Event-Based Programming without Inversion of Control

Philipp Haller, Martin Odersky

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

1 Introduction

Concurrent programming is indispensable. On the one hand, distributed and mobile environments naturally involve concurrency. On the other hand, there is a general trend towards multi-core processors that are capable of running multiple threads in parallel.

With *actors* there exists a computation model which is especially suited for concurrent and distributed computations [16,1]. Actors are basically concurrent processes which communicate through *asynchronous message passing*. When combined with *pattern matching* for messages, actor-based process models have been proven to be very effective, as the success of Erlang documents [3,25].

Erlang [4] is a dynamically typed functional programming language designed for programming real-time control systems. Examples of such systems are telephone exchanges, network simulators and distributed resource controllers. In these systems, large numbers of concurrent processes can be active simultaneously. Moreover, it is generally difficult to predict the number of processes and their memory requirements as they vary with time.

For the implementation of these processes, operating system threads and threads of virtual machines, such as the Java Virtual Machine [22], are usually too heavyweight. The main reasons are: (1) Over-provisioning of stacks leads to quick exhaustion of virtual address space and (2) locking mechanisms often lack suitable contention managers [12]. Therefore, Erlang implements concurrent processes by its own runtime system and not by the underlying operating system [5].

Actor abstractions as lightweight as Erlang's processes have been unavailable on popular virtual machines so far. At the same time, standard virtual machines are becoming an increasingly important platform for exactly the same domain of applications in which Erlang–because of its process model–has been so successful: Real-time control systems [23,27].

Another domain where virtual machines are expected to become ubiquitous are applications running on mobile devices, such as cellular phones or personal digital assistants [20]. Usually, these devices are exposed to severe resource constraints. On such devices, only a few hundred kilobytes of memory is available to a virtual machine and applications.

This has important consequences: (1) A virtual machine for mobile devices usually offers only a restricted subset of the services of a common virtual machine for desktop or server computers. For example, the KVM[1] has no support for reflection (introspection) and serialization. (2) Programming abstractions used by applications have to be very lightweight to be useful. Again, thread-based concurrency abstractions are too heavyweight. Furthermore, programming models have to cope with the restricted set of services a mobile virtual machine provides.

A common alternative to programming with threads are event-driven programming models. Programming in explicitly event-driven models is very difficult [21].

Most programming models support event-driven programming only through *inversion of control*. Instead of calling blocking operations (e.g. for obtaining user input), a program merely registers its interest to be resumed on certain *events* (e.g. an event signaling a pressed button, or changed contents of a text field). In the process, *event handlers* are installed in the execution environment which are called when certain events occur. The program never calls these event handlers itself. Instead, the execution environment dispatches events to the installed handlers. Thus, control over the execution of program logic is "inverted".

Virtually all approaches based on inversion of control suffer from the following two problems: First, the interactive logic of a program is fragmented across multiple event handlers (or classes, as in the state design pattern [13]). Second, control flow among handlers is expressed implicitly through manipulation of shared state [10].

To obtain very lightweight abstractions without inversion of control, we make actors *thread-less*. We introduce *event-based actors* as an implementation technique for lightweight actor abstractions on *non-cooperative* virtual machines such as the JVM. Non-cooperative means that the virtual machine provides no means to explicitly manage the execution state of a program.

The central idea is as follows: An actor that waits in a receive statement is not represented by a blocked thread but by a closure that captures the rest of the actor's computation. The closure is executed once a message is sent to the actor that matches one of the message patterns specified in the receive. The execution of the closure is "piggy-backed" on the thread of the sender. If the receiving closure terminates, control is returned to the sender as if a procedure returns. If the receiving closure blocks in a second receive, control is returned to the sender by throwing a special exception that unwinds the receiver's call stack.

A necessary condition for the scheme to work is that receivers never return normally to their enclosing actor. In other words, no code in an actor can depend on the termination or the result of a receive block. We can express this non-returning property at compile time through Scala's type system. This is not a severe restriction in practice, as programs can always be organized in a way so that the "rest of the computation" of an actor is executed from within a receive.

[1] See http://java.sun.com/products/cldc/.

To the best of our knowledge, event-based actors are the first to (1) allow reactive behavior to be expressed without *inversion of control*, and (2) support arbitrary blocking operations in reactions, at the same time. Our actor library outperforms other state-of-the-art actor languages with respect to message passing speed and memory consumption by several orders of magnitude. Our implementation is able to make use of multi-processors and multi-core processors because reactions can be executed simultaneously on multiple processors. By extending our event-based actors with a portable runtime system, we show how the essence of distributed Erlang [31] can be implemented in Scala. Our library supports virtually all primitives and built-in-functions which are introduced in the Erlang book [4]. The portability of our runtime system is established by two working prototypes based on TCP and the JXTA[2] peer-to-peer framework, respectively.

All this has been achieved without extending or changing the programming language. The event-based actor library is thus a good demonstrator of Scala's abstraction capabilities. Beginning with the upcoming release 2.1.7, it is part of the Scala standard distribution[3].

