An actor system for Scala.js
Semester project, Fall 2013

Sébastien Doeraene
Under the supervision of Martin Odersky

January 31, 2014

Abstract

This reports presents the design of an actor system for Scala.js that we implemented. Scala.js is a Scala to JavaScript compiler, which we wrote as previous work and enables developers to write the client-side of Web applications entirely in Scala. The actor system we designed is very similar to Akka, exposing almost the same API and abstractions. It is however designed to be run in Web page scripts, in a single-threaded environment. The implementation supports remoting with Web Workers as well as transparent communication with a server running Akka on the JVM.

1 Introduction

The goal of this project was to implement an actor sytem for Scala.js [7]. In doing so, we wanted to expose a model and an API as similar as possible to that of Akka [1], so that Scala developers could reuse knowledge between their client code and server code. It turned out that Scala.js was mature enough that we could do much more than that: our actor system is closer to a port of Akka to Scala.js than a new implementation from scratch, effectively sharing several thousands of lines of code with the original implementation. However, three core aspects of the underlying implementation had to be completely rethought, which is what we will focus on in this report:

- From a thread-based implementation to an event loop-based implementation,
- Serialization of messages for "remoting",
- The "remoting" itself, with its two flavors: across Web Workers and between client and server on top of Web Sockets.

In this report, we will refer to our library as Akka/JS.

2 An event loop-based implementation

It may seem contradictory to implement an actor model, which is inherently concurrent and asynchronous, on a purely single-threaded platform like Scala.js. However, concurrency and asynchrony must not be confused with parallelism. While parallelism involves physically executing different tasks at the same time, e.g., on multiple processors or multiple machines, concurrency is a form of modularity which allows to model software components as independent units of execution and behavior which can communicate between each other. Similarly, asynchronous execution does not imply parallelism, nor concurrency. In an asynchronous call, the caller does not wait for the callee to finish (or begin) its execution, but continues instead, potentially holding a handle to the result which will be available later (JavaScript calls this handle a Promise [5], whereas Scala calls this a Future [6]).

All JavaScript environments provide an *event loop*, which is the top-level executor. Events are posted to the global event queue, and are dequeued and dispatched by the top-level event loop. There are several sources of events:

- user interactions with the Web page,

- I/O events: success or failure of an XHR, messages arriving to the Web Worker's port, messages arriving through a Web Socket, and so on.
- user-defined timers (`setTimeout()` and `setInterval()`),
- user-defined immediate callbacks (`setImmediate()`), which is more recent and not supported everywhere, but polyfills exist that fall back on `setTimeout(f, 0)`.[1]
- and so on.

The processing of one event is essentially atomic: only when it is done can other messages be handled. Therefore, processing an event should be a short task.

2.1 Messages as events, and mailboxes

It seems like messages sent to an actor are a canonical instantiation of the more general notion of event. A naive implementation of a message send from an actor A to an actor B would be to post an event to the global event queue, which when executed will invoke the `receive` method of B. However, there are at least one good reason not to do this: with this approach, it is impossible to prioritize *system messages* over user messages.

System messages are special messages exchanged between actors to implement the actor model itself, in particular the life cycle and supervision, as well as death watch registrations. For correctness of the various internal algorithms, an actor must always process outstanding system messages before any enqueued user message, even if a user message was enqueued before a system message.

If messages were all posted to the global event queue, a user message enqueued before a system message would be handled before the latter. Instead, each actor maintains two queues of its own, one for system messages and one for user messages. The pair of them forms a *mailbox*. When a message is posted to a mailbox, the mailbox posts itself in the global event loop. Other messages posted to the mailbox are simply enqueued as long as the mailbox has not been scheduled by the event loop. When it is scheduled, it dispatches (processes) system messages before user messages, as long as there are messages.

Note that since an actor can post messages to itself, this can potentially never return. Therefore, one processing of a mailbox is bounded in terms of number of messages and execution time. If one of the bounds is reached before the mailbox has completely emptied, it is immediately reposted to the global event queue. This will give the opportunity to other mailboxes, but also to other sources of events, to be processed before that same mailbox runs again, ensuring progress of the whole application.

The attentive reader might suggest to always dispatch system messages synchronously, and always post user messages to the global event loop. Since there is only thread anyway, we can dispatch a system message of an actor B while another actor A is running. However, this solution would not generalize to the setups with remoting.

2.2 Non-blocking vs mutable data structures and algorithms

To achieve maximal efficiency, Akka internally uses many non-blocking data structures and algorithms. These algorithms are based on immutable data structures stored in mutable fields, the latter being updated by Compare-And-Swap (CAS) operations. In the Scala.js implementation, these optimizations become overhead, as they solve a non-existent problem.

