On the Integration of the Actor Model into Mainstream Technologies

A Scala Perspective

Philipp Haller

Typesafe, Inc.

philipp.haller@typesafe.com

Abstract

Integrating the actor model into mainstream software platforms is challenging because typical runtime environments, such as the Java Virtual Machine, have been designed for very different concurrency models. Moreover, to enable integration with existing infrastructures, execution modes and constructs foreign to the pure actor model have to be supported. This paper provides an overview of past and current efforts to address these challenges in the context of the Scala programming language.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming— Distributed and parallel programming; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

Keywords Concurrent programming, distributed programming, actors, threads

1. Introduction

Actors are a powerful abstraction for structuring highly concurrent software systems which scale up to many-core processors, as well as scale out to clusters and the cloud. The Scala community is well-known for its effort to bring the actor model to mainstream software engineering. The first actors implementation was released as part of the Scala standard library in 2006 [6, 15]. Since then, there has been a steady stream of both research results and industrial development, contributing to a renewed interest in actors in academia, as well as innovations powering state-of-the-art frameworks like the Akka event-driven middleware [16].

Integrating the actor model into mainstream software platforms is a formidable challenge. On the one hand, industrial-strength implementations have to make optimal use of underlying runtime environments which typically have not been designed to support actors. On the other hand, in order to integrate with existing infrastructures, it is necessary to support execution modes and constructs that are rather foreign to a pure notion of actors, such as blocking operations, and interoperability with native platform threads.

This paper provides an overview of the challenges of providing an industrial-strength actor implementation on the Java Virtual Machine (JVM), in the context of the Scala programming language [13]. It aims to serve as an experience report on addressing these challenges through a combination of research and engineering advances.

We're going to focus on the two main actor implementations for Scala: Scala Actors [7] and Akka [16]. The former has been part of Scala's standard library since Scala version 2.1.7. Beginning with Scala version 2.10.0, Scala Actors are deprecated in favor of Akka's actor implementation, a new member of the Scala distribution. One goal of this paper is to motivate this transition, and to examine which ideas of Scala Actors are adopted in Akka, and what has changed in the design and implementation.

AGERE! October 21-22, 2012.

1.1 Overview

The design and implementation of industrial-strength actor implementations on mainstream platforms, such as the JVM, is driven by numerous requirements. Some requirements guided the design and implementation of Scala's first actors framework; these "early requirements" were:

- *Library-based implementation (R1).* It is unclear which concurrency abstraction is going to "win". Real-world concurrency tasks might even benefit from a combination of several different abstractions. Rather than betting on a single candidate and providing built-in language support, Scala's approach has been to enable flexible concurrency *libraries*.
- *High-level domain-specific language (R2).* While actors are provided using library abstractions in Scala, it is important that their programming interface is "competitive" with languages with specialized concurrency support.
- *Event-driven implementation (R3).* Actors should be mapped to lightweight tasks triggered by messaging events. Spending an entire virtual machine thread per actor does not scale to large numbers of actors. At the same time, the benefits of thread-based programming should remain accessible in cases where a purely event-driven model would be too restrictive.

In retrospect, these requirements are still valid, however, other requirements turned out to be more important in the context of industrial software development. With the growing use of actors in production Scala applications, it became clear that satisfying *only* these early requirements was not sufficient to meet all demands. Other requirements had to be added, and existing ones turned out to be useful beyond their initial goals. These "later requirements" were:

- *High performance (R4).* The majority of industrial applications where actors provide most benefits are highly performance sensitive. Past experience with industrial Erlang applications [1, 12] suggests that *scalability* is more important than raw performance. On the other hand, it is known that a high-performance virtual machine such as the JVM can enable applications to scale out much more gracefully than other runtime environments. In addition, benchmarking offers a simple evaluation strategy if the compared benchmark programs are of similar complexity.

