Akka.js: Towards a portable actor runtime environment

Gianluca Stivan Andrea Peruffo
UniCredit - Group Research & Open Innovation
Italy
GSTIVAN.external@unicredit.eu
Andrea.Peruffo@unicredit.eu

Philipp Haller
KTH Royal Institute of Technology
Sweden
phaller@kth.se

Abstract

Multiple mature implementations of the actor model of concurrency exist. Besides several ones available for the Java Virtual Machine, there are others, for example, written in SmallTalk or in C++, targeting native platforms or other virtual machines. Recently, runtime environments for platforms such as GPUs have also appeared.

However, so far, no full-featured actor runtime environment has allowed actor programs to run, unchanged, on both Java and JavaScript virtual machines. This paper describes our ongoing effort in providing a portable implementation of the widely-used Akka actor framework.

Keywords Actors, Portability, Akka, Scala, Java, JavaScript

1. Introduction

Since its introduction in 1995 [8] there has been an increasing interest in JavaScript. Originally, this programming language was developed by Netscape to provide a lightweight interpreted language to attract non-professional programmers. Later on, thanks to a standardization process started in 1996 and still continuing today with the release of the sixth version in June 2015 [3], JavaScript has become the most popular programming language on the web [7], and is widely regarded as the assembler of the web.

It is interesting to analyze the evolution of the language. Before the introduction of Ajax [1] JavaScript was mainly used for simple scripting tasks in the browser. In 2009 CommonJS [2] and Node.js [6] proposed an ecosystem capable of running JavaScript outside the browser. With the rise of single-page web applications [9] and source-to-source compilers [5], this once very limited language is now being used to power a large variety of projects.

Despite the improvements made by the JavaScript community, there are still several issues that make developing software using JavaScript hard, which is why it is commonly considered inconvenient to work with. Since it was born as a scripting language there are a number of questionable points:

- it did not provide a module system up to ES6 [4];
- it is weakly and dynamically typed;
- the syntax for function definition is verbose;[1]
- late binding has poor guarantees;[2]
- the behavior of *this*;
- implicit casting, which not always has desired consequences.

The biggest issue that JavaScript has, though, is its almost complete lack of support for concurrency. As a language that is by default asynchronous, since all I/O is done via event-based APIs, it is surprising that the only way to write asynchronous code is through callbacks (up to ES6) and promises [21] (from ES6 upwards). This quickly leads to confusing application flow because inversion of control, i.e., the code suffers from deeply-nested functions, is prominent.

The actor model [10, 18] has been successfully used to write concurrent and distributed applications in a variety of domains, ranging from telephone switches [11], to critical infrastructure services written in Erlang [12] such as Amazon's S3 and so on. As such the model would be a perfect fit for JavaScript, but unfortunately there is no widely used or supported implementation of such a model that targets JavaScript runtimes.

After evaluating different virtual machine environments, we found in Akka [30] a mature actor framework for the JVM [20], which can be used both from Scala [27] and from Java. Given the existence of Scala.js [14], a Scala-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

AGERE! '15, Pittsburgh, Pennsylvania, USA.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Even if JavaScript's support for closures is commonly noted as being one of its's redeeming features.

[2] Early binding allows for static verification of the existence of method-signature pairs.

to-JavaScript compiler, we started investigating the feasibility of porting said Akka framework to JavaScript. Unfortunately, as the framework is written today, it is heavily dependent on the JVM.

Specifically, this paper presents our ongoing effort to enable *portability* of systems and applications based on actors in Scala [15], using the Akka actor-based middleware. Traditionally, actor-based programs in Scala have been restricted to run on the JVM, Scala's main compilation target.

We strongly believe the actor model to be beneficial to JavaScript. Enabling actor-based applications to be deployed on JavaScript runtimes presents numerous advantages. To begin with, it brings an arguably very good model, which handles concurrency and, thanks to features of widely-available implementations, also fault handling admirably, to a runtime which lacks good support for both. Moreover, it promotes application-level state isolation, which simplifies reasoning about application code. The message passing abstraction fits incredibly well to a variety of I/O operations that JavaScript provides. WebSockets, WebWorkers, AJAX, and DOM events are all asynchronous and event-based, making a wrapper that forwards events as messages to actors at the same time convenient and simple to use. Finally, enabling interactions between standard Akka server side applications and the ones deployed on JavaScript VM through transparent message passing on transports like WebSockets will ease web development in seamless server-to-client inter-operations.

