Akka.js, an actor system for Scala.js

Faculty of Computer Science and Engineering

Bachelor in Computer Science and Engineering

Candidate

Gianluca Stivan
ID number 10178

Thesis Advisor

Prof. Mouzhi Ge

Co-Advisor

Prof. Sven Helmer

Academic Year 2014/2015

Akka.js, an actor system for Scala.js
Bachelor thesis. Free University of Bozen/Bolzano

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: June 29, 2015

Author's email: me@yawnt.com

Abstract

This thesis is about the conception and development of Akka.js, a library written in Scala which thanks to a Scala-to-JavaScript compiler is apt to run on JavaScript runtimes such as Node.js and every major browser. Akka.js is actually a port of Akka, a Scala framework for the Java Virtual Machine, which, among many features, sports a complete implementation of the Actor Model a-la Erlang.

Multicore computing changed the landscape of programming in recent years. Issues that parallel programming has brought to light are discussed with countermeasures developed by computer scientists. The Scala.js compiler, the tool responsible for converting pure Scala to vanilla JavaScript, is presented. The framework Akka is also introduced and finally Akka.js, the main topic of these thesis, along with the rationale of why such a project can be immensely beneficial.

The document introduces first the existing projects on top of which Akka.js is based, explaining how they differ. The main section about Akka.js is divided into two parts, in the first one the structure of the project is described, in the second the implementation, presenting design decisions, difficulties encountered and solutions adopted.

At the end, a paragraph details how the project can be extended and the steps to take in order to turn Akka.js into a complete, battle-tested framework for web development.

Acknowledgments

This thesis wouldn't exist without the help and support of many people, whom I'd like to thank explicitly. First of all, thanks to my parents, who although not always agreed with me, always did support me and encouraged me to pursue a thesis in a different environment. Thanks to my sister Samira as well, with whom I shared the stress of deadlines and exams!
Thanks to Prof. Mouzhi Ge and Prof. Sven Helmer for the invaluable feedback and help they provided while writing down this document. Finally, thanks to the team at Unicredit Research and Development, who supported me while developing the topic of this thesis.

Contents

Chapter 1

Introduction

1.1 Rise of the multicore machines

There is a trend which has been part of the computing world for the past decade which is forcing computer scientists to rethink the way they write computer programs: the rise of multicore processors. This CPU generation has brought upon us a computing model which was before only privy to big corporations: parallel programming which, together with an incredible amount of advantages, unfortunately also presents some well known disadvantages.

Such a model, where execution of tasks is not deterministic, is bound to produce a lot of edge cases which, if not correctly handled, can become critical issues. Among the most frequent bugs like deadlocks and memory leaks, there are a couple which are particularly interesting to the purpose of this thesis: *race conditions* and problems related to *shared memory access*

A race condition appears when the result of a computation depends on the order in which the single steps are executed, resulting usually in a non consistent behaviour of the code.

Shared memory problems, instead, fall into a trickier category because they usually manifest themselves in many different ways depending on the specific case. At the core however, what happens is that the result of a read/write operation in a memory address is not guaranteed to be atomic because threads share address space (and thus memory). A classic example might be an integer which is decremented by two threads, but in the end is only decremememented once, because the last thread commits read the value before the second committed.

Moreover, although not bugs, parallel programs are notoriously harder to debug, deploy and maintain. This has urged researchers and programmers alike to research a model which could simplify dealing with this enormous complexity and, among the many that were developed, there is the Actor Model.
The Actor Model was the result of a joint effort between Carl Hewitt, Peter Bishop and Richard Steiger. Published in 1973, this revolutionary architecture was, accord-

ing to its creators, inspired by physics and programming languages such as Simula and LISP.

The philosophy at the foundation of this model is that everything is an actor, where an actor is an entity which does exactly three things:

- send messages to other actors
- create a finite number of actors
- react with a defined behaviour to the next incoming message

Every actor is identified by an 'address', which uniquely identifies it in the domain. The address is particularly interesting because it introduces the notion of *location transparency*, meaning that the physical location of an actor is not needed, as long as an address pointing to it is available, because the delivery mechanism will take care of everything.
Other two important innovations that the Actor Model introduced were decoupling sender from communications and the inherently parallel nature of its three tasks, which can be carried out simultaneously. The first allows to enable asynchronous communication, while the latter makes the model suited to massively concurrent tasks. In this thesis an implementation of the Actor Model will be presented, and it will also be explained how it solves the issue mentioned above.