- *Flexible remote actors (R5).* Many medium-sized and large applications can benefit from remote actors, i.e., actors that communicate transparently over the network. In many cases, flexible deployment mechanisms, e.g., using external configuration, are very important.

The early requirement of a library-based implementation turned out to provide additional benefits: first, it enables existing tools, such as IDEs and debuggers, to be readily supported. Second, it is possible to provide APIs for several languages. For example, the Akka framework has both a Scala and a Java API.

In the following we are going to "tackle" these requirements, in two groups: the first group is concerned with the *programming interface* (see Section 2) which addresses requirements R1 and R2. The second group is concerned with the *actor runtime* (see Section 3) which addresses requirements R3 and R4. Remote actors (R5) are beyond the scope of this paper.

2. The Programming Interface

This section provides an overview of the programming interface of both Akka and Scala Actors, and how the interface is realized as a library in Scala.

An actor is a process that communicates with other actors by exchanging messages. The principal message send operation is asynchronous. Therefore, an actor buffers incoming messages in a message queue, its *mailbox*. The *behavior* of an actor determines how the messages in its mailbox are processed. Since defining an actor's behavior is a rather important activity when programming with actors, it is crucial that an actor programming system has good support for it. In Scala, the behavior of an actor can be defined by creating a new class type that extends a predefined `Actor` trait.[1] Figure 1 shows an example using Scala Actors (top) and Akka Actors (bottom), respectively.

In Scala Actors, the body of the `act` method (inherited from `Actor`) defines an actor's behavior. In the above example, it repeatedly calls the `receive` operation to try to receive a message. The receive operation has the following form:

```
receive {
  case msgpat_1 => action_1
  ...
  case msgpat_n => action_n
}
```

The first message which matches any of the patterns $msgpat_i$ is removed from the mailbox, and the corresponding $action_i$ is executed. If no pattern matches, the actor suspends.

The example in Figure 1 (top) uses `receive` to wait for two kinds of messages. The `Order(item)` message handles an order for `item`. An object which represents the order is created and an acknowledgment containing a reference to the order object is sent back to the sender. The `Cancel(o)` message cancels order `o` if it is still pending. In this case, an acknowledgment is sent back to the sender. Otherwise a `NoAck` message is sent, signaling the cancellation of a non-pending order.

The API of Akka's actors is similar to that of Scala Actors. The principal way of defining a message handler for incoming messages is the implementation of the `receive` method, which is inherited from the `Actor` trait. The body of the `receive` method has the same form as in the case of `receive` in Scala Actors.

For simplicity, we're going to refer to the latter as "sreceive" and to the former as "areceive" (Akka's `receive`) in the following. The main difference between "sreceive" and "areceive" is that the former operation is *blocking*, i.e., the current actor is suspended until a matching message can be removed from its mailbox. On the other hand, "areceive" is used to define a global message handler, which, by default, is used for processing all messages that the actor receives over the course of its life time. Moreover, the message handler defined by "areceive" only gets activated when a message can be removed from the mailbox. Another important difference is that "areceive" will never leave a message in the mailbox if there is no matching pattern which is different compared to "sreceive". Whenever the actor is ready to process the next message, it is removed from the mailbox; if there is no pattern that matches the removed message, an event is published to the system, signaling an unhandled message.

The example in Figure 1 (bottom) defines a global message handler which handles the same two kinds of messages as the example at the top.

[1] A trait in Scala is an abstract class that can be mixin-composed with other traits.

```
class OrderManager extends Actor {
  def act() {
    while (true) {
      receive {
        case Order(item) =>
          val o = handleOrder(sender, item)
          sender ! Ack(o)
        case Cancel(o) =>
          if (o.pending) {
            cancelOrder(o)
            sender ! Ack(o)
          } else sender ! NoAck
      }
    }
  }
}
```

```
class OrderManager extends Actor {
  def receive = {
    case Order(item) =>
      // same as above
    case Cancel(o) =>
      // same as above
  }
}
```

Figure 1. Example: orders and cancellations.

