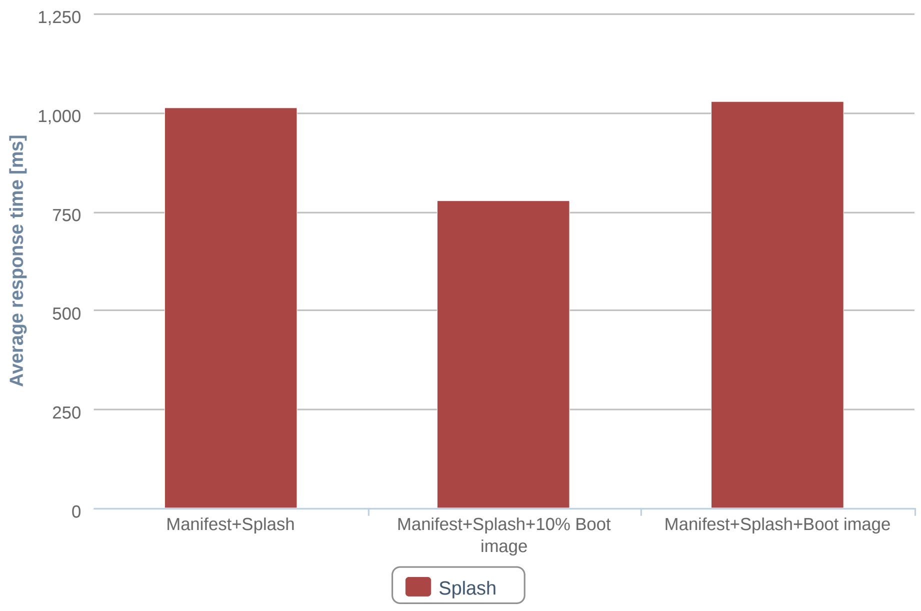

Figure 4.3: *Test 1: Splash image average response time*

model in **node.js**, which does not force any task to return control of the CPU. The calculation of the hash sum runs until completion, during which time no other requests can be handled. The incoming requests are instead put on hold in a queue. Hence, the wedge shape in the response time graph.