Chapter 2

Background

2.1 Concurrency and Parallelism

Concurrency, at its core, means that multiple tasks can safely run at the same time and are not run sequentially, but rather in interleaved time periods. One of the best known abstractions for performing concurrent tasks are threads. While they are not by any means a novelty, having been around since the late '60s, they do introduce a useful distinction:

- *cooperative* multithreading occurs when concurrency is achieved when one thread willingly relinquishes control of the execution to another

- *preemptive* multithreading occurs when a scheduler is in charge of deciding the running task and can, at any moment, swap the running thread with a previously idle one

Albeit very useful, concurrency just simulates the perception that tasks are running in parallel as the processor can only process one task simultaneously. That was the case until multicore processors started appearing on the market in the early 2000's, which meant that, for the first time, programs could truly run at the same time.

Parallel computing is then a specific kind of concurrency. A one in where tasks do run in a non-sequential fashion but instead of running in overlapping time periods, they run simultaneously.

As previously mentioned, parallelism can become a source of bugs in computer programs. For instance:

```
class Addition {
  var x = 0
  def add(y: Int) = {
    x += y
  }
}
```

If two threads were to access an instance of the class at the same time, the following scenario would be likely:

- thread A reads x (0)
- thread B reads x (0)
- thread B adds 2
- thread B writes 2 back to memory
- thread A adds 3
- thread A writes 3 back to memory

The result would be a count of 3, whereas it should have been 5. Of course the order of operations is decided by the operating system, so A could read first and so on. This is a textbook example of a race condition.

2.2 Scala and Scala.js

Scala is a programming language created by Prof. Martin Odersky at the École polytechnique fédérale de Lausanne. Its main characteristic is its multi-paradigm nature which combines element of functional and object-oriented programming and its compatibility with the Java ecosystem, thanks to the fact that it compiles directly to Java bytecode and runs on the Java Virtual Machine. It has an advanced type system with support for algebraic data types, higher-oder- and anonymous types. In addition it also heavily borrows concepts from LISPs and Haskell such as lazy evaluation, pattern matching, immutability and currying, albeit it uses a syntax which is closer to C.

Scala.js, on the other hand, is an ongoing effort lead by Sébastien Doeraene, PhD student at EPFL, to write a compiler that, instead of Java bytecode, targets JavaScript, a dynamic language that is heavily used nowadays due to its ubiquity in both backend and frontend applications. The compiler is able to translate the standard library and Scala code non depending on Java Virtual Machine parts, so that Scala programs can run in both the browser and server side environments such as Node.js. An optimizing pipeline is also employed (Google Closure Compiler) which allows the minification of the program to reduce code size.

2.3 Akka

Akka is a library written for the Java Virtual Machine which enables programmers to write scalable, resilient and responsive applications. It does so by providing a high-level abstraction built on top of the Actor Model and, although written in Scala, it can be used from other JVM languages as well.

It was originally created by Jonas Boner, now CTO at Typesafe, but has since evolved to provide all kinds of abstractions over actors and is included by default in the Scala standard library. The philosophy behind the platform is based on the core points of the Reactive Manifesto which include:

- Providing simple concurrency and distribution through high level abstractions

- Being resilient by design by employing self-healing actors and supervision trees
- Achieving high performances (Akka is able to process 50 million msg/sec and can fit 2.5 million actors per GB of heap)
- Being elastic and decentralized, through load balancing, routing, partitioning and remoting
- Promoting extensibility thanks to its built-in extension system

It also became part of the Reactive Stack, which is a group of libraries officially supported by Typesafe, the company co-founded by Prof. Martin Odersky, the creator of scala. In this thesis, *Akka.JVM* will identify the original Akka, while *Akka.js* refers to the JavaScript port (see next section).