2.1 Bridging the Gap

The semantics of "sreceive" and "areceive" are quite different. "sreceive" has the same semantics as "receive" in Erlang [2]. On the other hand, "areceive" can be implemented more efficiently on the JVM. Each construct enables a different programming style for messaging protocols. To support both styles, Akka 2.0 introduces a `Stash` trait which an `Actor` subclass can optionally mix in. Together with methods to change the global message handler of an actor (called `become` and `unbecome` in Akka), the stash enables the familiar Erlang style also using Akka.

2.2 Creating Actors

In Scala Actors, creating a new instance of a subclass of `Actor` (such as `OrderManager` in Figure 1) creates an actor with the behavior defined by that class. All interaction with the actor (message sends etc.) is done using references to that instance.

In Akka Actors, an actor is created using one of several factory methods of an instance of type `ActorRefFactory`, say `factory`:

```
val actor = factory.actorOf(Props[OrderManager])
```

In many cases, the `factory` object is the "actor system", the container which provides shared facilities (e.g., task scheduling) to all actors created in that container. (The `factory` can also be a "context" object which is used to create supervision hierarchies for fault handling.) The expression `Props[OrderManager]` in Scala is equivalent to `Props.apply[OrderManager]`, an invocation of the `apply` factory method of the `Props` singleton object. Singleton objects have exactly one instance at runtime, and their methods are similar to static methods in Java. The `Props.apply` method returns an instance of the `Props` class type, which contains all information necessary for creating new actors.

The main difference between creating an actor in Scala Actors and in Akka is that the above `actorOf` method in Akka returns an instance of type `ActorRef` instead of an instance of the specific `Actor` subclass. One of the main reasons is *encapsulation*.

2.3 Encapsulation

The actor runtime guarantees thread safety of actor interactions only if actors communicate only by passing messages. However, in Scala Actors it is possible for an actor to directly call a (public) method on a different actor instance. This breaks encapsulation and can lead to race conditions if the state of the target actor is accessed concurrently [10].

To prevent such encapsulation breaches, in Akka actors have a very limited interface, `ActorRef`, which basically only provides methods to send or forward messages to its actor. Akka has built-in checks to ensure that no direct reference to an instance of an `Actor` subclass is accessible after an actor is created. This mechanism works surprisingly well in practice, although it can be circumvented. [9]

An alternative approach to ensuring encapsulation of actors is a typing discipline such as uniqueness types [3]. The capability-based *separate uniqueness* type system [8] has been implemented as a prototype for Scala [5]. However, more research needs to be done to make such type systems practical.

2.4 Implementation

Looking at the examples shown above, it might seem that Scala is a language specialized for actor concurrency. In fact, this is not true. Scala only assumes the basic thread model of the underlying host. All higher-level operations shown in the examples are defined as classes and methods of the Scala library. In the rest of this section, we look "under the covers" to find out how selected constructs are defined and implemented.

The send operation `!` is used to send a message to an actor. The syntax `a ! msg` is simply an abbreviation for the method call `a.!(msg)`, just like `x + y` in Scala is an abbreviation for `x.+(y)`. Consequently, `!` can be defined as a regular method:

```
  def !(msg: Any): Unit = ...
```

The `receive` constructs are more interesting. In Scala, the pattern matching expression inside braces is treated as a first-class object that is passed as an argument to "sreceive", and returned from "areceive", respectively. The argument's type is an instance of `PartialFunction`, which is a subtrait of `Function1`, the type of unary functions. The two traits are defined as follows.

```
trait Function1[-A, +B] {
  def apply(x: A): B
}
trait PartialFunction[-A, +B]
  extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
}
```

Functions are objects which have an `apply` method. Partial functions are objects which have in addition a method `isDefinedAt` which tests whether a function is defined for a given argument. Both traits are parameterized; the first type parameter `A` indicates the function's argument type and the second type parameter `B` indicates its result type[2].