1.1 Contributions

Akka.js[3] is our solution to the problem of providing a fully-featured Scala actor framework for JavaScript runtimes (ES5 and above). It enables portable actor-based applications, compiled against the Akka API, to target both the JVM and JavaScript runtimes. Another important goal of the project is seamless messaging between actors running on JavaScript runtimes (*e.g.,* web browsers) and actors running on JVMs.

More specifically, our approach can be divided into the following contributions:

- Porting the original JVM-based implementation of Akka (from now on referred to as Akka.JVM) to JavaScript;
- Techniques for sharing Scala code between Akka.JVM and Akka.js;
- A solution for enabling transparent interoperability between actors running on JavaScript runtimes and actors running on JVMs.

The second issue is of major importance: the lack of a shared code base would require *manual synchronization* between two large code bases, with a corresponding large engineering cost. Moreover, the lack of automatic, compile-time checks would make it extremely difficult to ensure the portability of Akka-based programs across Akka.JVM and Akka.js in the long run.

Using Akka.js we have developed several demo applications,[4] including an implementation of the Raft consensus algorithm [28], as well as the "reactive calculator" exercise from the Coursera MOOC on "Principles of Reactive Programming" [25]; the demos are runnable directly from within supported web browsers.

The paper proceeds as follows. We first give a brief overview of the Scala.js compiler, the Akka actor framework, as well as a previous prototype of an actor system for Scala.js (§ 2). We then discuss the major challenges encountered during the development of Akka.js (§ 3), which include reliance on JVM constructs (data structures, patterns, and libraries), usage of reflection, blocking APIs, and serialization. We also detail the solutions adopted to circumvent the presented issues. After that, we outline a number of potential use cases for Akka.js (§ 4). Finally, following a discussion of related work (§ 5), we offer an outlook and conclude (§ 6).

2. Background

2.1 Scala.js

Scala.js is an alternative compiler for the Scala language that instead of targeting the Java Virtual Machine's bytecode, targets JavaScript. As with many other efforts in and out the JVM (GWT, CoffeeScript, ...), the goal of the project is to provide a language which is higher-level than JavaScript, in order to achieve easier code reusability and to provide better abstractions and tooling. One key point of Scala.js, and crucial to the topic of this paper, is bi-directional JavaScript interoperability, meaning that plain JavaScript libraries can be wrapped by Scala interfaces and used in Scala.js and Scala.js can expose user-defined APIs to JavaScript. The project actually features a 100% coverage of the Scala language (apart from few intentional differences due to semantics) and provides a lot of useful, typed facades to the most popular JavaScript libraries. Moreover, since many compilers that target JavaScript suffer from performance penalties, it is important to note that fully-optimized Scala.js code executes at almost native JavaScript speed [14].

2.2 Akka

Akka [30] 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-based languages as well.

Akka was originally created by Jonas Bonér, and has since evolved to provide all kinds of abstractions over actors. It is also included by default in the Scala standard library.

Akka provides a number of advantages:

[3] See `https://github.com/unicredit/akka.js`

[4] See `http://unicredit.github.io/akka.js-examples/`

- Simple and high-level abstractions for concurrency and parallelism.
- Asynchronous, non-blocking and highly-performant event-driven programming model.
- Very lightweight event-driven processes (several million actors per GB of heap memory).
- Fault tolerance through supervisor hierarchies with "let-it-crash" semantics.
- Location transparency in distributed environments, pure asynchronous message passing.

Thus, both aspects of the actor model are covered, (a) an API that supports writing concurrent software using the abstraction provided by actors, and (b) a runtime library that provides powerful features for error handling and actor life-cycle management.

Moreover, Akka seamlessly integrates the actor model into the unified object-oriented and functional programming style that Scala enables [27]. Techniques for integrating actors into Scala have been explored in the context of Akka's predecessor, Scala Actors [16].

2.2.1 Programming model

To introduce Akka's programming model, we look at its components, first of all the ActorSystem, which is the very first element you interact with when using this library. An ActorSystem is a high-level abstraction over a container of actors; it is heavyweight and wraps a thread pool to abstract the underlying, JVM-specific concurrency model. All actors must be spawned within an ActorSystem. By creating several ActorSystems, it is possible to separate different runtime requirements even within the same JVM process. Creating a new ActorSystem is dead simple:

```
1  val system = ActorSystem.create
```

Akka actors are containers for state, behavior, a mailbox, children, and a supervision strategy. The detached interface that enables the interaction with the rest of the world is represented by an ActorRef, short for actor reference. Akka provides a special Actor trait[5] with concrete and abstract members that can be extended by any arbitrary subclass. In this case, the only abstract member that subclasses have to define in order to create a concrete instance of an actor is the `receive` method:

```
1  def receive: PartialFunction[Any, Unit]
```

Method `receive` returns a partial function which is synchronously applied to each message that is dequeued from the actor's mailbox. Messages are automatically processed one by one as long as the mailbox is not empty.

