Concurrency in Erlang and Scala

Ruben Vermeersch
Katholieke Universiteit Leuven

11th January 2009

1 Introduction

The continuous increase of computer processor clock rate has recently slowed down, in part due to problems with heat dissipation. As the clock rate of a processor is increased, the heat generated increases too. Increasing current clock rates even further is troublesome, as the heat generated would push computer chips towards the limits of what is physically possible. Instead, processor manufacturers have turned towards multi-core processors, which are processors capable of doing multiple calculations in parallel. There has been an increased interest in software engineering techniques to fully utilize the capabilities offered by these processors.

At the same time, applications are more frequently built in a distributed way. In order to perform efficiently, without wasting most of the time on waiting for I/O operations, applications are forced to do more and more operations concurrently, to obtain maximum efficiency.

However, reasoning about concurrent systems can be far from trivial, as there can be a large number of interacting processes, with no predefined execution order. This paper discusses the concurrency model in Erlang and Scala, two languages that have recently gained in popularity, in part due to their support for scalable concurrency. This first section introduces both languages, by describing their key features.

1.1 Erlang

Erlang [1] is a concurrent programming language, originally designed by Ericsson (Erlang is named after A. K. Erlang, a Danish mathematician and Ericsson: **Er**icsson **Lang**uage). It was designed to be distributed and fault-tolerant, for use in highly-available (non-stop) soft real-time telecom applications.

Erlang is a pure functional language, it features single assignment and eager evaluation. It also has built-in language constructs for distribution and concurrency. Erlang has a dynamic type system, where the typing of expressions is checked at run-time [5]. Typing information is fully optional and only used by the static type-checker, the run-time environment even allows running applications with invalid typing specifications. Run-time type checking is always performed by looking at the types of the data itself. This usually allows applications to run correctly, even when encountering unexpected data types.

Listings 1 and 2 show a simple program, written in Erlang and its execution in the Erlang shell. This program performs a recursive calculation of the factorial function.

```
1  -module(factorial).
2  -export([fact/1]).
3
4  fact(0) -> 1;
5  fact(N) -> N * fact(N-1).
```

Listing 1: A simple program in Erlang

```
1  $ erl
2  Erlang (BEAM) emulator version 5.6.3 [source] [async-threads:0]
3                                       [hipe] [kernel-poll:false]
4
5  Eshell V5.6.3  (abort with ^G)
6  1> c(factorial).
7  {ok,factorial}
8  2> factorial:fact(8).
9  40320
10 3>
```

Listing 2: Using the Erlang program

To allow for non-stop application, Erlang support hot-swapping of code. This means that it is possible to replace code while executing a program, making it possible to do upgrades and maintenance without interrupting the running system.

1.2 Scala

Scala [3], which stands for Scalable Language, is a programming language that aims to provide both object-oriented and functional programming styles, while staying compatible with the Java Virtual Machine (JVM). It was designed in 2001 by Martin Odersky and his group at EPFL in Lausanne, Switzerland, by combining the experiences gathered from designing multiple other languages.

The Scala language has been designed to be *scalable* [11], as it is supposed to scale along with the needs of its users. Offering functional constructs allows the developer to write short and concise code, whereas having object-oriented concepts allows the language to be used for large complex projects. Scala is fully interoperable with the Java language, which allows developers to use all Java libraries from within Scala. As such, it is not a separate language community, but rather allows you to take advantage of the enormous Java ecosystem. There is also an initiative to build a Scala version that runs on top of the .NET Common Language Runtime (CLR) [2], this ensures the portability of Scala to other underlying platforms.

Scala tries to stay to stay close to pure object-oriented programming and therefore does not have constructs like static fields and methods. Every value in Scala is an object and every operation is a method call [12]. Scala allows you to define your own operators and language constructs. This makes it possible to extend the Scala language according to your specific needs, which again helps it to grow (scale up) along with the project.

Scala uses strict-typing, yet allows most of the typing to be unspecified. When type information is not specified, the compiler will do smart type inference and attempt to infer this information from the code itself. This save programming effort and allows for more generic code. Type information is only required when the compiler cannot prove that correct type usage will happen.

```
1 object TestFactorial extends Application {
2     def fact(n: Int): Int = {
3         if (n == 0) 1
4         else n * fact(n-1)
5     }
6
7     println("The factorial of 8 is "+fact(8))
8     // Output: The factorial of 8 is 40320
9 }
```

Listing 3: A simple program in Scala

Listing 3 shows a simple program in Scala, which shows some of its functional features, as well as some object-oriented features. As can be seen, an object with a method is defined. The definition of this method however is done in a pure functional style. There is no need to write a return statement, as the value of the function is considered to be the return statement.