A pattern matching expression
`{ case` p_1 `=>` e_1`; ...; case` p_n `=>` e_n `}` is then a partial function whose methods are defined as follows.

- The `isDefinedAt` method returns `true` if one of the patterns p_i matches the argument, `false` otherwise.
- The `apply` method returns the value e_i for the first pattern p_i that matches its argument. If none of the patterns match, a `MatchError` exception is thrown.

The two methods are used in the implementation of "sreceive" as follows. First, messages in the mailbox are scanned in the order they appear. If the argument `f` of "sreceive" is defined for a message, that message is removed from the mailbox and `f` is applied to it. On the other hand, if `f.isDefinedAt(m)` is `false` for every message `m` in the mailbox, the receiving actor is suspended.

The Akka runtime uses partial functions differently: first, the behavior of an actor is defined by implementing the `receive` method; this method *returns* a partial function, say, `f`. The messages in the actor's mailbox are processed in FIFO order. The Akka runtime guarantees that at most one message (per receiving actor) is processed at a time. Each message, say `msg` is removed from the mailbox regardless of `f`. If `f` is defined for `msg`, `f` is applied to it. On the other hand, if `f.isDefinedAt(msg)` is `false`, `msg` is published as an "unhandled message" event to the system (wrapped in an object which additionally contains references to the sender and receiver).

3. The Actor Runtime

As motivated in the introduction, the most important features of the actor runtime are (a) a lightweight execution environment, and (b) high performance. In the following we will outline how these features are realized in Akka and which ideas of Scala Actors stood the test of time.

3.1 Event-Based Actors

Scala Actors [6] introduced a new approach to decouple actors and threads by providing an event-based operation for receiving messages, called "react". In this approach, an actor waiting for a message that it can process is not modeled by blocking a thread; instead, it is modeled by a closure which is set up to be scheduled for execution when a suitable message is received. At that point a *task* is created which executes this continuation closure, and submitted to a thread pool for (asynchronous) execution.

In this approach, the continuation closure is actually an instance of type `PartialFunction[Any, Unit]` (see Section 2.4). Akka has adopted this idea: the continuation of an actor waiting for a message is an instance of the same type. The main difference is that when using "react", this continuation closure is provided *per message* to be received; in contrast, in Akka the continuation closure is defined once, to be used for many (or all) messages. Additionally, Akka provides methods to change the global continuation (`become`/`unbecome`). The main advantage of Akka's approach is that it can be implemented much more efficiently on the JVM. In

[2] Parameters can carry `+` or `-` variance annotations which specify the relationship between instantiation and subtyping. The `-A, +B` annotations indicate that functions are contravariant in their argument and covariant in their result. In other words `Function1[X1, Y1]` is a subtype of `Function1[X2, Y2]` if `X2` is a subtype of `X1` and `Y1` is a subtype of `Y2`.

the absence of first-class continuations, implementing "react" requires the use of *control-flow exceptions* to unwind the call stack, so that each message is handled on a call stack which is basically empty. Throwing and catching a control exception for each message is additional overhead compared to Akka's execution strategy.

3.2 Lightweight Execution Environment

A key realization of Scala Actors is the fact that for actor programs a workstealing thread pool with local task queues [11] scales much better than a thread pool with a global task queue. The main idea is as follows: when creating a task which executes the message handler to process a message, that task is submitted to the *local task queue* of the current worker thread. This avoids an important bottleneck of thread pools with a global task submission queue which can quickly become heavily contended.

Like Scala Actors, Akka uses Lea's fork/join pool (an evolution of [11], released as part of JDK 7 [14]). In addition, and unlike Scala Actors, Akka uses non-blocking algorithms for inserting messages into actor mailboxes, and scheduling tasks for execution, which results in a substantial performance boost.