Furthermore, the following example can be useful to understand how a framework such as Akka can help when dealing with bugs such as the race condition described above.

```
class Addition extends Actor {
  var x = 0
  def receive = {  // what does it do when it receives a message?
    case y: Int => // if it is an Int
      x += y
  }
}
```

In practice what happens, when two threads interact with this actor is the following:

- thread A sends a message *2: Int* to Actor
- thread B sends a message *3: Int* to Actor
- Actor reads x (0) and processes the first message
- Actor writes 3 back to memory
- Actor reads x (3) and processes the second message
- Actor writes 5 back to memory

Again, the first message processed could have come from the thread B, but since messages are picked from a queue, there's only at most one task being executed on an actor at any given time which makes it considerably harder to write programs that suffer from race conditions.

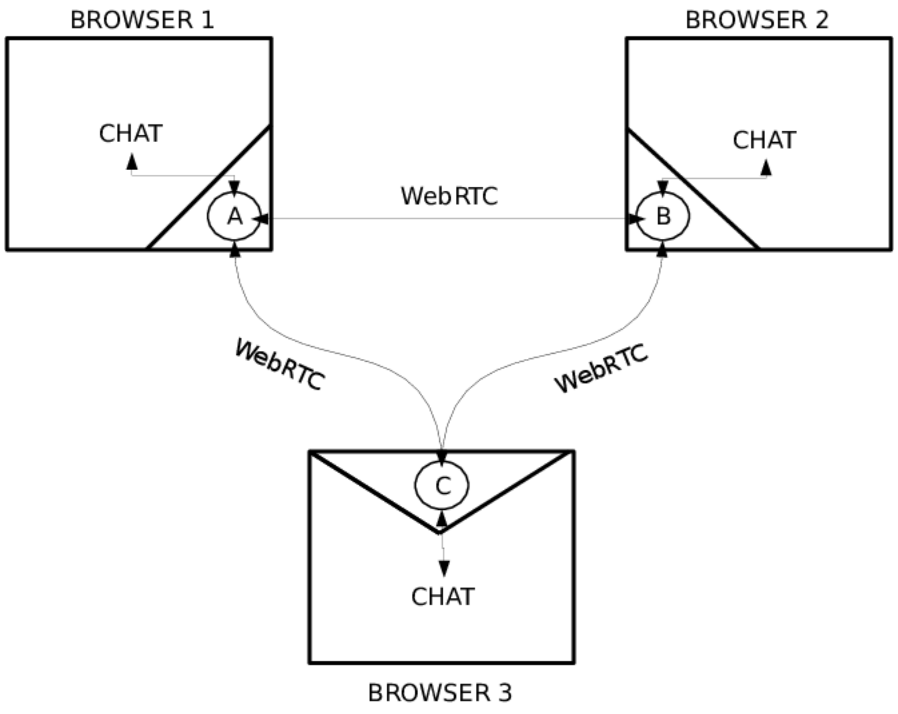
2.4 Akka.js

Akka.js is the core project of this thesis and builds on *scala-js-actor*, a previous PhD semester project developed at École polytechnique fédérale de Lausanne by Sébastien Doeraene. The original codebase has been pretty much rewritten from scratch to more closely mimic the Akka.JVM code, but Doeraene's work has served as a great source of inspiration for this thesis. What Akka.js really is, is a partial port of the Akka ecosystem to Scala.js, which allows Akka programs to run directly everywhere JavaScript is available. It achieves this by replacing all the Java Virtual Machines dependencies with their JavaScript counterparts while keeping the semantics of the library unchanged.