2 A different way of concurrency: The Actor Model

2.1 The problem with threads

The traditional way of offering concurrency in a programming language is by using threads. In this model, the execution of the program is split up into concurrently running tasks. It is as if the program is being executed multiple times, the difference being that each of these copies operated on shared memory.

This can lead to a series of hard to debug problems, as can be seen below. The first problem, on the left, is the *lost-update* problem. Suppose two processes try to increment the value of a shared object `acc`. They both retrieve the value of the object, increment the value and store it back into the shared object. As these operations are not atomic, it is possible that their execution gets interleaved, leading to an incorrectly updated value of `acc`, as shown in the example.

The solution to this problems is the use of locks. Locks provide mutual exclusion, meaning that only one process can acquire the lock at the same time. By using a locking protocol, making sure the right locks are acquired before using an object, lost-update problems are avoided. However, locks have their own share of problems. One of them is the *deadlock* problem, which is pictured on the right. In this example two processes try to acquire the same two locks `A` and `B`. When both do so, but in a different order, a deadlock occurs. Both wait on the other to release the lock, which will never happen.

These are just some of the problems that might occur when attempting to use threads and locks.

Lost Update Problem

Process 1	Process 2
`a = acc.get()`	
`a = a + 100`	`b = acc.get()`
	`b = b + 50`
	`acc.set(b)`
`acc.set(a)`	

Deadlock Problem

Process 1	Process 2
`lock(A)`	`lock(B)`
`lock(B)`	`lock(A)`
. . . Deadlock! . . .	

Both Erlang and Scala take a different approach to concurrency: the *Actor Model*. It is necessary to look at the concepts of the actor model first, before studying the peculiarities of the languages itself.

2.2 The Actor Model

The Actor Model, which was first proposed by Carl Hewitt in 1973 [9] and was improved, among others, by Gul Agha [4]. This model takes a different approach to concurrency, which should avoid the problems caused by threading and locking.

In the actor model, each object is an actor. This is an entity that has a mailbox and a behaviour. Messages can be exchanged between actors, which will be buffered in the mailbox. Upon receiving a message, the behaviour of the actor is executed, upon which the actor can: send a number of messages to other actors, create a number of actors and assume new behaviour for the next message to be received.

Of importance in this model is that all communications are performed asynchronously. This implies that the sender does not wait for a message to be received upon sending it, it immediately continues its execution. There are no guarantees in which order messages will be received by the recipient, but they will eventually be delivered.

A second important property is that all communications happen by means of messages: there is no shared state between actors. If an actor wishes to obtain information about the internal state of another actor, it

will have to use messages to request this information. This allows actors to control access to their state, avoiding problems like the lost-update problem. Manipulation of the internal state also happens through messages.

Each actor runs concurrently with other actors: it can be seen as a small independently running process.

3 Actors in Erlang

In Erlang, which is designed for concurrency, distribution and scalability, actors are part of the language itself. Due to its roots in the telecom industry, where a very large amount of concurrent processes are normal, it is almost impossible to think of Erlang without actors, which are also used to provide distribution. Actors in Erlang are called processes and are started using the built-in `spawn` function.

A simple application that uses an actor can be seen in listing 4. In this application, an actor is defined which acts as a basic counter. We send 100.000 increment messages to the actor and then request it to print its internal value.

Lines 1 & 2 defines the module and the exported functions. Lines 4 till 7 contain the `run` function, which starts a counter process and starts sending increment messages. Sending these messages happens in lines 15 till 18, using the message-passing operator (`!`). As Erlang is a purely functional language, it has no loop structures. Therefore, this has to be expressed using recursion. These extremely deep recursion stacks would lead to stack overflows in Java, yet Erlang is optimized for these usage patterns. The increment message in this example also carries a parameter, to show Erlangs parameter capabilities. The state of the counter is also maintained using recursion: upon receiving an `inc` message, the counter calls itself with the new value which causes it to receive the next message. If no messages are available yet, the counter will block and wait for the next message.

```
1  -module(counter).
2  -export([run/0, counter/1]).
3
4  run() ->
5      S = spawn(counter, counter, [0]),
6      send_msgs(S, 100000),
7      S.
8
9  counter(Sum) ->
10     receive
11         value -> io:fwrite("Value is ~w~n", [Sum]);
12         {inc, Amount} -> counter(Sum+Amount)
13     end.
14
15 send_msgs(_, 0) -> true;
16 send_msgs(S, Count) ->
17     S ! {inc, 1},
18     send_msgs(S, Count-1).
19
20 % Usage:
21 %     1> c(counter).
22 %     2> S = counter:run().
23 %         ... Wait a bit until all children have run ...
24 %     3> S ! value.
25 %     Value is 100000
```

Listing 4: Actors in Erlang

3.1 Actor scheduling in Erlang

Erlang uses a preemptive scheduler for the scheduling of processes [14]. When they have executed for a too long period of time (usually measured in the amount of methods invoked or the amount of CPU-cycles used), or when they enter a `receive` statement with no messages available, the process is halted and placed on a scheduling queue.