A common practice in Scala is to use *pattern matching* to destructure the messages an actor receives. The following simple actor prints a message to the console when it receives the string "ping."

```
1  class PingReceiver extends Actor {
2    def receive = {
3      case "ping" => println("Ping message received")
4    }
5  }
```

Since actor objects have specific properties (*e.g.,* their state is fully encapsulated), they cannot be instantiated using regular means. A previously-created ActorSystem enables spawning new actors within itself using so-called "Props" objects. A Props object is a container of (actor) creation properties which specifies various runtime configuration properties (the discussion of which is outside the scope of this short overview). At the minimum, an `Actor`-extending class containing the actor's behavior must be specified:

```
1  val actor = system.actorOf(Props[PingReceiver])
```

In order to send messages to the spawned actor, the usual `!` operator is used:

```
1  actor ! "ping"
```

In this case, the expected result, "Ping message received", is printed to the console.

Akka not only provides what was covered in this brief overview, but a rich toolkit for creating distributed actor-based systems on elastic cloud computing platforms as well.

2.3 Scala.js Actors

The original source of inspiration for Akka.js was the *Scala-js-actors* project. Developed by EPFL PhD student Sébastien Doeraene in Fall 2013, it was designed as a way to evaluate the real world capabilities of Scala.js. The code contains a small subset of the Akka.JVM codebase, but already has support for actors, supervision and fault recovery. Scala-js-actors was further extended to provide interoperability with an Akka.JVM backend through WebSockets and multi-core computation using WebWorkers.

While an inspiring piece of engineering, the limitations of Scala-js-actors are:

- No strong 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 using the same abstraction that one would use for local actors. Ultimately, it is the implementation of the location transparency concept of the actor model. In Akka.JVM the programmer needs only

[5] Scala supports a safe form of multiple inheritance through modular mix-in composition of traits. Traits in Scala can be thought of as more flexible Java interfaces.