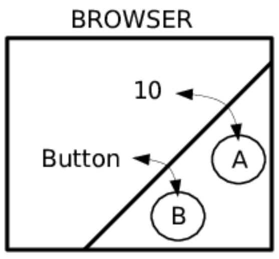
The reason why Akka on JavaScript would be beneficial are many, but the central point has its root in the concurrency model that JavaScript employs. Javascript is, infact, a single-threaded programming language which uses an event-loop model to achieve concurrency, meaning events are emitted by various components in a program and are added to an internal queue. A loop acts as a dispatcher, selecting events from the previously mentioned queue and executes the corresponding associated handler. Unfortunately this method results in an inversion of control (the so called *callbacks*) which make code hard to read. Akka.js also, given its higher-level nature, allows for expressing complex concepts in orders of magnitude less lines than their equivalent JavaScript counterpart. As such, a program written in Akka.js results in better structured code, improve testability and readability and reduce refactoring time.

2.5 Rationale behind Akka.js

Akka.js especially shines when combined with applications that are by nature concurrent or asynchronous. The following examples illustrate potential scenarios where such library could be beneficial.



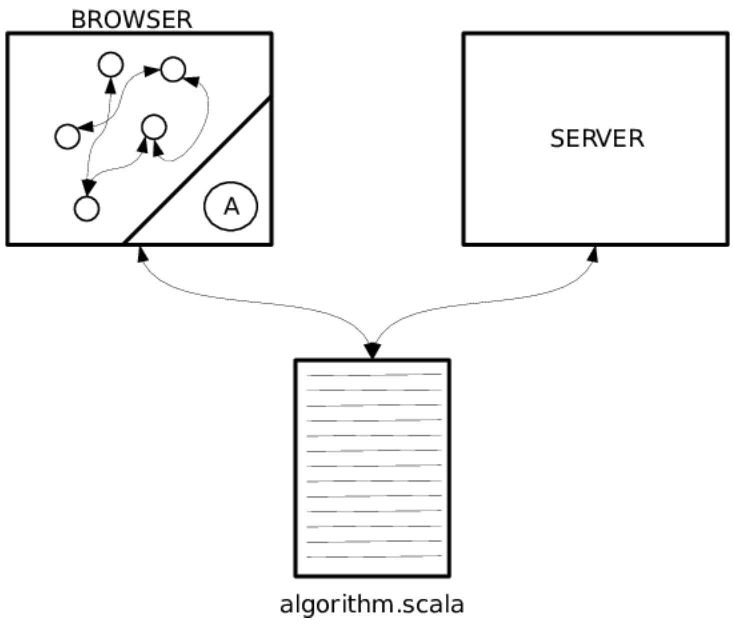
In this first scenario, a distributed application can be devised using WebRTC as a transport layer. WebRTC is a technology that allows applications to exploit the benefits of peer-to-peer, from inside the browser. Given that Akka.js has built-in routing and works by message passing, it would be possible to abstract away all the complexity from WebRTC and present to the programmer a cluster of actors which appears local but instead is running in the nodes of a peer-to-peer network.



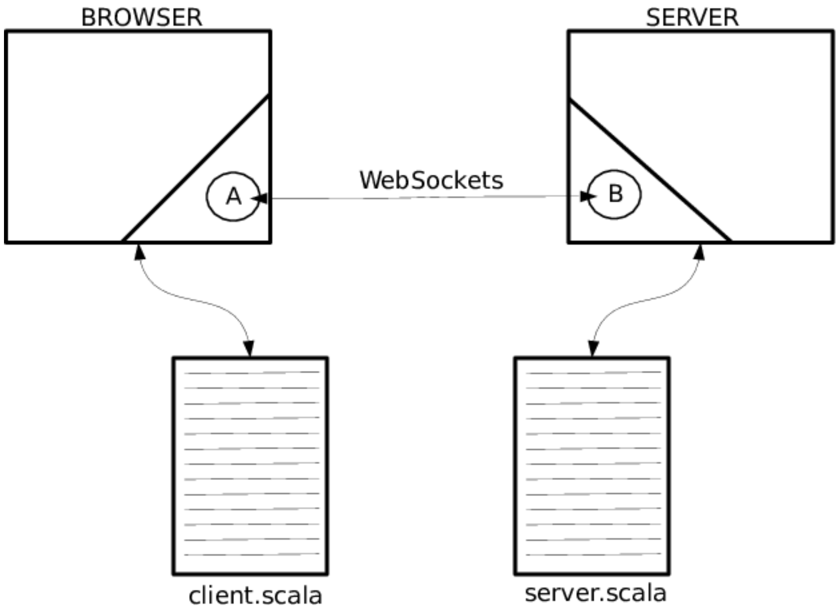
```
class B extends Actor {
  var counter = 0
  def receive = {
    case Increment =>
      counter += 1
      // ui updates automatically
    ...
  }
}
```

```
class A extends Actor {
  def receive = {
    case MouseClick =>
      b ! Increment
    ...
  }
}
```