Other Related Work. Actalk [8] implements actors as a library for Smalltalk-80 by extending a minimal kernel of pure Smalltalk objects. Their implementation is not event-based and Smalltalk-80 does not support parallel execution of concurrent actors on multi-processors (or multi-core processors).

Actra [29] extends the Smalltalk/V virtual machine with an object-based real-time kernel which provides lightweight processes. In contrast, we implement lightweight actors on unmodified virtual machines.

Chrysanthakopoulos and Singh [11] discuss the design and implementation of a channel-based asynchronous messaging library. Channels can be viewed as special state-less actors which have to be instantiated to indicate the types of messages they can receive. Instead of using heavyweight operating system threads they develop their own scheduler to support continuation passing style (CPS) code. Using CLU-style iterators blocking-style code is CPS-transformed by the C# compiler.

SALSA (Simple Actor Language, System and Architecture) [30] extends Java with concurrency constructs that directly support the notion of actors. A preprocessor translates SALSA programs into Java source code which in turn is linked to a custom-built actor library. As SALSA implements actors on the JVM, it is somewhat closer related to our work than Smalltalk-based actors or channels. Moreover, performance results have been published which enables us to compare our system with SALSA, using ports of existing benchmarks.

Timber is an object-oriented and functional programming language designed for real-time embedded systems [6]. It offers message passing primitives for both synchronous and asynchronous communication between concurrent *reactive ob-*

[2] See http://www.jxta.org/.
[3] Available from http://scala.epfl.ch/.

```
class Counter extends Actor {
  override def run(): unit  =  loop(0)

  def loop(value: int): unit = {
    Console.println("Value:␣" + value)
    receive {
      case Incr()   =>  loop(value + 1)
      case Value(a) =>  a ! value; loop(value)
      case Lock(a)  =>  a ! value
                        receive { case UnLock(v) => loop(v) }
      case _        =>  loop(value)
    }
  }
}
```

Fig. 1. A simple counter actor.

jects. In contrast to event-based actors, reactive objects cannot call operations that might block indefinitely. Instead, they install call-back methods in the computing environment which executes these operations on behalf of them.

Frugal objects [14] (FROBs) are distributed reactive objects that communicate through typed events. FROBs are basically actors with an event-based computation model, just as our event-based actors. The goals of FROBs and event-based actors are orthogonal, though. The former provide a *computing model* suited for resource-constrained devices, whereas our approach offers a *programming model* (i.e. a convenient syntax) for event-based actors, such as FROBs. Currently, FROBs can only be programmed using a fairly low-level Java API. In the future, we plan to cooperate with the authors to integrate our two orthogonal approaches.

The rest of this paper is structured as follows. Section 2 shows how conventional, thread-based actors are represented as a Scala library. Section 3 shows how to modify the actor model so that it becomes event-based. Section 4 outlines Scala's package for distributed actors. Section 5 evaluates the performance of our actor libraries. Section 6 concludes.

2 Decomposing Actors

This section describes a Scala library that implements abstractions similar to processes in Erlang. Actors are self-contained, logically active entities that communicate through asynchronous message passing. Figure 1 shows the definition of a counter actor. The actor repeatedly executes a `receive` operation, which waits for three kinds of messages:

- The `Incr` message causes the counter's value to be incremented.
- The `Value` message causes the counter's current value to be communicated to the given actor `a`.
- The `Lock` message is thrown in to make things more interesting. When receiving a `Lock`, the counter will communicate its current value to the given actor `a`. It then blocks until it receives an `UnLock` message. The latter message also specifies the value with which the counter continues from there. Thus, other processes cannot observe state changes of a locked counter until it is unlocked again.

Messages that do not match the patterns `Incr()`, `Value(a)`, or `Lock(a)` are ignored. A typical communication with a counter actor could proceed as follows.

```
val counter = new Counter
counter.start()
counter ! Incr()
counter ! Value(this)
receive { case cvalue => Console.println(cvalue) }
```

This creates a new `Counter` actor, starts it, increments it by sending it the `Incr()` message, and then sends it the `Value` query with the currently executing actor `this` as argument. It then waits for a response of the counter actor in a `receive`. Once some response is received, its value is printed (this value should be one, unless there are other actors interacting with the counter).

Messages in this model are arbitrary objects. In contrast to channel-based programming [11] where a channel usually has to be (generically) instantiated with the types of messages it can handle, an actor can receive messages of any type.

In our example, actors communicate using instances of the following four message classes.

```
case class Incr()
case class Value(a: Actor)
case class Lock(a: Actor)
case class UnLock(value: int)
```

All classes have a **`case`** modifier which enables constructor patterns for the class (see below). Neither class has a body. The `Incr` class has a constructor that takes no arguments, the `Value` and `Lock` classes have a constructor that takes an `Actor` as a parameter, and the `UnLock` class has a constructor that takes an integer argument.

A message send `a!m` sends the message `m` to the actor `a`. The communication is asynchronous: if `a` is not ready to receive `m`, then `m` is queued in a mailbox of `a` and the send operation terminates immediately.