This allows for a large number of processes to run, with a certain amount of fairness. Long running computations will not cause other processes to become unresponsive.

Starting with release R11B, which appeared in May 2006, the Erlang run-time environment has support for symmetric multiprocessing (SMP) [6]. This means that it is able to schedule processes in parallel on multiple CPUs, allowing it to take advantage of multi-core processors. The functional nature of Erlang allows for easy parallelization. An Erlang lightweight process (actor) will never run in parallel on multiple processors, but using a multi-threaded run-time allows multiple processes to run at the same time. Big performance speedups have been observed using this technique.

4 Actors in Scala

Actors in Scala are available through the `scala.actors` library. Their implementation is a great testament for the expressiveness of Scala: all functionality, operators and other language constructs included, is implemented in pure Scala, as a library, without requiring changes to Scala itself.

The same sample application, this time written in Scala can be seen in listing 5.

We can see the following code: Lines 1 & 2 import the abstract `Actor` class[1] and its members (we need the `!` operator for sending messages). Lines 4 & 5 define the `Inc` and `Value` case classes, which will be used as message identifiers. The increment message has a parameter, as an example to demonstrate this ability.

Lines 7 till 21 define the `Counter` actor, as a subclass of `Actor`. The `act()` method is overridden, which provides the behavior of the actor (lines 10-20). This version of the counter actor is written using a more object-oriented style (though Scala fully supports the pure functional way, as shown in the Erlang example too). The state of the actor is maintained in an integer field counter. In the `act()` method, an endless receive loop is executed. This block processes any incoming messages, either by updating the internal state, or by printing its value and exiting.

Finally, on lines 23 till 32, we find the main application, which first constructs a counter, then sends 100.000 `Inc` messages and finally send it the `Value` message. The `!` operator is used to send a message to an actor, a notation that was borrowed from Erlang.

```
1  import scala.actors.Actor
2  import scala.actors.Actor._
3
4  case class Inc(amount: Int)
5  case class Value
6
7  class Counter extends Actor {
8      var counter: Int = 0;
9
10     def act() = {
11         while (true) {
```

[1] Which is actually a *trait*, a special composition mechanism used by Scala, but this is out of the scope of this paper.

```
12              receive {
13                  case Inc(amount) =>
14                      counter += amount
15                  case Value =>
16                      println("Value is "+counter)
17                      exit()
18              }
19          }
20      }
21  }
22
23  object ActorTest extends Application {
24      val counter = new Counter
25      counter.start()
26
27      for (i <- 0 until 100000) {
28          counter ! Inc(1)
29      }
30      counter ! Value
31      // Output: Value is 100000
32  }
```

Listing 5: Actors in Scala

The output of this program shows that the counter has been incremented up to a value of 100.000. This means that in this case all our messages were delivered in order. This might not always be the case: recall that there are no guarantees on the order of message delivery in the actor model.

The example above shows the ease with which actors can be used in Scala, even though they are not part of the language itself.

It also shows the similarities between the actors library in Scala and the Erlang language constructs. This is no coincidence, as the Scala actors library was heavily inspired by Erlang. The Scala developers have however expanded upon Erlangs concepts and added a number of features, which will be highlighted in the following sections.

4.1 Replies

The authors of `scala.actors` noticed a recurring pattern in the usage of actors: a request/reply pattern [8]. This pattern is illustrated in listing 6. Often, a message is sent to an actor and in that message, the sender is passed along. This allows the receiving actor to reply to the message.

To facilitate this, a `reply()` construct was added. This removes the need to send the sender along in the message and provides easy syntax for replying.

```
1  receive {
2      case Msg(sender, value) =>
3          val r = process(value)
4          sender ! Response(r)
5  }
```

Listing 6: Normal request/reply

```
1  receive {
2      case Msg(value) =>
3          val r = process(value)
4          reply(Response(r))
5  }
```

Listing 7: Request/reply using `reply()`

A `reply()` construct is not present in Erlang, where you are forced to include the sender each time you want to be able to receive replies. This is not a bad thing however: Scala messages always carry the

identity of the sender with them to enable this functionality. This causes a tiny bit of extra overhead, which might be too much in performance critical applications.