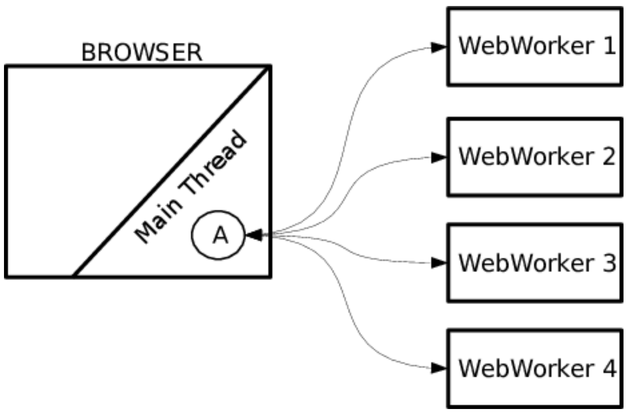
In the second scenario, a new user interface framework could be developed by exploiting the event-driven architecture of the browser. All user interactions in the browser happen through so called *events* which can be handled by the programmer. These events could be wrapped and sent to actors as messages, transforming all the interactions into one single actor system.



The third example is potentially interesting for didactic purposes. It is often difficult to picture exactly how an algorithm works just by looking at the code. Many algorithms are also implemented in Akka.JVM, so it would be interesting to insert some barriers into the algorithm which update a GUI showing the step-by-step inner workings of the code.



This fourth example depicts a bi-directional communication channel. Akka.js serves really well as an abstraction layer that can unite all the different communication protocols. In this case, a frontend application written in Akka.js can seamlessly interact with a JVM backend running Akka.JVM. The advantage is a unique approach to coding and the benefit of needing to master only one language, instead of continuosly switching from JavaScript to Scala/Java.



In this final scenario, Akka.js abstracts away the complexity of dealing with WebWorkers. As mentioned before, WebWorkers are a set of API allowing multicore in the browser. Unfortunaly because it is only possible to interact with WebWorkers through message passing, it is usually less than trivial to manage everything with

JS and a programmer needs to worry about serializing and unserializing messages. Akka.js simplifies the approach, in the sense that the programmer can use the architecture he's already familiar with and Akka.js automatically deals with the low-level components of the WebWorkers without exposing any of its complexity.

Chapter 3

Previous work and current status

3.1 scala-js-actors

Scala-js-actors, as previously mentioned, is a project that was developed as a semester project by EPFL PhD student Sébastien Doeraene in fall 2013.
Doeraene is the creator of Scala-js, a Scala to JS compiler, and scala-js-actors was designed as a way to prove Scala-js' real world capabilities.
The code contains a small subset of the entire Akka.JVM codebase, but is already able to provide support for actors, supervision and fault-tolerancy. Scala-js-actors was further extended to provide interoperability with an Akka backend through WebSockets and multicore capabilities using WebWorkers. While an amazing piece of engineering, the limits of scala-js-actors are:

- no relationship with the original Akka codebase
- no testing suite
- different semantics from Akka.JVM remote, the Akka module for communicating with remote nodes

To elaborate a bit further, the first two points where due to the fact that scala-js-actors was a research project, a proof of concept of the maturity of Scala.js, hence it was not designed with a long-term strategy in mind. The third point is more vital. Akka.JVM remote is the component that allows different JVMs to seamlessly communicate between them using the same abstraction that one would use for local actors. It's the ultimate implementation of the location transparency concept of the actor model. Whereas in Akka.JVM the programmer needs only to configure the cluster and there is no perceivable difference between remote and local actors (please note that Scala, as Erlang, uses the *a ! msg* notation to denote a message *msg* being sent to an actor *a*):

```
localActor ! "msg"
remoteActor ! "msg"
```