Messages are processed by the `receive` construct, which has the following form:

```
receive {
  case p1 => e1
  ...
  case pn => en
}
```

Here, messages in an actor's mailbox are matched against the patterns $p_1, \ldots, p_n$. Patterns consist of constructors and variables. A constructor pattern names a case class; it matches all instances of this class. A variable pattern matches every value and binds the value to the variable. For example, the pattern `Value(a)` matches all instances v of the `Value` class and binds the variable `a` to the constructor argument of v.

A `receive` will select the first message in an actor's mailbox that matches any of its patterns. If a pattern p_i matches, its corresponding action e_i is executed. If no message in the mailbox matches a pattern, the actor will suspend, waiting for further messages to arrive.

Looking at the example 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 example 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 each construct is defined and implemented.

An actor is simply a subclass of the host environment's `Thread` class that defines methods `!` and `receive` for sending and receiving messages.

```
abstract class Actor extends Thread {
  private var mailbox: List[Any]
  def !(msg: Any) = ...
  def receive[a](f: PartialFunction[Any, a]): a = ...
  ...
}
```

The `!` method is used to send a message to an actor. The send syntax `a!m` is simply an abbreviation of the method call `a.!(m)`, just like `x+y` in Scala is an abbreviation for `x.+(y)`. The method does two things. First, it enqueues the message argument in the actor's mailbox, which is represented as a private field of type `List[Any]`. Second, if the receiving actor is currently suspended in a `receive` that could handle the sent message, the execution of the actor is resumed.

The `receive { ... }` construct is more interesting. Here, the pattern matching expression inside the braces is treated in Scala as a first-class object that is passed as an argument to the `receive` method. The argument's type is an instance of `PartialFunction`, which is a subclass of `Function1`, the class of unary functions. The two classes are defined as follows.

```
abstract class Function1[-a,+b] {
  def apply(x: a): b
}
abstract class PartialFunction[-a,+b] extends Function1[a,b] {
  def isDefinedAt(x: a): boolean
}
```

So we see that functions are objects which have an `apply` method. Partial functions are objects which have in addition a method `isDefinedAt` which can be used to find out whether a function is defined at a given value. Both classes are parameterized; the first type parameter `a` indicates the function's argument type and the second type parameter `b` indicates its result type[4].

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 `receive` as follows. First, messages `m` in the mailbox are scanned in the order they appear. If `receive`'s argument `f` is defined for some of the messages, 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 in the mailbox, the thread associated with the actor is suspended.

This sums up the essential implementation of thread-based actors. There is also some other functionality in Scala's actor libraries which we have not covered. For instance, there is a method `receiveWithin` which can be used to specify a time span in which a message should be received allowing an actor to time-out while waiting for a message. Upon timeout the action associated with a special `TIMEOUT()` pattern is fired. Timeouts can be used to suspend an actor, completely flush the mailbox, or to implement priority messages [4].

Thread-based actors are useful as a higher-level abstraction of threads, which replace error-prone shared memory accesses and locks by asynchronous message passing. However, like threads they incur a performance penalty on standard platforms such as the JVM, which prevents scalability. In the next section we show how the actor model can be changed so that actors become disassociated from threads.

[4] 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`.

3 Recomposing Actors

Logically, an actor is not bound to a thread of execution. Nevertheless, virtually all implementations of actor models associate a separate thread or even an operating system process with each actor [8,29,9,30].

In Scala, thread abstractions of the standard library are mapped onto the thread model and implementation of the corresponding target platform, which at the moment consists of the JVM and Microsoft's CLR [15].

To overcome the resulting problems with scalability, we propose an event-based implementation where (1) actors are thread-less, and (2) computations between two events are allowed to run to completion. An event in our library corresponds to the arrival of a new message in an actor's mailbox.

3.1 Execution Example

First, we want to give an intuitive explanation of how our event-based implementation works. For this, we revisit our counter example from section 2.

Let `c` be a new instance of a lockable counter (with an empty mailbox). After starting `c` it immediately blocks, waiting for a matching message. Consider the case where another actor `p` sends the message `Lock(p)` to `c` (`c ! Lock(p)`). Because the arrival of this `Lock` message enables `c` to continue, send transfers control to `c`. `c` resumes the `receive` statement that caused it to block. Instead of executing the receiving actor on its own thread, we reuse the sender's thread.

According to the semantics of `receive`, the new message is selected and removed from the mailbox because it matches the first case of the outer `receive`. Then, the corresponding action is executed with the pattern variables bound to the constituents of the matched message:

```
{ case Incr()   =>  loop(value + 1)
  case Value(a) =>  a ! value; loop(value)
  case Lock(a)  =>  a ! value
                    receive { case UnLock(v) => loop(v) }
  case _        =>  loop(value)
}.apply(Lock(p))
```

Intuitively, this reduces to

```
p ! value
receive { case UnLock(v) => loop(v) }
```

After executing the message send `p ! value`, the call to `receive` blocks as there are no other messages in `c`'s mailbox. Remember that we are still inside `p`'s original message send (i.e. the send did not return, yet). Thus, blocking the current thread (e.g., by issuing a call to `wait()`) would also block `p`.