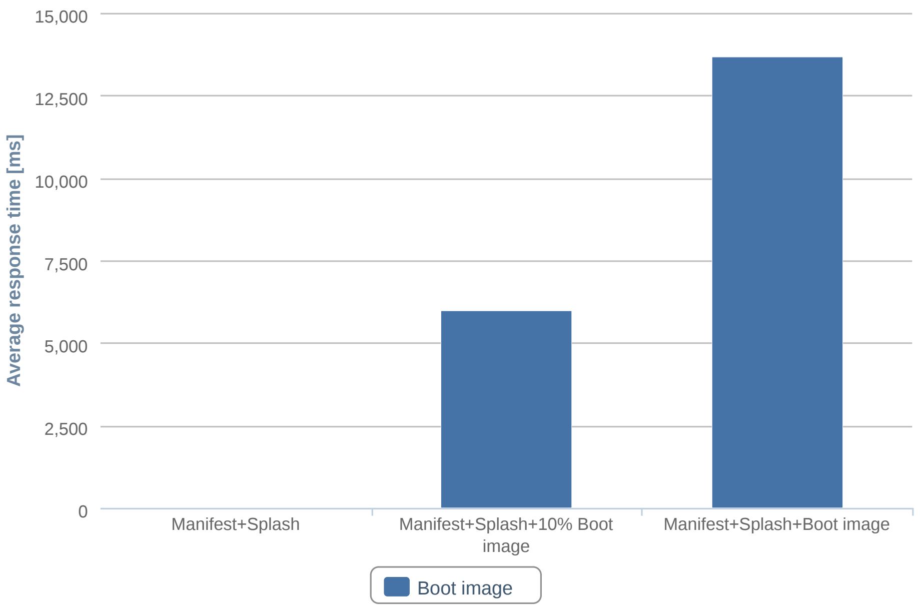

Figure 4.4: *Test 1: Boot image average response time*

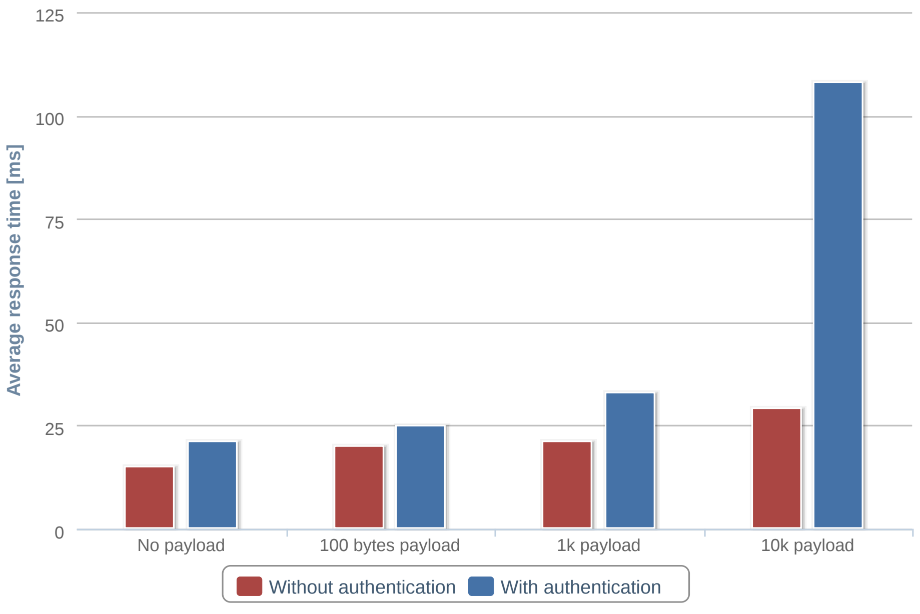

Figure 4.5: *Test 2: Impact of authentication*

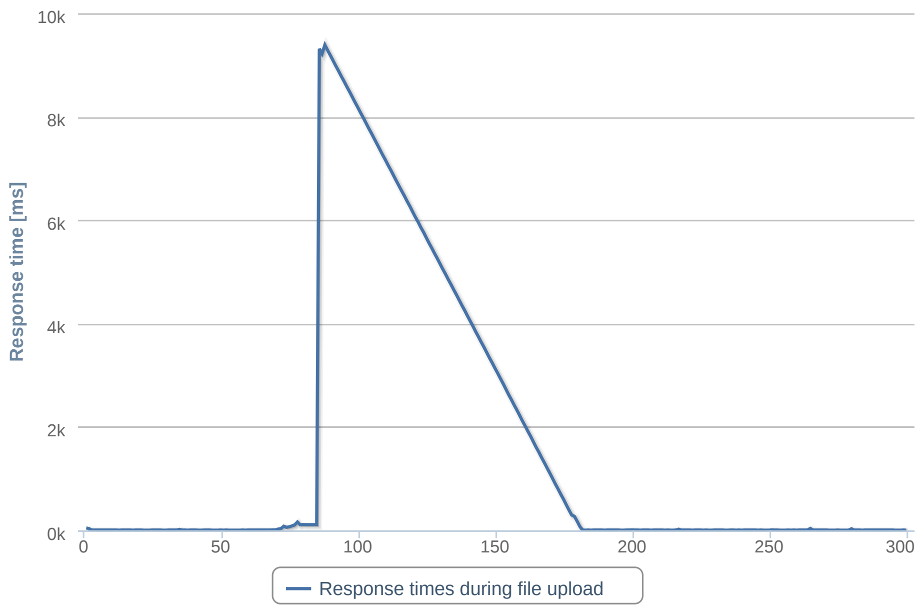

Figure 4.6: *Test 3: Response time during long-running blocking task*

5 Discussion

This chapter discusses the results presented earlier. It begins with an evaluation of the asynchronous paradigm followed by HTTP authentication. Then the maturity of **node.js** as a platform and its ecosytem of modules is evaluated. After that the performance of **node.js** solutions is discussed. Lastly some thought on front-end development based on our experience from the implementation of Wellington.

5.1 Thoughts on asynchronous programming

During our work with **node.js** we have encountered some situations that were new to us. The style of creating callback functions and passing them around differs from other paradigms we have encountered earlier but was rather easy to grasp. As the code grew larger however, some functions were getting too messy. Multiple levels of nested anonymous functions can be hard to follow and the execution does not necessarily flow from top to bottom.

The use of promises (section 3.3.5) was a good way to counter the mess. Promises made heavily nested and asynchronous code look more sequential and easier to follow. The technique could also be used to prevent race conditions by writing code like: when this value is ready, do this.

During development we tested working with web sockets (*The WebSocket API, Editors Draft* 2012) for real-time two-way communication between server and client. Web sockets can be used to push data to the client, instead of letting the client periodically poll for changes. However, Wellington did not need to push state to the client, so no web socket technique was used in the end. Our impression is however that it is easy to achieve real-time communication via websockets in **node.js**.

This switch in paradigm have not been a major issue, switching language or from one project to another may impose as great or greater changes in the way one has to think as a programmer. The mental model is not overly complex as long as not too many resources are shared.