We have redesigned and reimplemented all these internal operations so that they use mutable data structures instead. A major example is the list of children of an actor, which also stores the terminating/terminated state of the actor (class `akka.actor.dungeon.ChildrenContainer`. In Akka, it is a sealed hierarchy of immutable classes, containing immutable maps of child name to child actor ref. State change methods return new instances of classes in that hierarchy. In the Scala.js version, they become mutable classes, with a mutable JavaScript dictionary mapping child names to their references. A dictionary is simply a JavaScript object, and gets and sets are simple field selections and updates, which are basic oper-

[1]As per the HTML specification, timeout delays are raised to 4 ms starting from the fifth nested timer event. [3]

ations of JavaScript VMs. To avoid unnecessary allocations for leaf actors, the empty state is kept immutable so that it can be shared amond all leaf actors.

Other non-blocking algorithms have been redesigned similarly. Some low-level mechanisms even become completely unnecessary in a single threaded setup, e.g., the process of reserving a child name before it is actually created. These are removed, thereby simplifying some operations and making them more efficient.

3 Serialization, aka pickling

Arguably the most difficult challenge we had to face was that of serialization. As long as actors communicate with other actors in the same memory space (i.e., in the same Worker, in a Web setup), messages can simply be kept in memory and a pointer to them be given to the receiving actor. However, we also want to support remoting, in the forms of cross-worker communications, and also between client in Akka/JS and server in Akka/JVM. In that setup, messages crossing the memory space borders must be serialized.

Messages sent to another Web Worker are copied using an algorithm called *structured clone* [2]. In a nutshell, values copyable through that algorithm are a small superset of values that can be serialized to JSON. Messages sent through a WebSocket must be serializable as strings or blobs, effectively reducing that set to JSON-serializable. Since the additional supported values for structured clone do not buy anything in our setup, we designed a serialization mechanism whose format is a JSON-encodable JavaScript object. Such an encoding can be sent directly through Web Workers, or can be serialized to strings by the native call to `JSON.stringify()`.

On the JVM, serialization is typically achieved through *reflection* (or through the native JVM serialization mechanism, which essentially boils down to reflection as well). However, Scala.js does not support runtime reflection, for several reasons which are out of the scope of this report.

A promising approach for compile-time only serialization is that of the Scala Pickling project [8]. However, even Scala Pickling falls back on runtime reflection when the exact type of a value (or a finite and closed enumeration of subtypes) cannot be determined at compile time. As it turns out, in our use case, exact types can *never* be known at compile time, because messages are of type `Any`. We could imagine transferring implicit picklers all the way from the `!` method through the framework to the pickle point; but it would never work for the unpickling phase.

3.1 Explicit registration

In Scala.js, we need a mechanism that *never* falls back on runtime reflection. The only reflective operation we are able to perform is to get the runtime class of an object (`getClass()`) and the full name of that class (`getClass().getName()`).

Our solution is inspired by Scala Pickling, in that it is built on implicit picklers and unpicklers for the exact types of values that need to be pickled. The Scala.js pickling library obviously provides picklers and unpicklers for primitive data types and strings. It also provides automatic pickler and unpickler generation for case classes and case objects, using macros. However, picklers and unpicklers are not instantiated automatically at pickle and unpickle calls. Instead, they must be *registered* in advance to a `PicklerRegistry`. Registering types for which implicit picklers and unpicklers are in scope (which include any case class, thanks to implicit macros), is done as

```
picklerRegistry.register[SomeClass]
```

whereas registering a case object is done as

```
picklerRegistry.register(SomeCaseObject)
```

These calls must be done once for the pickler registry before that registry can be used to pickle values of these classes. Note that custom picklers and unpicklers can be defined by providing custom implicits for types, just like in Scala Pickling.

Having to register classes and objects in advance, explicitly, can be seen as an annoyance. Future work could remove this burden by providing automatic registration for certain classes, selected by well-chosen criteria. A simple criterium that would give good results would be: all case classes and case objects extending `java.io.Serializable`. The Scala.js compiler could help in providing automatic registration of picklers and unpicklers for the selected classes and objects.

3.2 Multiple formats

In anticipation of the client-server use case, the Scala.js pickling machinery must be available both on the client, compiled with Scala.js, and on the server, compiled with Scala. On the client, the pickling format should be primitive JavaScript values, objects and arrays. On the server, on the other hand, it should be a representation of JSON data in some JSON manipulation library.