In scala-js-actors configuration is not supported, meaning that the programmer has to take care of the setup

```
// Main
...
WebWorkerRouter.initializeAsRoot()
val workerAddress = WebWorkerRouter.createChild("worker.js")
// Child
...
WebWorkerRouter.setupAsChild()
WebWorkerRouter.onInitialized { ... }
```

In the above example, *WebWorkerRouter* is necessary because it takes care of routing the messages to the correct actor, since there is no underlying platform that takes care of it, as it is the case with Akka.JVM remote. This poses serious problems when it comes to Akka.JVM interoperability, so for these reasons, it was decided that it would be beneficial to iterate on the original scala-js-actors codebase and provide a library that diverges as little as possible from the original Akka, so that it would be instant or trivial to port Akka applications to the browser. In particular the benefits Akka.js addresses the 3 main concerns about scala-js-actors by providing a test suite, a 1-1 ratio with Akka.JVM code and includes in the roadmap full support for akka-remote semantics.

Chapter 4

Akka.js

4.1 Structure of the project

The Akka.js project is divided into two main components:

- *akka-js-actor*: the main library, it contains the original code from Akka.JVM with the necessary modifications

- *akka-js-testkit*: Akka.JVM's testing library, ported to Scala-js to allow regression testing in Akka.js

4.1.1 akka-js-actor

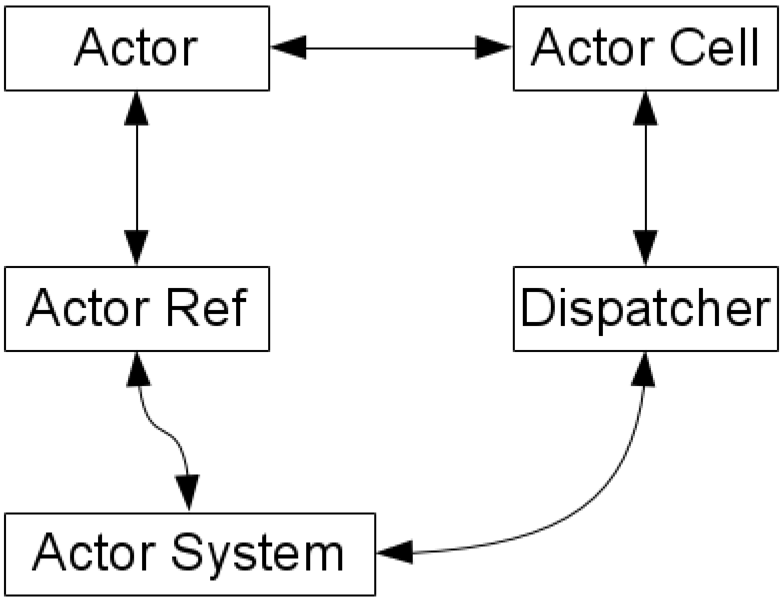
The project is divided into what is called a *Cross Build Project* in Scala.js terms. What this means is that code is divided into 2 folders

- *js/* contains JavaScript specific code which replaces JVM specific parts of Akka.JVM

- *shared/* contains code which is immutated from the Akka.JVM code and can thus be safely shared

The benefit of this approach is that it cleanly separates the independent versus dependent parts of the code, allowing for easy maintainability, testing and debuggability.

This part of the project is responsible for all the core features of Akka, including actor spawning, message dispatching, fault tolerance handling and supervision.
Its architecture is the following:



- *Actor* is the starting point of the Akka API, it's a Scala trait (similar to Java's interfaces) which the programmer extends in his/her actor implementation, by providing the required logic.

- *Actor Cell* encapsulates the state of the Actor and is where the Mailbox resides. The Mailbox is a queue which holds the messages which arrived to the actor and are yet to be processed.

- *Actor System* is an object which can contain one or more actors and acts as an Actor factory. The only way to spawn new actors is through this object, using methods such as *actorOf*.