This is illegal because in our programming model the send operation (`!`) has a non-blocking semantics. Instead, we need to suspend `c` in such a way that allows `p` to continue. For this, inside the (logically) blocking `receive`, first, we remember the rest of `c`'s computation. In this case, it suffices to save the closure of

```
receive { case UnLock(v) => loop(v) }
```

Second, to let `p`'s call of the send operation return, we need to unwind the runtime stack up to the point where control was transferred to `c`. We do this by throwing a special exception. The `!` method catches this exception and returns normally, keeping its non-blocking semantics.

In general, though, it is not sufficient to save a closure to capture the rest of the computation of an actor. For example, consider an actor executing the following statements:

```
val x = receive { case y => f(y) }
g(x)
```

Here, `receive` produces a value which is then passed to a function. Assume `receive` blocks. Remember that we would need to save the rest of the computation *inside* the blocking `receive`.

To save information about statements following `receive`, we would need to save the call-stack, or capture a (first-class) continuation. Virtual machines such as the JVM provide no means for explicit stack management, mainly because of security reasons. Thus, languages implementing first-class continuations have to simulate the run-time stack on the heap which poses serious performance problems [7]. Moreover, programming tools such as debuggers and profilers rely on run-time information on the native VM stack which they are unable to find if the stack that programs are using is allocated on the heap. Consequently, existing tools cannot be used with programs compiled using a heap-allocated stack.

Thus, most ports of languages with continuation support (e.g. Scheme [18], Ruby [24]) onto non-cooperative virtual machines abandon first-class continuations altogether (e.g. JScheme [2], JRuby[5]). Scala does not support first-class continuations either, primarily because of compatibility and interoperability issues with existing Java code.

To conclude, managing information about statements following a call to `receive` would require changes either to the compiler or the VM. Following our rationale for a library-based approach, we want to avoid those changes.

Instead, we require that `receive` *never returns normally*. Thus, managing information about succeeding statements is unnecessary. Moreover, we can enforce this "no-return" property at compile time through Scala's type system which

[5] See http://jruby.sourceforge.net/.

states that statements following calls to functions (or methods) with return type `Nothing` will never get executed ("dead code") [26]. Note that returning by throwing an exception is still possible. In fact, as already mentioned above, our implementation of `receive` relies on it.

Using a non-returning `receive`, the above example could be coded like this:

```
receive { case y => x = f(y); g(x) }
```

Basically, the rest of the actor's computation has to be called at the end of each case inside the argument function of `receive` ("continuation passing" style).

3.2 Single-Threaded Actors

As we want to avoid inversion of control `receive` will (conceptually) be executed at the expense of the sender. If all actors are running on a single thread, sending a message to an actor A will resume the execution of `receive` which caused A to suspend. The code below shows a simplified implementation of the send operation for actors that run on a single thread:

```
def !(msg: Any): unit = {
  mailbox += msg
  if (continuation != null && continuation.isDefinedAt(msg))
    try { receive(continuation) }
    catch {
      case Done => // do nothing
    }
}
```

The sent message is appended to the mailbox of the actor which is the target of the send operation. Let A denote the target actor. If the `continuation` attribute is set to a non-null value then A is suspended waiting for an appropriate message (otherwise, A did not execute a call to `receive`, yet). As `continuation` refers to (the closure of) the partial function with which the last blocking `receive` was called, we can test if the newly appended message allows A to continue.

Note that if, instead, we would save `receive(f)` as continuation for a blocking `receive(f)` we would not be able to test this but rather had to blindly call the continuation. If the newly appended message would not match any of the defined patterns, `receive` would go through all messages in the mailbox again trying to find the first matching message. Of course, the attempt would be in vain as only the newly appended message could have enabled A to continue.

If A is able to process the newly arrived message we let A continue until it blocks on a nested `receive(g)` or finishes its computation. In the former case, we first save the closure of `g` as A's continuation. Then, the send operation that originated A's execution has to return because of its non-blocking semantics. For this, the blocking `receive` throws a special exception of type `Done` (see below)

which is caught in the send method (`!`). Technically, this trick unwinds the call-stack up to the point where the message send transferred control to A. Thus, to complete the explanation of how the implementation of the send operation works, we need to dive into the implementation of `receive`.

The `receive` method selects messages from an actor's mailbox and is responsible for saving the continuation as well as abandoning the evaluation context:

```
def receive(f: PartialFunction[Any, unit]): Nothing = {
  mailbox.dequeueFirst(f.isDefinedAt) match {
    case Some(msg) => continuation = null
                      f(msg)
    case None      => continuation = f
  }
  throw new Done
}
```

Naturally, we dequeue the first message in our mailbox which matches one of the cases defined by the partial function which is provided as an argument to `receive`. Note that `f.isDefinedAt` has type `Any => boolean`. As the type of the resulting object is `Option[Any]` which has two cases defined, we can select between these cases using pattern matching. When there was a message dequeued we first reset the saved continuation. This is necessary to prevent a former continuation to be called multiple times when there is a send to the current actor inside the call `f(msg)`.