to configure the cluster and there is no perceivable difference between remote and local actors:

```
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 Akka applications can transparently be ported to the browser. In particular Akka.js addresses the above three main concerns about Scala-js-actors by providing a test suite, a 1-1 correspondence with the Akka.JVM code base, and it includes in the roadmap full support for Akka.JVM remote semantics.

3. Portable runtime environment

In order to achieve a perfectly portable runtime environment, it is first necessary to understand the semantics of the target. JavaScript can run in many different environments, including

- Node.js, a JavaScript runtime for server side applications;
- Phantom.js, a headless, programmable WebKit browser;
- Rhino, a JavaScript engine running on the JVM;
- Browsers, in general.

These environments, albeit not identical, share some common traits, which simplify the process of understanding where to focus the porting effort. One such trait is the *execution model of JavaScript*.

3.1 JavaScript execution model

JavaScript is, as we briefly mentioned, a dynamic, single-threaded programming language. It provides some functional concepts such as closures and partial support for operating on collections (*e.g.,* map, reduce, filter). JavaScript is also inherently concurrent, since all its I/O operations are asynchronous and event based. In order to better understand how this affects the development of programs, it is necessary to explicitly specify what we mean when we say *asynchronous event-based I/O*.

JavaScript only has a few ways of interacting with the outside world: one is the Document Object Model, a tree-like structure which represents the HTML of the current web page, another is WebSockets and all the AJAX family which

include facilities to communicate with a remote server, and finally there is WebWorkers, a novel standard which allows multiple JavaScript runtimes to be spawned and communicate with one another through message passing. Every time we interact with one of these components a return value isn't immediately available, but the programmer must register explicit interest in the appropriate event. This is where the event loop comes into play. This event loop is a regular loop which listens for events and subsequentially executes the corresponding handler that the programmer has specified, using a callback mechanism.

3.2 Akka.js

Akka.js is the core project of this paper, and as previously mentioned, builds on *scala-js-actor*. 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. 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 project provided some interesting challenges, some of which are mentioned below.

3.2.1 Challenge 1: JVM dependency

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. 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.

3.2.2 Challenge 2: JVM-style reflection

A further problem which was encountered during the development of Akka.js was due to the reflection usage on the JVM. 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. 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 completely remove the last workarounds currently present in the code.

3.2.3 Challenge 3: blocking APIs

Finally, the usage of JVM's blocking APIs in the test code has proven to be a source of difficulties. 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.

3.3 Cross compilation

As long as the Akka project remains very active, we face the problem of a divergent code base and it has been so since the very first versions of Akka.js. The analysis of the key differences highlighted the following major conflicts:

- Semantic differences between components usable in a JavaScript environment;
- Java API, which is partially implemented in Java and we cannot support;
- Missing annotations vital to expose functionalities to the JavaScript runtime;
- A tiny number of small hacks used to keep the developer API clean and simple (mostly reflection-based);
- Java data structures.

Considering the advantages of having a project that can be compiled reusing a large part of the original code, it was decided to adopt some strategies in order to follow a strategy of "non-intrusive" patching as much as possible, for as many source files as possible.
In a file where implementations do not conform to the semantics of the JavaScript Virtual Machines, the implementations and interfaces are split in different files, so that Akka.js only depends on the interfaces. The changes are then forwarded to the upstream Akka repository by means of a Pull Request. Since the Java API is hard-wired directly into the interfaces, the current solution drops the methods at compile time via a compiler plugin. The same solution, using a compiler plugin, is also adopted to add Scala.js annotations at compile time, where required, through AST manipulation. By analyzing the data structures used in Akka's codebase, we must keep in mind that:

- Akka was developed with JVM, concurrency and performances in mind.
- The standard Scala collections library was designed to provide an excellent developer experience and a consistent fully functional API across different kinds of collections, despite possibly worse performance.

The obvious conclusion is that the collections that Akka employs are, for said performance reasons, those extremely efficient ones provided by the Java standard library. When the