- *Actor Ref* is an object representing a reference to an Actor. It is what the programmer uses when interacting with an actual actor and provides methods for sending messages to it, such as the *!* (bang) method.

- *Dispatcher* is a component which takes care of executing code at intervals. Its task is to process messages to and from actors by invoking the required callback in the code.

The first four components had to be adapted by removing all JVM specific code, while the *Dispatcher*, as explained below, had to be rewritten from scratch.

The original codebase of Akka numbers at around 327.000 lines of code, so a big chunk of time was dedicated to getting familiar with the technology stack and the code. A problem, as often happens in these cases, when working with such big codebases is that it is very hard to immediately test the result of a change, because so many components are dependant on each other and thus themselves require a rewrite. In the end the subset ported currently numbers at 22.000 lines of code, with plans to improve further as Akka.js matures, by porting most of the libraries which make up the Akka framework, except those that make little sense because of semantic reasons.

4.1.2 akka-js-testkit

Akka-js-testkit provides a few core utilities that facilitate the testing of Akka actors. This ensures that the cross-compiled code actually works as expected in JavaScript runtimes and produces the correct results. It's also setup as a *Cross Build Project*, similarly to akka-js-actor.

4.2 Implementation of Akka.js

4.2.1 Design decisions and Implementation Details

During the development of Akka.js, certain guidelines and principles where followed. First of all, the projects strives to maintain compatibility with the JVM version of the project. This was achieved by keeping intact the external APIs, so that the programmer feels as if he/she is using the original Akka. Moreover, this characteristic allows to easily port Scala code to the browser, without need for a lot of refactoring. This is especially interesting when combined with the intrinsic data visualization nature of the browser, because it allows to easily represent the inner workings of a Akka code through visual cues.

Dispatcher

A big change in the internal components of Akka.js was the switch from a Thread based execution model to an Event Loop one. As JavaScript, being single threaded, has no notion of threads it was necessary to refactor out the dependencies on the JVM Thread implementation and replace them with non blocking calls that dispatch on the event loop. Thankfully, due to the modular nature of Akka, it was possible to substitute the underlying *Dispatcher* with one written ad-hoc with our own custom logic to fix this problem. To be more specific the original code used *ThreadPoolBuilder* to create a thread pool and dispatch on it, whereas Akka.js now uses a *Executor* based on *setTimeout*, a function available in JavaScript runtimes which runs the callback passed as an argument after some time (the second argument) has passed.

Optimization

A performance optimization was also required. The concurrent and parallel nature of the JVM requires for Akka.JVM to use specific data structures, mostly belonging to the *java.util.concurrent* package. These data structures introduce, although acceptable in the case of the Java Virtual Machine, performance overheads which are absolutely unnecessary in JavaScript, given that it employs no threads. All such data structures where replaced by the respective JavaScript counterparts or Scala.js implementation, to ensure good performances even in a more costrained environment such as the browser.

A simple example is the *AtomicLong* class, which is guaranteed to be atomic in the JVM even when used from multiple threads. Given that JavaScript only has one thread, the *AtomicLong* implementation is a simple wrapper around a number with an incrementer, effectively removing the overhead.

Web Workers

Finally, a novel HTML5 tecnique known as WebWorkers was employed to allow for true parallel computing in JavaScript. WebWorkers are a new set of APIs which allow a programmer to spawn multiple JavaScript runtimes from the browser and interact with them through message passing.
Given how the actor model is suited for this kind of behaviour, an abstraction was implemented which resulted in a novel *Actor System* and *Actor Ref* which allows the programmer to reason simply in terms of actors, while at the same time distributing the computation over multiple processes. This is vital for application which require CPU intensive tasks, since computing in the main browser thread would inevitably result in freezing the User Interface.

4.2.2 Difficulties

The project also provided some interesting challenges, some of which are mentioned below.