If we didn't find a matching message in the mailbox, we remember the continuation which is the closure of `f`. In both cases we need to abandon the evaluation context by throwing a special exception of type `Done`, so the sender which originated the call to `receive` can continue normally (see above).

3.3 Multi-Threaded Actors

To leverage the increasingly important class of multi-core processors (and also multi-processors) we want to execute concurrent activities on multiple threads. We rely on modern VM implementations to execute concurrent VM threads on multiple processor cores in parallel.

A scheduler decides how many threads to spend for a given workload of concurrent actors, and, naturally, implements a specific scheduling strategy. Because of its asychronous nature, a message send introduces a concurrent activity, namely the resumption of a previously suspended actor. We encapsulate this activity in a *task item* which gets submitted to the scheduler (in a sense this is a *rescheduling send* [28]):

```
def send(msg: Any): unit = synchronized {
  if (continuation != null
      && continuation.isDefinedAt(msg)
      && !scheduled) {
    scheduled = true
    Scheduler.putTask(new ReceiverTask(this, msg))
  } else mailbox += msg
}
```

If a call to send finds the current continuation of the receiving actor A to be undefined, A is not waiting for a message. Usually, this is the case when a task for A has been scheduled that has not been executed, yet. Basically, send appends the argument message to the mailbox unless the receiving actor is waiting for a message and is able to process the argument message. In this case, we schedule the continuation of the receiving actor for execution by submitting a new task item to the scheduler.

The scheduler maintains a pool of worker threads which execute task items. A `ReceiverTask` is basically a Java `java.lang.Runnable` that receives a specified message and has an exception handler that handles requests for abandoning the evaluation context:

```
class ReceiverTask(actor: Actor, msg: Any) extends Runnable {
  def run(): unit =
    try { actor receiveMsg msg }
    catch {
      case Done => // do nothing
    }
}
```

`receiveMsg` is a special form of `receive` which processes a given message according to the actor's continuation.

Actors are not prevented from calling operations which can block indefinitely. In the following we describe a scheduler which guarantees progress even in the presence of blocking operations.

3.4 Blocking Operations

The event-based character of our implementation stems from the fact that (1) actors are thread-less, and (2) computations between the reception of two messages are allowed to run to completion. The second property is common for event-driven systems [17] and reflects our assumption of a rather interactive character for most actors. Consequently, computations between arrival of messages are expected to be rather short compared to the communication overhead.

Nevertheless, we also want to support long-running, CPU-bound actors. Such actors should not prevent other actors from making progress. Likewise, it would

be unfortunate if a single blocking actor could cause the whole application to stop responding, thereby hindering other actors to make progress.

We face the same problems as user-level thread libraries: Processes yield control to the scheduler only at certain program points. In between they cannot be prevented from calling blocking operations or executing infinite loops. For example, an actor might call a native method which issues a blocking system call.

In our case, the scheduler is executed only when sending a message leads to the resumption of another actor. Because send is not allowed to block, the receiver (which is resumed) needs to be executed on a different thread. This way, the sender is not blocked even if the receiver executes a blocking operation.

As the scheduler might not have an idle worker thread available (because all of them are blocked), it needs to create new worker threads as needed. However, if there is at least one worker thread runnable (i.e. busy executing an actor), we do not create a new thread. This is to prevent the creation of too many threads even in the absence of blocking operations.

Actors are still thread-less, though: Each time an actor is suspended because of a blocking (which means unsuccessful) receive, instead of blocking the thread, it is *detached* from its thread. The thread now becomes idle, because it has finished executing a receiver task item. It will ask the scheduler for more work. Thereby, threads are reused for the execution of multiple actors.

Using this method, an actor-based application with low concurrency can be executed by as few as two threads, regardless of the number of simultaneously active actors.

Implementation. Unfortunately, it is impossible for user-level code to find out if a thread running on the JVM is blocked. We therefore implemented a simple heuristic that tries to approximate if a worker thread which executes an actor is blocked, or not.

The basic idea is that actors provide the scheduler with life-beats during their execution. That is, the send (`!`) and `receive` methods call a `tick` method of the scheduler. The scheduler then looks up the worker thread which is currently executing the corresponding actor, and updates its time stamp. When a new receiver task item is submitted to the scheduler, it first checks if all worker threads are blocked. Worker threads with "recent" time stamps are assumed not to be blocked. Only if all worker threads are assumed to be blocked (because of old time stamps), a new worker thread is created. Otherwise, the task item is simply put into a queue waiting to be consumed by an idle worker thread. Figure 2 shows the main part of the scheduler implementation.