Node.js is not the only platform that supports asynchronous programming. In the planned version of C# (5.0), there will be support for writing asynchronous code as well.

5.2 Maturity of the platform

In this section, the question about **node.js**'s maturity and its ecosystem is discussed.

Node.js is still a work in progress, and the developers have not yet released version 1.0. The API is not fully finalized and changes to the API occur between versions. Upgrading to a newer version of **node.js** is not always backwards compatible.

In the documentation of **node.js** there are stability remarks that specify if a certain part of the API is likely to change or not. This make the programmer aware of what might change in future releases of **node.js**.

Nevertheless, there are large deployments of web services built upon **node.js**. One example is Voxer that are serving "billions" of requests per day (Ranney, Matt (@mranney) 2011).

Node.js comes bundled with a package manager, **npm**. At the time of writing the *npm registry* (2012), a list of all packages in npm, contains a total of 9,420 packages. There are a lot of packages that do the same thing and try to solve the same problems. Moreover, many of them are in a very early stage of development. As a result, this can make it hard to find the right package to use in certain situations. From time to time, we have found that some packages are no longer being maintained.

Node.js's API is quite low level and that is by intention. Therefore, a lot of libraries are needed to make development of web pages easy and fast. There are modules for most functionality, but there is still no coherent full stack framework (such as rails and django). Since the ecosystem around **node.js** is young it does not yet have as feature-rich components as there are in some other languages and ecosystems.

Much functionality that is already present on other platforms is being constructed for **node.js** as well. Everything is not complete yet, but it is progressing. During the course of this thesis work a lot of improvements to both **node.js** and other modules that were used have been observed.

5.2.1 Tools to write better JavaScript

As discussed in section 3.2.4 JavaScript has good and bad parts. There are many tools and techniques to write JavaScript using only the good parts. CoffeeScript is a language that compiles into JavaScript. It helps avoid a lot of problems by compiling to high quality JavaScript that follow best practices. For example, CoffeeScript avoids the problem of automatic semicolon insertion all together. Moreover, it optimizes for-loops and allows writing more expressive code with list comprehensions and easier use of variadic arguments. Some disadvantages with CoffeeScript is covered in section 4.2.

Another toolset is Google Closure Tools which contain a compiler that can perform static type checking via structured comments that add information about types. It also acts as a minifier and can prune unused code.

Google Web Toolkit (GWT) compiles Java-code to JavaScript. There is an implementation of **node.js** in GWT (Retz 2011).

Neither Google Closure Tools nor GWT have been tested in this thesis work.

5.3 Performance

One thing we thought of when starting to evaluate **node.js** was if the merging of web server and web application in **node.js** would result in bad performance because parts that would be built in C or C++ in other solutions (like PHP or Python on Apache) would be built in JavaScript. Smedberg (2010) shows that JavaScript has become a much faster language than it has been historically. It is still possible to write faster and more efficient code in C or C++ than in JavaScript, but then the code is often longer and more complex.

A problem we found was when calculating the authentication hash (SHA1-HMAC) of a large file upload (30 MB). Calculating a hash is a very CPU-intensive task and in our case it took 9 seconds. During this time the server could not respond to other requests. Hashing/authentication of normal requests is fast enough, and no noticeable delay is introduced for the end user. But that a file upload makes the server unavailable for such a long time is not an acceptable solution. For the scope of implementing Wellington, a "simple-auth" mechanism was introduced to speed up file uploads. The "simple-auth" simply means to sign only a subset of the request; in this case by simply ignoring the request body. This is a trade-off of security in favor of performance, and could result in a more vulnerable system. A malicious attacker might be able to change what a user is uploading to another file.

Since it is possible to add native modules to **node.js**, this could be used to achieve higher performance by writing an optimized native version. Since security wasn't a highly prioritized requirement, no more work was put into increasing the hashing performance.

5.4 Front end development

We have learned a few things about front end development during this thesis work.

The front end in Wellington is a RIA and uses a lot of JavaScript. This made it important to structure the front end properly and to use design patterns like MVC.

More functionality has entered to the browser, like routing and template rendering. This took more time to develop than static content templates. It also made it important to test the product in many different browsers, since front end bugs will not only be cosmetic but may actually break functionality. Since we targeted and developed on very recent browser versions, some issues appeared in older browsers.

The front end is to be considered an actual application and not just something that is generated on the server.

5.5 Future Work

It would be interesting to test **node.js** in a large scale deployment with multiple servers, to see how it actually scales out. To make this a fair test one also has to put effort in optimizing the code to make most efficient use of the platform.

Another test would be to implement the same service in another language using another platform and compare performance and code maintainability.