To that effect, the core of Scala.js pickling is agnostic of a particular format of JSON representation (although it does assume JSON-like data). The definitions of the pickler and unpickler traits are parameterized with the pickle format `P`, and take implicit pickle builders and readers, respectively:

```
trait Pickler[A] {
  def pickle[P](obj: A)(
      implicit registry: PicklerRegistry,
      builder: PBuilder[P]): P
}
trait Unpickler[A] {
  def unpickle[P](pickle: P)(
      implicit registry: PicklerRegistry,
      reader: PReader[P]): A
}
```

where the pickle builder and reader traits are defined as:

```
trait PBuilder[P] {
  def makeNull(): P
  def makeBoolean(b: Boolean): P
  def makeNumber(x: Double): P
  def makeString(s: String): P
  def makeArray(elems: P*): P
  def makeObject(fields: (String, P)*): P
}

trait PReader[P] {
  def isUndefined(x: P): Boolean
  def isNull(x: P): Boolean
  def readBoolean(x: P): Boolean
  def readNumber(x: P): Double
  def readString(x: P): String
  def readArrayLength(x: P): Int
  def readArrayElem(x: P, i: Int): P
  def readObjectField(x: P, f: String): P
}
```

The API of builders and readers was designed assuming that the most direct way to access array elements and object fields is by random access rather than traversing. This is the case for JavaScript values in Scala.js, as well as the representation for arrays in the JSON library of the Play! framework. It is not true for objects in the JSON library of Play!, however, but we had to make a trade-off somewhere.

Using this abstraction, the core pickling library can be cross-compiled in Scala.js and Scala. It is also possible to write code that uses the core that are also generic in terms of the pickle format, and hence can also be cross-compiled. Most of the client-server communication layer is written that way, as we will see in Section 5.

4 Remoting across Web Workers

Web Workers [4] are a new technology, part of HTML 5, that allows scripts on Web pages to spawn truly parallel computations. Two flavors of workers exist: dedicated workers and shared workers. Since shared workers are still virtually non existent in current implementations, we covered only dedicated workers in this project, and will imply the dedicated flavor in this report unless explicitly stated otherwise.

A Web worker, child of the currently executing script, can be created with the following JavaScript call:

```
var worker = new Worker('worker-script.js');
```

This instructs the browser to spawn an entirely new JavaScript VM in a separate thread, and to execute the given script in the context of that VM. A Worker and its parent script have distinct memory spaces, i.e., they do not have any shared memory. The only means of communication between them is message passing, where messages are copied from one VM to the other using *structured cloning.* The parent script can post a message to its child with

```
worker.postMessage(msg);
```

whereas a child script can post a message to its parent with

```
<global scope>.postMessage(msg);
```

The messages are received as “message” *events* sent to the global scope of the child, and the `worker` object in the parent, respectively. It is only natural that we want to be able to talk to actors on a separate worker transparently through remote actor refs.

Akka/JS, just like Akka, organizes actors in a hierachical supervision structure with one root per actor system. Actors can be identified by their *path*, which contains two parts: the *address* of their owning actor system, and the path through the hierarchy that leads to the actor. Akka/JS has a restrictive notion of address: an address is relative to a Web page, and identifies one Worker spawned directly or indirectly by that Web page.[2] Hence, paths are also relative to Web page, and scripts cannot represent the path to an actor on another page, much less hold an ActorRef to such an actor.

Akka/JS represents ActorRefs pointing to "remote" actors, i.e., actors located on other Workers, as a special subclass of `ActorRef`: `WorkerActorRef`. Basically, a `WorkerActorRef` only stores the path to the corresponding actor. Every time a message is sent to that ref, the message is pickled with the infrastructure described in Section 3, routed to the Worker specified by the address, then unpickled and delivered to the receiving actor. The actual actor instance to deliver the message to is lookup up from the actor system's root down through the supervision hierarchy.

4.1 Pickling ActorRefs

The sender of a message, as well as any ActorRefs in sent messages, must be somehow pickled to be sent to another Worker. ActorRefs are not case classes, and they internally contain a reference to the mutable instance of the actor, so something must be done to pickle them. The pickler registry used by `WorkerActorRef` is overridden to give an appropriate treatment to subclasses of `ActorRef`: instead of trying to pickle the internals of the actor ref, only its path (containing the address) is pickled. When unpickling an actor ref under its path representation, the unpickler tests whether that path happens to point to an actor in the receiving system. If so, the corresponding `LocalActorRef` is looked up. Otherwise, a `WorkerActorRef` is created.

[2]Actually, it also identifies one specific actor system within that Worker, but we will omit this in the discussion for simplicity.