Note that using the described approximation, it is impossible to distinguish blocked threads from threads that perform long-running computations. This means basically that compute-bound actors execute on their own thread.

```
def execute(item: ReceiverTask): unit = synchronized {
  if (idle.length > 0) {
    val worker = idle.dequeue
    executing.update(item.actor, worker)
    worker.execute(item)
  } else {
    val iter = workers.elements
    var foundBusy = false
    while (iter.hasNext && !foundBusy) {
      val worker = iter.next
      ticks.get(worker) match {
        case None => foundBusy = true
        case Some(ts) => {
          val currTime = System.currentTimeMillis
          if (currTime - ts < TICKFREQ)
            foundBusy = true
        }
      }
    }
    if (!foundBusy) {
      val worker = new WorkerThread(this)
      workers += worker
      executing.update(item.actor, worker)
      worker.execute(item)
      worker.start()
    } else tasks += item
  }
}
```

Fig. 2. Scheduling work items.

For some applications it might be worth using a scheduler which optimizes the number of spare worker threads depending on runtime profiles. User-defined schedulers are easy to implement and use with our library.

In summary, additional threads are created only when needed to support (unexpected) blocking operations. The only blocking operation that is handled without thread support is `receive`. Thus, a large number of non-cooperative actors (those using blocking operations other than what our library provides), may lead to a significant increase in memory consumption as the scheduler creates more and more threads.

On the other hand, our approach adds significant flexibility, as the library does not need to be changed when the user decides to use a blocking operation which has no special library support. This also means that actors and standard VM threads can be combined seamlessly. We discovered an important use case when porting our runtime system to use JXTA as transport layer: Providing an actor-based interface to a thread-based library.

4 Distributed Actors

With the help of a portable runtime system actors can be executed in a distributed fashion. More specifically, message sends are location transparent and actors can be spawned on remote nodes. As we also target resource-constrained devices, runtime services need to be runnable on virtual machines which offer only a subset of the functionality of standard desktop virtual machines. For example, the KVM[6] does not support reflection. Thus, our serialization mechanism is not based on a general reflective scheme. Instead, we provide a combinator library which allows efficient picklers for custom datatypes to be constructed easily. The pickler combinators are based on Kennedy's library for Haskell [19]. The generated byte arrays are compact because of (1) structure sharing, and (2) base128 encoded integers.

Our runtime system is portable in the sense that network protocol dependent parts are isolated in a separate layer which provides network services (connection management, message transmission, etc.). Two working prototype implementations of the service layer based on TCP and JXTA, respectively, establish portability in practice. TCP and JXTA are protocols different enough that we expect no difficulties porting our runtime system to other network protocols in the future.

We are currently working on the addition of the SOAP[7] XML-over-HTTP protocol as transport layer. One of the goals is to provide an actor-based interface to web services such as the publicly exposed APIs of Google and Amazon. Moreover, we want to build web services in terms of actors.

5 Performance Evaluation

In this section we examine performance properties of our event-based implementation of actors. In the process, we compare benchmark execution times with SALSA [30], a state-of-the-art Java-based actor language, as well as with a thread-based version of our library. As a reference we also show the performance of a straight-forward implementation using threads and synchronized data structures. In addition to execution time we are also interested in scalability with respect to the number of simultaneously active actors each system can handle.

Experimental Set-Up. We measure the throughput of blocking operations in a queue-based application. The application is structured as a ring of n producers/consumers (in the following called *processes*) with a shared queue between each of them. Initially, k of these queues contain tokens and the others are empty.

[6] See http://java.sun.com/products/cldc/.
[7] See http://www.w3.org/2000/xp/Group/.

Each process loops removing an item from the queue on its right and placing it in the queue on its left.

The following tests were run on a 1.60GHz Intel Pentium M processor with 512 MB memory, running Sun's Java HotSpot Client VM 1.5.0 under Linux 2.6.12. We set the JVM's maximum heap size to 256 MB to provide for sufficient physical memory to avoid any disk activity. In each case we took the median of 5 runs.

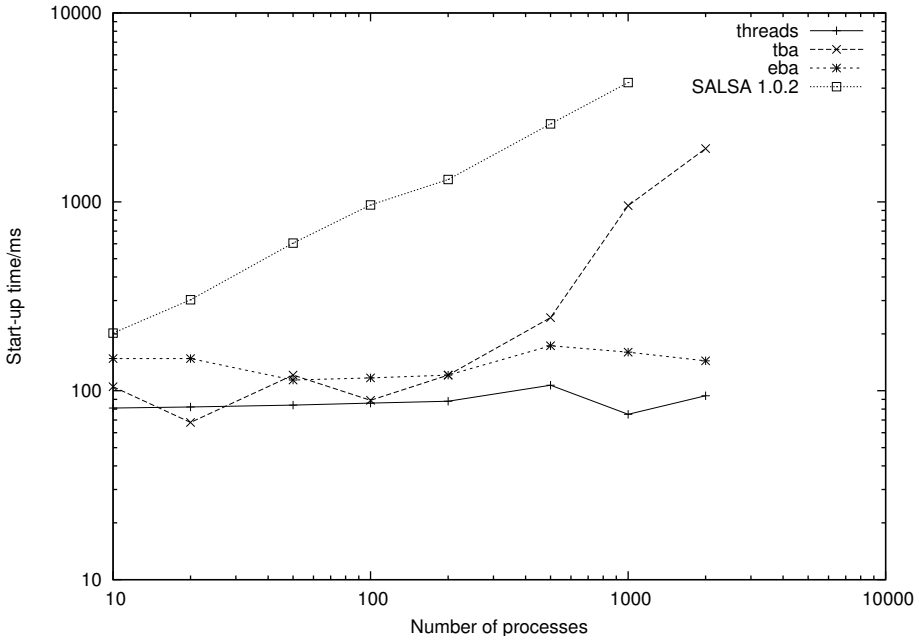

Fig. 3. Start-up time.