There are other rising platforms based on the event driven model who want to compete with **node.js**. Alternatives such as Vert.x[1], Twisted, eventmachine, Lift[2], and Tornado[3].

Since we chose to use MySQL as database, this imposes some limitations on the number of concurrent connections. For future improvements it might be worth investigating if it is worth replacing MySQL with a NoSQL database, such as MongoDB or CouchDB.

[1]`http://vertx.io/`
[2]`http://liftweb.net/`
[3]`http://www.tornadoweb.org/`

6 Conclusions

It is possible to write web services using **node.js** and we have done so. The asynchronous programming style can be confusing at times and needs some thought but through Promises/Deferred and other frameworks and tools available as open source much of the complex cases can be solved easily.

JavaScript is a powerful and expressive language, even though it is often misunderstood, and is suitable for the event based paradigm for example with the help of closures. Closures remedy the stack handling problem from many other languages where this has to be handled manually when building cooperative concurrency.

A cautious recommendation to use **node.js** is hereby issued. It is a good platform to build upon but it does not solve all problems in one stroke and building large scalable applications is still a difficult task. Use it but be aware of the problems that might occur and its limitations.

We think that **node.js** is a good way for programmers with knowledge of JavaScript, and other higher level languages, to learn more about network programming. The event based paradigm of **node.js** is a good start to write simple and yet powerful networking applications without sacrificing performance when building more complex systems.

Bibliography

Adya, A., Howell, J., Theimer, M., Bolosky, W. J. & Douceur, J. R. (2002), Cooperative task management without manual stack management, *in* 'Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference', ATEC '02, USENIX Association, Berkeley, CA, USA, pp. 289–302.
URL: `http://dl.acm.org/citation.cfm?id=647057.713851`

Andreessen, M. (1998), www.
URL: `http://web.archive.org/web/20080208124612/http://wp.netscape.com/comprod/columns/techvision/innovators_be.html`

Ashkenas, J. (2012), *CoffeeScript*.
URL: `http://coffeescript.org/`

Baker, G. & Arvidsson, E. (2012), 'Optimizing javascript code', www.
URL: `https://developers.google.com/speed/articles/optimizing-javascript`

Basili, V. (1996), The role of experimentation in software engineering: past, current, and future, *in* 'Software Engineering, 1996., Proceedings of the 18th International Conference on', pp. 442 –449.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. (2001), 'Manifesto for agile software development'.
URL: `http://www.agilemanifesto.org/`

Crockford, D. (2001), 'Javascript: The world's most misunderstood programming language', www.
URL: `http://www.crockford.com/javascript/javascript.html`

Crockford, D. (2008), *JavaScript: The Good Parts*, O'Reilly Media, Inc.

Dangoor, K. (2009), 'What server side javascript needs', www.
URL: `http://www.blueskyonmars.com/2009/01/29/what-server-side-javascript-needs/`

Dangoor, K. et al. (2012), *CommonJS API*.
URL: `http://www.commonjs.org/specs/`

Dziuba, T. (2011), 'Node.js is cancer', www.
URL: `http://teddziuba.com/2011/10/node-js-is-cancer.html`

Ecma International (2011), *ECMA-262: ECMAScript Language Specification*, 5.1 edn, ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland.
URL: `http://www.ecma-international.org/publications/standards/Ecma-262.htm`

Eich, B. (2008), www.
URL: `http://brendaneich.com/2008/04/popularity/`

Elhage, N. (2012), www.
URL: `http://blog.nelhage.com/2012/03/why-node-js-is-cool/`

Fielding, R. T. (2000), Architectural styles and the design of network-based software architectures, PhD thesis. AAI9980887.

Florencio, D. & Herley, C. (2007), A large-scale study of web password habits, *in* 'Proceedings of the 16th international conference on World Wide Web', WWW '07, ACM, New York, NY, USA, pp. 657–666.
URL: `http://research.microsoft.com/pubs/74164/www2007.pdf`

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A. & Stewart, L. (1999), 'HTTP Authentication: Basic and Digest Access Authentication', RFC 2617 (Draft Standard).
URL: `http://www.ietf.org/rfc/rfc2617.txt`

Google (2012), 'V8 design elements', www.
URL: `https://developers.google.com/v8/design#mach_code`

Herman, D. (2011), 'Why coroutines won't work on the web'.
URL: `http://calculist.org/blog/2011/12/14/why-coroutines-wont-work-on-the-web/`

Joyent (2012), 'Node's goal is to provide an easy way to build scalable network programs', www.
URL: `http://nodejs.org/about/`

Joyent Inc. (2012), 'Node's goal is to provide an easy way to build scalable network programs', www.
URL: `http://nodejs.org/about/`