4.2 Synchronous messaging

Scala contains a construct which can be used to wait for message replies. This allows for synchronous invocation, which is more like method invocation. The syntax to do this is shown in listing 9, contrasted by the normal way of writing this in listing 8.

When entering the `receive` block, or upon using the `!?` operator, the actor waits until it receives a message matched by any of the `case` clauses. When the actor receives a message that is not matched, it will stay in the mailbox of the actor and retried when a new `receive` block is entered.

```
1 myService ! Msg(value)
2 receive {
3     case Response(r) => // ...
4 }
```

Listing 8: Waiting for a reply

```
1 myService !? Msg(value) match {
2     case Response(r) => // ...
3 }
```

Listing 9: Synchronous invocation using `!?`

4.3 Channels

Messages, as used in the previous examples, are used in a somewhat loosely typed fashion. It is possible to send any kind of message to an actor. Messages that are not matched by any of the `case` clauses will remain in the mailbox, rather than causing an error.

Scala has a very rich type system and the Scala developers wanted to take advantage of this. Therefore they added the concept of channels [8]. These allow you to specify the type of messages that can be accepted using generics. This enables type-safe communication.

The mailbox of an actor is a channel that accepts any type of message.

4.4 Thread-based vs. Event-based actors

Scala makes the distinction between thread-based and event-based actors.

Thread-based actors are actors which each run in their own JVM thread. They are scheduled by the Java thread scheduler, which uses a preemptive priority-based scheduler. When the actor enters a `receive` block, the thread is blocked until messages arrive. Thread-based actors make it possible to do long-running computations, or blocking I/O operations inside actors, without hindering the execution of other actors.

There is an important drawback to this method: each thread can be considered as being heavy-weight and uses a certain amount of memory and imposes some scheduling overhead. When large amounts of actors are started, the virtual machine might run out of memory or it might perform suboptimal due to large scheduling overhead.

In situations where this is unacceptable, *event-based actors* can be used. These actor are not implemented by means of one thread per actor, yet instead they run on the same thread. An actor that waits for a message to be received is not represented by a blocked thread, but by a closure. This closure captures the state of the actor, such that it's computation can be continued upon receiving a message [7]. The execution of this closure happens on the thread of the sender.

Event-based actors provide a more light-weight alternative, allowing for very large numbers of concurrently running actors. They should however not be used for parallelism: since all actors execute on the same thread, there is no scheduling fairness.

A Scala programmer can use event-based actors by using a `react` block instead of a `receive` block. There is one big limitation to using event-based actors: upon entering a `react` block, the control flow can never return to the enclosing actor. In practice, this does not prove to be a severe limitation, as the code can usually be rearranged to fit this scheme. This property is enforced through the advanced Scala type system, which allows specifying that a method never returns normally (the `Nothing` type). The compiler can thus check if code meets this requirement.

4.5 Actor scheduling in Scala

Scala allows programmers to mix thread-based actors and event-based actors in the same program: this way programmers can choose whether they want scalable, lightweight event-based actors, or thread-based actors that allow for parallelism, depending on the situation in which they are needed.

Scala uses a thread pool to execute actors. This thread pool will be resized automatically whenever necessary. If only event-based actors are used, the size of this thread pool will remain constant. When blocking operations are used, like `receive` blocks, the scheduler (which runs in its own separate thread) will start new threads when needed. Periodically, the scheduler will check if there are runnable tasks in the task queue, it will then check if all worker threads are blocked and start a new worker thread if needed.

5 Evaluation

The next section will evaluate some of the features in both languages. It aims to show that while some of these features facilitate the implementation, their power also comes with a risk. One should be aware of the possible drawbacks of the chosen technologies, to avoid potential pitfalls.

5.1 The dangers of synchronous message passing

The synchronous message passing style available in Scala (using `!?`) provides programmers with a convenient way of doing messaging round-trips. This allows for a familiar style of programming, similar to remote method invocation.

It should however be used with great care. Due to the very selective nature of the `match` clause that follows the use of `!?` (it usually matches only one type of message), the actor is effectively blocked until a suitable reply is received. This is implemented using a private return channel [8], which means that the progress of the actor is fully dependent on the actor from which it awaits a reply: it cannot handle any messages other than the expected reply, not even if they come from the actor from which it awaits reply.