The execution times of three equivalent actor-based implementations written using (1) our event-based actor library, (2) a thread-based version of a similar library, and (3) SALSA, respectively, are compared.

Benchmark Results. Figure 3 shows start-up times of the ring for up to 2000 processes (note that both scales are logarithmic). For event-based actors and the naïve thread-based implementation, start-up time is basically constant. Event-based actors are about 60% slower than pure threads. However, we have reasons to suggest that this is due to the different benchmark implementations. In all actor-based implementations, start-up time is measured by starting all actors and letting them wait to receive a special "continue" message. In contrast, the thread-based implementation only *creates* all required threads without starting them. Our measurements suggest that the used JVM optimizes thread creation,

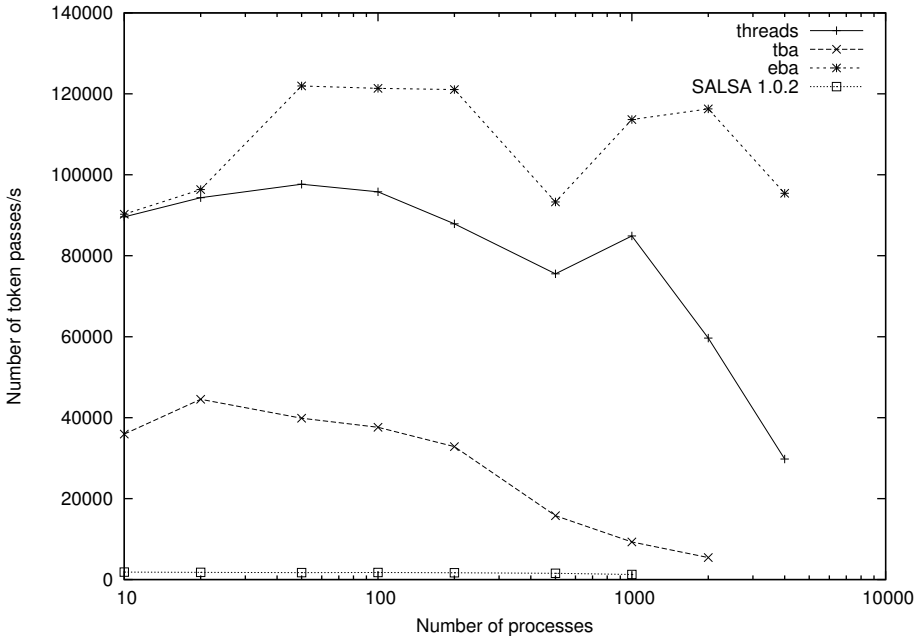

Fig. 4. Throughput (number of token passes per second) for a fixed number of 10 tokens.

potentially creating required runtime structures lazily on start-up. For thread-based actors, start-up time increases exponentially when the number of processes approaches a thousand. With 4000 processes the JVM crashes because of exhaustion of maximum heap size.

Using SALSA, the VM was unable to create 2000 processes. As each actor has a thread-based state object associated with it, the VM is unable to handle stack space requirements at this point. In contrast, using event-based actors the ring can be operated with up to 310000 processes that are created in about 10 seconds.

Looking at the generated Java code shows that SALSA spends a lot of time setting up actors for remote communication (creating locality descriptors, name table management, etc.), whereas in our case, an actor must announce explicitly that it wants to participate in remote communications (by calling `alive()`). Creation of locality descriptors and name table management can be delayed up to this point. Also, when an actor is created in SALSA, it sends itself a special "construct" message which takes additional time.

Figure 4 shows the number of token passes per second depending on the ring size. We chose a logarithmic scale for the number of processes to better depict effects which are confined to a high and strongly increasing number of processes. For up to 1000 processes, increase in throughput for event-based actors com-

pared to pure threads averages 22%. As blocking operations clearly dominate, overhead of threads is likely to stem from context switches and contention for locks. Interestingly, overhead vanishes for a small number of processes (10 and 20 processes, respectively). This behavior suggests that contention is not an issue in this case, as uncontended lock management is optimized in Sun's HotSpot VM 1.5. Contention for locks becomes significant at about 2000 processes. Finally, when the number of processes reaches 4000, the threads' time is consumed managing the shared buffers rather than exchanging tokens through them. At this point throughput of event-based actors is about 3 times higher.

For SALSA, throughput is about two orders of magnitude lower compared to event-based actors. The average for 10 to 1000 processes amounts to only 1700 token passes per second. Looking at the generated Java source code revealed that every message send involves a reflective method call. We found reflective method calls to be about 30 times slower than JIT-compiled method calls on our testing machine.