Kegel, D. (2011), 'The c10k problem', www.
URL: `http://www.kegel.com/c10k.html`

Kowal, K. (2009), www.
URL: `http://arstechnica.com/web/news/2009/12/commonjs-effort-sets-javascript-on-path-for-world-domination.ars`

Levis, P. (2006), 'Tinyos programming', www.
URL: `csl.stanford.edu/~pal/pubs/tinyos-programming.pdf`

Modules/1.1 - CommonJS Spec Wiki (2012), www.
URL: `http://wiki.commonjs.org/wiki/Modules/1.1`

Mozilla Developer Network (2012), *Threads*.
URL: `https://developer.mozilla.org/en/Code_snippets/Threads`

Network, M. D. (2012), 'Comparison operators', www.
URL: `https://developer.mozilla.org/en/JavaScript/Reference/Operators/Comparison_Operators`

nodejs wiki - Control flow / Async goodies (2012), www.
URL: `https://github.com/joyent/node/wiki/modules#wiki-async-flow`

npm registry (2012), www.
URL: `search.npmjs.org`

Ousterhout, J. (1995), 'Why threads are a bad idea (for most purposes)'.
URL: `http://www.stanford.edu/~ouster/cgi-bin/papers/threads.pdf`

Promises/A - CommonJS Spec Wiki (2012), www.
URL: `http://wiki.commonjs.org/wiki/Promises/A`

Ranney, Matt (@mranney) (2011), Tweet. "After some great support from the node core team, Voxer is serving millions of users billions of requests per day with node 0.6.".
URL: `https://twitter.com/#!/mranney/status/145778414165569536`

Resig, J. (2008), 'Javascript performance stack', www.
URL: `http://ejohn.org/blog/javascript-performance-stack/`

Retz, C. (2011), 'gwt-node', www.
URL: `https://github.com/cretz/gwt-node`

Severance, C. (2012), 'Javascript: Designing a language in 10 days', *Computer* **45**, 7–8.

Smedberg, F. (2010), 'Performance analysis of javascript'.

Sparsky, J. (2001), 'Human task switches considered harmful', www.
URL: `http://www.joelonsoftware.com/articles/fog0000000022.html`

Sussman, G. J. & Steele Jr., G. L. (1975), Scheme: An interpreter for extended lambda calculus, *in* 'MEMO 349, MIT AI LAB'.

The Apache Software Foundation (2011), *Module Index*.
URL: `http://httpd.apache.org/docs/2.0/mod/`

The jQuery Foundation (2012), *Deferred Object*.
URL: `http://api.jquery.com/category/deferred-object/`

The WebSocket API, Editors Draft (2012), www.
URL: `http://dev.w3.org/html5/websockets/`

von Behren, R., Condit, J. & Brewer, E. (2003), Why events are a bad idea (for high-concurrency servers), *in* 'Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9', HOTOS'03, USENIX Association, Berkeley, CA, USA, pp. 4–4.
URL: `http://capriccio.cs.berkeley.edu/pubs/threads-hotos-2003.pdf`

von Leitner, F. (2003), 'Scalable network programming', www.
URL: `http://bulk.fefe.de/scalable-networking.pdf`

Welsh, M., Gribble, S. D., Brewer, E. A. & Culler, D. (2000), 'A Design Framework for Highly Concurrent Systems', *UC Berkeley Technical Report UCB/CSD-00-1108* .
URL: `http://www.eecs.harvard.edu/~mdw/papers/events.pdf`

A
Requirements

This appendix contains the requirements that were placed on the product, Wellington.

This list contains the requirements of the system:

- A server should be store the boot configuration of STBs
- An STB should be able to request and retrieve a boot image via HTTP
- The system should be able to handle the requests from the STBs via the protocol implemented in the STBs software
- A user of the system should have a list of its own STBs
- It should be possible to use the system via a modern web browser i.e the latest versions of Chrome and Firefox
- **Node.js** should be evaluated as server-side environment
- The server should expose an API (so it's possible to programatically manage the system if needed)
- The API is to follow the REST principles
- The server should be securely accessible via public Internet by authorized users
- A user should be able to choose a boot image that will be handed to a specified STB the next time the STB reboots
- The web interface should be "easy to use and look good"

- The server should be able to generate a splash image with information about IP, MAC, boot image and url to the server. This splash image should be shown on the STB when booting.
- It should be possible to upload boot image files to the server
- The system should be distributed so that a user can serve boot images from his/her own computer
- Each STB and boot image should have a comment associated with it to easier recognize it
- There should be a CLI which can be used to achieve the same things as the web GUI