3.3 Integrating Threads and Actors

Integrating (JVM) threads and event-based actors is useful to enable powerful message-processing operations also for regular threads. This facilitates interoperability with existing libraries and frameworks and offers additional convenience, since it enables actors to be more easily used from Scala's interactive REPL (read-eval-print-loop). Besides Scala Actors, there are other approaches attempting an integration of threads and event-based actors [4].

In Scala Actors, calling `receive` on a regular thread, which is not currently executing an actor, establishes an actor identity and mailbox in thread-local storage. This actor identity can be passed to other actors in messages, so as to add the thread actor to their set of acquaintances.

Akka version 2.1 introduces an `Inbox` abstraction which let's one create a first-class actor mailbox as follows:

```
implicit val i = ActorDSL.inbox()
someActor ! someMsg // replies will go to ‘i‘

val reply = i.receive()
val transformedReply = i.select(5 seconds) {
  case x: Int => 2 * x
}
```

The message send in the second line above *implicitly* transmits an `ActorRef` obtained from the `Inbox i` as the sender of `someMsg`. As a result, responses of the receiving actor (via `sender ! someResponse`) are enqueued in `i`. Methods such as `receive` and `select` enable blocking access to one message at a time. The downside of a first-class mailbox is, of course, that it does not come with a guarantee that there is only a single thread receiving from the same mailbox, since it could be shared among multiple threads. On the other hand, the advantage is that it allows an efficient implementation, and it is relatively straight-forward to avoid subtle memory leaks.

3.4 Summary

- Scala's partial functions are well-suited to represent an actor's continuation.
- The overhead of unwinding the call stack through exceptions can be avoided by using a single, global message message handler. Loss in flexibility can be recovered through constructs to replace the global message handler.
- A workstealing thread pool with local task queues is an ideal execution environment for event-based actors.
- Threads and actors can be integrated in a robust way using first-class actor mailboxes. On the other hand, it does not guarantee unique receivers.

4. Conclusion

In this paper we have outlined the requirements and forces of mainstream software engineering which have influenced past and present actor implementations for Scala. Based on these requirements, principles behind the design and implementation of actors in Scala are explained, covering (a) the programming interface, and (b) the actor runtime. It is our hope that the learned lessons will be helpful in the design of other actor implementations for platforms sharing at least some features with Scala and/or the Java Virtual Machine.

References

[1] J. Armstrong. Erlang — a survey of the language and its industrial applications. In *Proc. INAP*, pages 16–18, Oct. 1996.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[3] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS'08)*, pages 139–154. Springer, Dec. 2008.

[4] T. V. Cutsem, S. Mostinckx, and W. D. Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems & Structures*, 35(1):80–98, 2009.

[5] P. Haller. *Isolated Actors for Race-Free Concurrent Programming*. PhD thesis, EPFL, Switzerland, Nov. 2010.

[6] P. Haller and M. Odersky. Event-based programming without inversion of control. In D. E. Lightfoot and C. A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006. ISBN 3-540-40927-0.

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

[8] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, pages 354–378. Springer, June 2010. ISBN 978-3-642-14106-5.

[9] P. Haller and F. Sommers. *Actors in Scala*. Artima Press, 2012.

[10] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ'09)*, pages 11–20. ACM, Aug. 2009. ISBN 978-1-60558-598-7.

[11] D. Lea. A Java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[12] J. H. Nyström, P. W. Trinder, and D. J. King. Evaluating distributed functional languages for telecommunications software. In *Proc. Workshop on Erlang*, pages 1–7. ACM, Aug. 2003. ISBN 1-58113-772-9.

[13] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala, Second Edition*. Artima Press, 2010.

[14] Oracle, Inc. Java SE Development Kit 7. `http://openjdk.java.net/`.

[15] The Scala Programming Language. Home page. `http://www.scala-lang.org/`.

[16] Typesafe, Inc. Akka framework. `http://www.akka.io`.