4.2 Routing messages across Workers

The attentive reader may have noticed that we have avoided explaining what an address really was, and how it identified a Worker. The Worker API we can work with only allows a worker to send messages to its parent and to its children, but never to grandparents, grandchildren, or siblings. However, when actor refs are pickled and sent through messages, it can very well happen that a Worker gets to know an actor located on an unreachable Worker.

To be able to send messages to "distant" Workers, we have built a generic *routing* infrastructure for Workers. Each Worker has a router (the singleton object `WebWorkerRouter`), which knows its address and provides an API to send arbitrary messages (not only actor messages) to any Worker given its address. To do so, routers build a hierarchical structure of all the Workers, and addresses are simply paths from the root (the Web page script) to a given Worker. When creating a child Worker, the current Worker's router assigns a unique name to this child, and uses it as its address. It sends that name and its own address in an initialization message to the child Worker so that its own router can derive its address within the hierarchy.

4.3 Bootstrapping the communication

We have silently assumed that we already hold an ActorRef to the actor we want to send a message to. This can happen by receiving its reference in a message or as sender. But how do we get an initial reference to an actor on another Worker, with which we can bootstrap the communication?

The Akka/JS actor system class provides a means to send a message to an arbitrary *path*, independent of holding an ActorRef. Sending a message to a path may fail because there is no actor running at that path, however. The concept is basically a simplified version of *actor selections* in Akka, and could be generalized to its full power in future work.

5 Communication between client and server

The really fun feature of Akka/JS is its layer for communication between a client running Akka/JS and a server running Akka/JVM. We have implemented this feature on top of WebSockets, but its design could be backported to less efficient (but more widely supported) technologies like Server-Sent Events and AJAX calls. Unlike the cross-Workers communication, the WebSockets communication does not have a dedicated subclass of `ActorRef`. Instead, this layer is built on top of existing classes of actor refs, both on the client and on the server. Actually, it is completely agnostic of the implementation details of both libraries: it lives entirely in *user-space* compared to them (by user-space, we mean it does not use any package-private classes or methods, only the truly public API). The main reason for this choice was that, since the implementation layer must live in both the Akka/JS world and the Akka/JVM world, which have slightly different implementations, it was best not to rely on implementation details.

Ignoring the connection establishment protocol for now, we will start by showing the pictures in steady state. The client and the server have symmetric roles in steady state, hence we will use side A and side B interchageably for one and the other. When a side A holds an actor ref to an actor on side B, the locally available actor ref is not really directly pointing at the remote actor. Instead, there is a local *proxy* actor on side A which is managed by the communication layer. The proxy holds an ID chosen by the side B for the proxied actor, which is opaque (unlike in the Worker setup where the corresponding data was the actor's path, which is not opaque). When a message is sent to this proxy on side A, the proxy's message handler pickles the message, sends it through the WebSocket connection, asking side B that it be delivered to the actor whose ID is stored in the proxy. On side B, an actor listens to messages on the WebSocket, and dispatches messages to local actors by looking up the ID in a dictionary it maintains. That actor is called the connection proxy.

The connection proxy is the only actor that can send messages to the WebSocket, and read messages from it. It also has a custom pickler registry which is able to pickle and unpickle actor refs. When an actor ref is first pickled on side A, the connection proxy chooses a new unique ID for it, stores the mapping from ID to local actor ref in a map, and sends that ID in the pickle, with a flag saying it is an actor ref on A's side. The receiving side B then creates a local proxy storing that ID, and also stores the mapping from ID to proxy (in another map). The local proxy is created as a child of B's connection proxy. The second time B receives a foreign actor ref with the same ID, it reuses the same proxy.

Returning to the general case of pickling an actor ref on side A: if a mapping for that actor ref already exists in the ID-to-local-proxy map, it means it is actually an actor on B's side, and the pickle contains the ID chosen by B. If it is present in the other map, then A had already chosen an ID for that actor ref, and its sends the same ID to B. Otherwise, a new ID is chosen and stored in the map.

5.1 Connection establishment

Now that we understand how the communication layer works in steady state, let us describe the connection establishment protocol. We follow the general rule that Web clients initiate connection to Web servers, and not the other way around. The server must offer a WebSocket entry point under a certain URL, that the client will be able to connect to. When a new connection arrives on that connection, the server needs to choose a new or existing actor to be the *entry point* for the connection. The connection establishment protocol will send an actor ref to that entry point to the client, so that it can start communicating with the server. Further actor refs can be exchanged through the messages and senders.

The protocol is the following:

1. On the client, an actor A wants to initiate a connection. It creates a new actor of class `ClientProxy`, specifying the WebSocket URL it wants to connect to.