JVM dependencies

The biggest problem that arose was the heavy dependency on JVM code that Akka.JVM has. JVM code is intertwined with code that can be shared with Akka.JS and there is no easy way to remove it. Examples of such dependency are threads, JVM data structures and JS semantics differing from the JVM which can all be found in the components described above (*Actor Cell, Actor System and Dispatcher*)
The problem was solved in two ways. First, code was divided in two directories, one with JavaScript specific code, another with code that is shared with Akka.JVM. Second, the JavaScript specific code was rewritten from scratch to allow Akka to run in JavaScript runtimes with the same semantics as in the Java Virtual Machine.

Semantics

A common issue that was encounered while working with the Akka codebase was that of semantic. JVM and JavaScript are two very different runtimes and as such are hardly fully interoperable. What this means is that very often patterns were encountered which could not be ported directly to the JavaScript runtime. Because of the hard costraint of keeping the API compatible it was necessary to develop some inventive work-arounds to bypass these native limitations.
For instance, in one case, Akka used a library to override a private identifier, effectively allowing code outside the class itself to modify it. Because such library is not available in JavaScript, what ended up being used was a method which in run-time parses the JavaScript function definition and accesses the underlying compiled

variable, which is not protected given the dynamic nature of JavaScript. This was just one of many problems that arose because of different semantics, patterns and libraries throughout the codebase.

Reflection

A further problem which was encountered during the development of Akka.js was due to the reflection usage on the JVM, especially in the *Actor System* component. Reflection is a feature of the Java SDK which allows a programmer to inspect an object at runtime and interact with it in ways that would not normally be possible. Unfortunately, support for such features is non-existent in the JavaScript runtime, so it was necessary to develop a custom solution. A subset of the *java.lang.reflect* APIs was developed, which is semantically identical to the JVM counterparts. For example:

```
// Akka.JVM
val clazz = Class.forName(name)
// Akka.js
val clazz = val ctor =
          name.split("\\.").foldLeft(
             scala.scalajs.runtime.environmentInfo.exportsNamespace){
             (prev, part) =>
                 prev.selectDynamic(part)
             }
```

In the above snippet, a specific property of the Scala.js runtime was exploited to emulate the behaviour of the *forName* method in *java.lang.Class*. This resulted in being able to retain the original code in other modules which used this particular functionality.
A proposal has also been submitted to the Scala.js core team, to allow for a tighter integration with the compiler. If the proposal is accepted, this will result in a superior kind of integration and it will be possible to definitely eliminate the last hacky bits in the code.

Blocking API

Finally, the usage of JVM's blocking APIs in the test code has proven to be quite the source of difficulties. Such code is common in *akka-testkit*, Akka.JVM's test framework:

```
val future = getSomeFuture()
Await.result(future, timeout)
```

What this code does, is lock the thread until a future (a computation occuring at some point in the future) is finished, after which code execution resumes normally. Given its single threaded nature it is not normally possible to block in a JavaScript runtime, effectively rendering the available Akka test code unusable. The limitation was circumvented by writing a custom event loop dispatcher that allows for blocking operations. Normally this would result in a deadlock, but given the asynchronous nature of Akka, the only blocking code is the test itself, so it ends up working just fine.

Chapter 5

Conclusion and future work

A project was presented, which focuses on cross-compiling Akka to JavaScript, effectively enabling Akka programs to run in different JavaScript runtimes (browsers, Node.js and Phantom.JS to name a few). Akka.js leverages the ubiquity of JavaScript and empowers a whole new set of complex abstractions to be easily managed from one unique interface, thanks to the elegance of the Actor Model. Different use cases where presented which explain how the project has practical use cases and can already be used to solve real world problems. Moreover, as the Akka project is evolving and turning from a simple implementation of the Actor Model, to a complex platform which can support different kind of reactive applications, the potential for the evolution of Akka.js is enourmous. There are many modules which still compose Akka.JVM and it would be interesting to explore the possibilities that some of them enable. For instance:

- Akka Cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck.
- Akka Streams is an implementation of Reactive Streams, which is a standard for asynchronous stream processing with non-blocking backpressure (meaning that data are pulled, instead of pushed, to allow for flow control)
- Akka Typed is an extension providing statically typed actors

In conclusion, Akka is a mature project with strong potential and Akka.js is a first step to harnessing this capability to improve the way software is written for the web.