This is a dangerous situation, as it might lead to deadlocks. To see this, consider the following example (listings 10 and 11):

```
actorB !? Msg1(value) match {
    case Response1(r) => // ...
}

receive {
    case Msg2(value) =>
        reply(Response2(value))
}
```

Listing 10: Actor deadlock: Actor A

```
actorA !? Msg2(value) match {
    case Response2(r) => // ...
}

receive {
    case Msg1(value) =>
        reply(Response1(value))
}
```

Listing 11: Actor deadlock: Actor B

In this example, each actor sends a message to the other actor. It will never receive an answer, as the actor is first awaiting a different message. If it were implemented using a message loop, like you would generally do in Erlang, no problem would arise. This is shown in listings 12 and 13. This does not mean that synchronous message passing should be avoided: in certain cases, it is necessary and in these cases the extra syntax makes this much easier to program. It is however important to be aware of the potential problems caused by this programming style.

```
actorB ! Msg1(value)
while (true) {
    receive {
        case Msg2(value) =>
            reply(Response2(value))
        case Response1(r) => // ...
    }
}
```

Listing 12: Safe loop: Actor A

```
actorA ! Msg2(value)
while (true) {
    receive {
        case Msg1(value) =>
            reply(Response1(value))
        case Response2(r) => // ...
    }
}
```

Listing 13: Safe loop: Actor B

The full source code for these examples can be found online, see appendix A for more information.

5.2 Safety in Scala concurrency

Another potential pit-fall in Scala comes from the fact that it mixes actors with object-oriented programming. It is possible to expose the internal state of an actor through publicly available methods for retrieving and modifying this state. When doing so, it is possible to modify an object by directly invoking its methods, that is: without using messages. Doing so means that you no longer enjoy the safety provided by the actor model.

Erlang on the other hand, due to its functional nature, strictly enforces the use of messages between processes: there is no other way to retrieve and update information in other processes.

This illustrates possibly the biggest trade-off between Erlang and Scala: having a pure functional language, like Erlang, is safe, but more difficult for programmers to use. The object-oriented, imperative style of Scala is more familiar and makes programming easier, yet requires more discipline and care to produce safe and correct programs.

6 Summary

This paper described the actor model for the implementation of concurrency in applications, as an alternative to threading and locking. It highlighted Erlang and Scala, two languages with an implementation of the actor model and showed how these languages implement this model.

Erlang is a pure functional language, providing little more than the basic features of functional languages. This should certainly not be seen as a weakness though: this simplicity allows it to optimize specifically for the cases for which it was defined as well as implement more advanced features like hot-swapping of code.

Scala on the other hand uses a mix of object-oriented and functional styles. This makes it easier for a programmer to write code, especially given the extra constructs offered by Scala, but this flexibility comes with a warning: discipline should be used to avoid inconsistencies.

The differences between these languages should be seen and evaluated in their design context. Both however provide an easy to use implementation of the actor model, which greatly facilitates the implementation of concurrency in applications.

A Sample programs source code

The source code for the sample programs can be found on *http://files.savanne.be/kuleuven/erlang-scala-concurrency.tar.gz*. It has been tested on Ubuntu Linux 8.10, with Erlang 5.6.3 and Scala 2.7.2final (installed from Debian packages). It should work on any platform.

References

[1] Erlang. *http://www.erlang.org/*.

[2] Scala on Microsoft .NET. *http://www.scala-lang.org/node/168*.

[3] The Scala Programming Language. *http://www.scala-lang.org/*.

[4] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] J. Armstrong. The development of Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM.

[6] Ericsson. Erlang Goes Multi-Core. *http://tinyurl.com/erlang-multi-core*, 2006.

[7] P. Haller and M. Odersky. Event-Based Programming without Inversion of Control. In *Proc. JMLC 2006*, 2006.

[8] P. Haller and M. Odersky. Actors that Unify Threads and Events. In *In International Conference on Coordination Models and Languages, LNCS*, 2007.

[9] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[10] M. Odersky. Scala by Example. 2009.

[11] M. Odersky, L. Spoon, and B. Venners. Scala: A Scalable Language. *http://www.artima.com/scalazine/articles/scalable-language.html*, 2008.

[12] M. Odersky et al. An Overview of the Scala Programming Language, Second Edition. Technical Report LAMP-REPORT-2006-001, 2006.

[13] R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang (2nd Edition)*. Prentice-Hall, January 1996.

[14] U. Wiger. Erlang Scheduler: what does it do? (in an e-mail to the Erlang mailing list). *http://erlang.org/pipermail/erlang-questions/2001-April/003131.html*, 2001.