2. The client proxy opens a WebSocket connection to the given URL.

3. Upon WebSocket connection creation on the server, it creates or chooses an actor E which

is going to be the entry point.

4. The server creates an actor of class `ServerProxy`, specifying the handle of the WebSocket connection as well as an actor ref to E.

5. The server proxy sends a `Welcome(E)` message to the client, which involves pickling the actor ref to E.

6. The client proxy receives that message, which involves unpickling the actor ref to E and hence creating a local proxy for E.

7. The client proxy replies to the initiating actor A with a `WebSocketConnected(LocalE)` message.

From there, the initiating actor A holds an actor ref to the local proxy of E, and can therefore send arbitrary messages to E. The steady state behavior is applied from there.

5.2 Death watch notifications

To support death watch notifications across the WebSocket, when pickling a local actor ref, the connection proxy on side A also starts watching that actor ref. When it receives a `Terminated` message for a local actor ref, it looks up its associated ID and notifies side B. Side B then simply stops its local proxy, which is a child of the connection proxy. This will in turn send `Terminated` messages to all the actors that watched the proxy, thereby correctly forwarding the death watch notification.

5.3 Connection loss

When the connection is broken, the connection proxies on both sides receive a notification from the underlying WebSocket implementation. They do not try to do anything fancy by themselves. Instead they simply stop, which also involves stopping all the local proxies because they are children of the connection proxy. This will correctly trigger any death watch notifications to be sent due to the connection loss.

The connection proxy also closes explicitly the WebSocket connection if it is stopped externally. Hence, the connection proxy is stopped if and only if the WebSocket connection is closed or broken, which is an event the application can listen to by watching the connection proxy. It can then take appropriate action depending on the use case, which might involve trying to reopen the connection.

This strategy is in accordance with Akka's philosophy of "Let it crash".

5.4 Security concerns

When a client and directly send messages to a server's actors, it becomes necessary to think about security issues. The server certainly does not want an arbitrary client to send arbitrary messages to arbitrary actors, since the client can easily be impersonated. It is therefore important to think about security concerns.

Our approach to security is the following: a side B can only ever talk to an actor x on side A if A send an actor ref to x in a message it sent to B (including as sender information). Actor refs to actors not sent explicitly to B cannot be forged by B, nor by any other side C. In particular, side C cannot forge an actor ref to an actor on side A that was sent to side B but not to side C.

The application of these safety rules directly follows from the opaque ID management. Since IDs are "scoped" by connection, and only actor refs sent to B are ever assigned an ID in the "scope" of connection B, B can never forge an actor ref that was not sent to it.

6 Evaluation and conclusion

We evaluated our design and implementation using two focused tests and one larger application. The two focused tests test the Web Worker communication and the fault handling features of Akka, respectively. The latter is in fact an import from the Akka documentation.

The larger application puts everything together in a Chat implemented in Play+Akka on the server and Akka/JS on the client. It exercises heavily the client-server communication layer, as well as other small features: receive timeouts, `become()`, the Ask pattern. It provides auto-reconnect based on top of death watch notifications.

The chat features multiple rooms and private chats between 2 persons. In the implementation of the private chat, client communicate "directly" between each other, because they end up holding actor refs which represents (through 2 local proxies) an actor on the other client! This is all transparent.

Although some features of Akka core have not yet been ported, like the event stream, we believe Akka/JS is a very successful prototype. Akka/JS helps a lot in dealing with failures of the connection and the auto-reconnect mechanism, because its philosophy assumes things will crash and it provides effective ways of managing failures, rather than avoiding failure handling. It also provides an easy way to leverage Web Workers, and hence multi-core processors in Web pages needing heavy computations.

References

[1] Akka. URL: `http://akka.io/`

[2] HTML Structured Clone Algorithm. URL: `http://www.w3.org/TR/html5/infrastructure.html#safe-passing-of-structured-data`

[3] HTML Timers Specification. URL: `http://www.whatwg.org/specs/web-apps/current-work/multipage/timers.html`

[4] HTML Web Workers Specification. URL: `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html`

[5] Promises/A+ Specification. URL: `http://promises-aplus.github.io/promises-spec/`

[6] Scala Futures and Promises. URL: `http://docs.scala-lang.org/overviews/core/futures.html`

[7] Scala.js, a Scala to JavaScript compiler. URL: `http://www.scala-js.org/`

[8] Miller H., Haller P., Burmako E. and Odersky M.: Instant Pickles: Generating Object-oriented Pickler Combinators for Fast and Extensible Serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 183-202 (2013)