For thread-based actors, throughput is almost constant for up to 200 processes (on average about 38000 token passes per second). At 500 processes it is already less than half of that (15772 token passes per second). Similar to pure threads, throughput breaks in for 2000 processes (only 5426 token passes per second). Again, contended locks and context switching overhead are likely to cause this behavior. The VM is unable to create 4000 processes, because it runs out of memory.

Performance Summary. Event-based actors support a number of simultaneously active actors which is two orders of magnitude higher compared to SALSA. Measured throughput is over 50 times higher compared to SALSA. A naïve thread-based implementation of our benchmark performs surprisingly well. However, for high numbers of threads (about 2000), lock contention causes performance to break in. Also, the maximum number of threads is limited due to their memory consumption.

6 Conclusion

Scala is different from other concurrent languages in that it contains no language support for concurrency beyond the standard thread model offered by the host environment. Instead of specialized language constructs we rely on Scala's general abstraction capabilities to define higher-level concurrency models. In such a way, we were able to define all essential operations of Erlang's actor-based process model in the Scala library.

However, since Scala is implemented on the Java VM, we inherited some of the deficiencies of the host environment when it comes to concurrency, namely low maximum number of threads and high context-switch overhead. In this paper

we have shown how to turn this weakness into a strength. By defining a new event-based model for actors, we could increase dramatically their efficiency and scalability. At the same time, we kept to a large extent the programming model of thread-based actors, which would not have been possible if we had switched to a traditional event-based architecture, because the latter causes an inversion of control.

The techniques presented in this paper are a good showcase of the increased flexibility offered by library-based designs. It allowed us to quickly address problems with the previous thread-based actor model by developing a parallel class hierarchy for event-based actors. Today, the two approaches exist side by side. Thread-based actors are still useful since they allow returning from a receive operation. Event-based actors are more restrictive in the programming style they allow, but they are also more efficient.

In future work we plan to extend the event-based actor implementation to other communication infrastructures. We are also in train of discovering new ways to compose these actors.

References

1. Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, 1986.
2. Ken Anderson, Tim Hickey, and Peter Norvig. Jscheme.
3. 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, Hino, Tokyo, Japan, October 1996.
4. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition.* Prentice-Hall, 1996.
5. Joe L. Armstrong. The development of erlang. In *ICFP*, pages 196–203, 1997.
6. A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems, 2002.
7. Yannis Bres, Bernard P. Serpette, and Manuel Serrano. Bigloo.NET: compiling scheme to.NET CLR. *Journal of Object Technology*, 3(9):71–94, 2004.
8. Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *ECOOP*, pages 109–129, 1989.
9. Legand L. Burge III and K. M. George. JMAS: A Java-based mobile actor system for distributed parallel computation. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 115–129. The USENIX Association, 1999.
10. Brian Chin and Todd Millstein. Responders: Language support for interactive applications. In *ECOOP*, Nantes, France, July 2006.
11. Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for c#. In *Proceedings of Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL), OOPSLA*, 2005.
12. Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *LCN*, pages 455–462, 2004.

13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.
14. Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper H. Spring. Frugal Mobile Objects. Technical report, EPFL, 2005.
15. John Gough. *Compiling for the .NET Common Language Runtime.* .NET series. Prentice Hall, 2002.
16. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
17. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *ASPLOS*, pages 93–104, 2000.
18. Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.
19. Andrew Kennedy. Pickler combinators. *J. Funct. Program.*, 14(6):727–739, 2004.
20. George Lawton. Moving Java into mobile phones. *Computer*, 35(6):17–20, June 2002.
21. P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.
22. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
23. Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *RTSS*, pages 62–71. IEEE Computer Society, 2005.
24. Yukihiro Matsumoto. *The Ruby Programming Language.* Addison Wesley Professional, 2002.
25. J. H. Nyström, Philip W. Trinder, and David J. King. Evaluating distributed functional languages for telecommunications software. In Bjarne Däcker and Thomas Arts, editors, *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003*, pages 1–7. ACM, 2003.
26. Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.
27. F. Pizlo, J. M. Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: Design patterns and semantics. In *ISORC*, pages 101–110. IEEE Computer Society, 2004.
28. Erik Stenman and Konstantinos Sagonas. On reducing interprocess communication overhead in concurrent programs. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Erlang, Pittsburgh, PA, USA, October 7, 2002*, pages 58–63. ACM, 2002.
29. D. A. Thomas, W. R. Lalonde, J. Duimovich, M. Wilson, J. McAffer, and B. Berry. Actra A multitasking/multiprocessing smalltalk. *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, ACM SIGPLAN Notices*, 24(4):87–90, April 1989.
30. Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, 2001.
31. Claes Wikström. Distributed programming in erlang. In Hoon Hong, editor, *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94 (Hagenberg/Linz, Austria, September 26-28, 1994)*, volume 5 of *Lecture Note Series in Computing*, pages 412–421. World Scientific, Singapore-New Jersey-London-Hong Kong, 1994.