porting started, the interoperability between the Java standard library and Scala.js was limited to a subset functions concerning Array manipulation. Instead of changing the data structures inside Akka itself, to further our goal of compatibility, Scala.js' support of Java collections was extended. Since JavaScript is single-threaded and asynchronous, there was no need to worry about supporting thread-safe data structures as this property is intrinsically provided by the runtime. Collections were therefore implemented through binding on pure Scala data structures and developed respecting pedantically the documentation (JavaDocs) of the JDK. This resulted in an extensive improvement of the Scala.js/Java inter-compatibility layer that by default ships as part of the compiler. Today, all Java data structures used by Akka are available in the Scala.js library with the exception of the "thread-blocking" ones which would totally compromise the functionality of Akka.js and are, on a JavaScript runtime, therefore not meaningful.

3.4 Serialization

Serialization is essential for *remoting*; more specifically, for communicating messages (a) between different JavaScript WebWorkers, and (b) between actors running on a JavaScript runtime (using Akka.js) and actors running on a JVM (using Akka.JVM). In Akka.JVM, serialization is provided by the underlying virtual machine – any object can be serialized automatically if its class is marked as "serializable" and all referenced objects are transitively serializable [22, 29].

It turns out this reliance on the JVM for serialization is a challenge for Akka.js: contrary to approaches like GWT, Scala.js does not emulate JVM runtime services like serialization. Moreover, Scala.js does not support runtime reflection in order to enable important compile-time optimizations that improve runtime performance and reduce the resulting JavaScript output. Therefore, using Scala.js it is impossible to implement *universal serialization* (as provided by the JVM) using runtime reflection.

Serialization is also critical for *interoperability* between JavaScript runtimes and JVMs. In particular, an important goal of Akka.js is the ability to send objects as asynchronous messages between the JavaScript and the JVM world. Consequently, objects must be serialized in a compatible way.

3.4.1 Approach

Our approach to JavaScript-JVM compatibility is based on generating *pickling combinators* [24]. Compatibility is achieved through a combination of two techniques:[6]

- Serialization is based on a platform-independent external representation.

[6] At the time of writing, our implementation is a work-in-progress. Integration with one of the next major releases of Scala Pickling is planned (see `https://github.com/scala/pickling`).

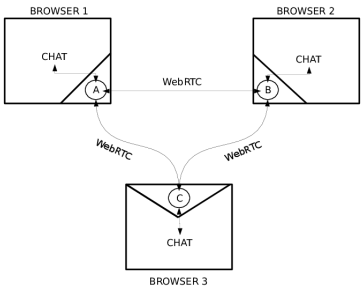

Figure 1. Remoting

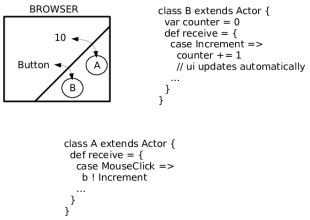

Figure 2. UI Framework

- Generated serializers (or *picklers*) are cross compiled; as a result, serialization on JavaScript runtimes has the same semantics as on the JVM.

In order to avoid additional boilerplate compared to Akka.JVM, we are exploring the use of a Scala compiler plug-in to detect declarations of classes marked as serializable. For such classes, cross-platform picklers are generated and automatically registered with the serialization subsystem of Akka.js. This approach does not require source code changes, at the cost of a minor addition to the build process.

What is noteworthy about our approach is the fact that we extend an existing, well-known serialization framework, instead of adopting a custom solution just for Akka.js. Thus, the developed portability and cross-compilation features benefit other libraries requiring serialization, such as RPC frameworks. This is similar to our extensions to Scala.js (see § 3.3), which benefit a wide range of projects.

4. Use cases

There are many potential use cases which demonstrate why Akka.js can be extremely beneficial, introducing a new way of thinking about interaction in JavaScript environments. The following examples illustrate potential scenarios where such a library could be successfully employed.

In Figure 1, 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.

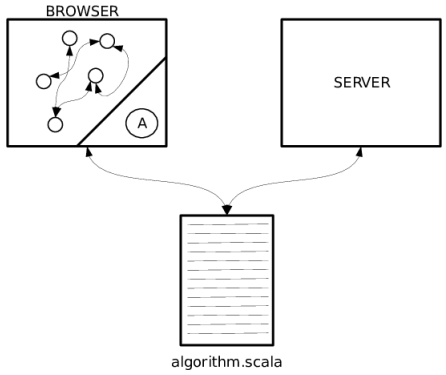

Figure 3. Algorithm visualization

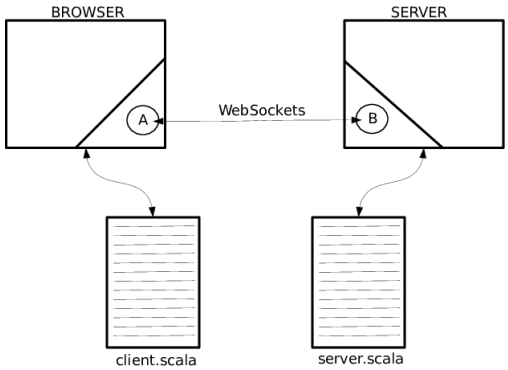

Figure 4. Bi-Directional communication

In Figure 2, 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 *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 in Figure 3 is potentially interesting for didactic purposes. It is often difficult to picture exactly how an algorithm works just by looking at a particular implementation. 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 in Figure 4 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 continuously switching from JavaScript to Scala/-Java.

In the final Figure 5, Akka.js abstracts away the complexity of dealing with WebWorkers. As mentioned before, WebWorkers are a set of API enabling the use of multi-cores in the browser. Unfortunately, because it is only possible to interact with WebWorkers through message passing, it is usually less than trivial to manage everything with JavaScript and a programmer needs to worry about serializing and deserializing messages. Akka.js simplifies the approach, in the sense that the programmer can use the architecture they are already familiar with and Akka.js automatically deals with the low-level components of the WebWorkers without exposing any of its complexity.

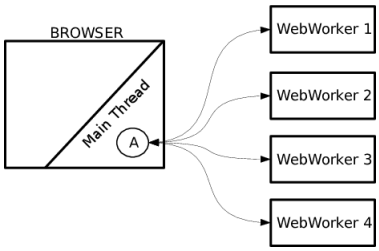

Figure 5. WebWorkers

5. Related work

The Clojure language [17] supports CSP-style programming [19] using its core.async library. This library is also supported by ClojureScript [26], a compiler for Clojure that targets JavaScript. The main difference in comparison to our effort is the fact that core.async implements the CSP model, whereas Akka.js implements the actor model. CSP also does not directly support distributed computation, meaning that something akin to location-transparent, distributed actors would not be possible.

CAF [13] is an open-source implementation[7] of the actor model in C++. Like many other mature actor frameworks it enables transparently connecting actors running on different machines and OSes via the network due to its native nature. What is most notable is that it integrates multiple computing devices such as multi-core CPUs, GPGPUs, and even embedded hardware. Moreover, it is also possible to create a message-passing interface for OpenCL backends.

Funk[8] is a library that supports functional programming in Haxe.[9] Haxe itself is an open-source toolkit based on a modern, high-level, strictly-typed programming language, a cross compiler, a complete cross-platform standard library and ways to access each platform's native capabilities. The Funk library contains a basic actor framework with an API inspired by Akka. While it enables, thanks to Haxe, code reuse on a number of different platforms, distributed actors are not supported.

Jetlang[10] are two high performance threading libraries written for C# and Java. They do not support remoting, but

[7] See `http://actor-framework.org`

[8] See `https://github.com/SimonRichardson/funk`

[9] See `http://haxe.org`

[10] See `https://github.com/jetlang`

are optimized for in-memory message passing. They support sequential delivery using a special kind of interface.

Fantom[11] is a high-level language with support for actors. It was designed with portability in mind and Fantom programs can seamlessly run on the JVM and the .NET CLR. Other goals of the programming language include: elegant APIs, modularity, support for OOP and FP, and declarative programming. Fantom also has support for JavaScript runtimes.

The Reactive Extensions model [23] has been ported from .NET to many other runtimes, including Java (RxJava by NetFlix) and JavaScript (RxJS by Microsoft). In contrast to Akka.js, these libraries are not intended to enable code written in one language to be run on another platform; instead, each library implements the same programming model on different runtimes in a way that is more or less compatible with the original .NET implementation. Moreover, the programming model differs from the actor model, e.g., concurrent processes are not first class.

6. 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 were 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.

[11] See `http://www.fandev.org/`

7. Acknowledgments

This work would not have been possible without UniCredit and specifically the Group Research & Open Innovation team, led by Riccardo Prodam. Their support, help, and encouragement has been instrumental to this work and we are deeply thankful for that! Moreover, we would also like to thank Sébastien Doeraene from EPFL, whose work (Scala.js and scala-js-actors) laid the foundations for this paper.

References

[1] AJAX. `https://developer.mozilla.org/en-US/docs/AJAX`.

[2] Common.js. `http://requirejs.org/docs/commonjs.html`.

[3] EcmaScript 6. `http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts`.

[4] EcmaScript 6 - Modules. `http://wiki.ecmascript.org/doku.php?id=harmony:modules`.

[5] List of languages that compile to JS. `https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-JS`.

[6] Node.js. `http://nodejs.org`.

[7] Redmonk programming language rankings: Summer 2015. `http://redmonk.com/jgovernor/2015/07/31/programming-language-rankings-summer-2015/`.

[8] A short history of JavaScript. `https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript`.

[9] Single-page application. `https://en.wikipedia.org/wiki/Single-page_application`.

[10] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press, 1990.

[11] J. Armstrong. Erlang — a survey of the language and its industrial applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Oct. 1996.

[12] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.

[13] D. Charousset, R. Hiesgen, and T. C. Schmidt. CAF - the C++ actor framework for scalable and resource-efficient applications. In *AGERE!@SPLASH*, pages 15–28. ACM, 2014.

[14] S. Doeraene. Scala.js. `http://www.scala-js.org/`, 2013.

[15] P. Haller. On the integration of the actor model in mainstream technologies: The Scala perspective. In *AGERE!@SPLASH*, pages 1–6. ACM, 2012.

[16] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.

[17] S. Halloway. *Programming Clojure*. The Pragmatic Bookshelf, Raleigh, NC, 2009.

[18] C. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell*, 8(3):323–364, 1977.

[19] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, second edition, 2000.

[21] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, pages 260–267. ACM, 1988.

[22] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *PPOPP*, pages 173–182, Aug. 1999.

[23] E. Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[24] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *OOPSLA*, pages 183–202. ACM, 2013.

[25] H. Miller, P. Haller, L. Rytz, and M. Odersky. Functional programming for all! scaling a MOOC for students and professionals alike. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *ICSE Companion*, pages 256–263. ACM, 2014.

[26] D. Nolen and contributors. ClojureScript. `https://github.com/clojure/clojurescript`, 2011.

[27] M. Odersky and T. Rompf. Unifying functional and object-oriented programming with Scala. *Commun. ACM*, 57(4):76–86, 2014.

[28] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In G. Gibson and N. Zeldovich, editors, *USENIX Annual Technical Conference*, pages 305–319. USENIX Association, 2014.

[29] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency - Practice and Experience*, 12(7):495–518, 2000.

[30] Typesafe, Inc. Akka. `http://akka.io/`, 2009.