Theoretical Computer Science 410 (2009) 202–220

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

# Scala Actors: Unifying thread-based and event-based programming☆

Philipp Haller *, Martin Odersky

*EPFL, Switzerland*

ARTICLE INFO

*Keywords:*
Concurrent programming
Actors
Threads
Events

ABSTRACT

There is an impedance mismatch between message-passing concurrency and virtual machines, such as the JVM. VMs usually map their threads to heavyweight OS processes. Without a lightweight process abstraction, users are often forced to write parts of concurrent applications in an event-driven style which obscures control flow, and increases the burden on the programmer.

In this paper we show how thread-based and event-based programming can be unified under a single actor abstraction. Using advanced abstraction mechanisms of the Scala programming language, we implement our approach on unmodified JVMs. Our programming model integrates well with the threading model of the underlying VM.

© 2008 Elsevier B.V. All rights reserved.

## 1. Introduction

Concurrency issues have lately received enormous interest because of two converging trends: first, multi-core processors make concurrency an essential ingredient of efficient program execution. Second, distributed computing and web services are inherently concurrent. Message-based concurrency is attractive because it might provide a way to address the two challenges at the same time. It can be seen as a higher-level model for threads with the potential to generalize to distributed computation. Many message passing systems used in practice are instantiations of the actor model [28,2]. A popular implementation of this form of concurrency is the Erlang programming language [4]. Erlang supports massively concurrent systems such as telephone exchanges by using a very lightweight implementation of concurrent processes [3,36].

On mainstream platforms such as the JVM [34], an equally attractive implementation was, as yet, missing. Their standard concurrency constructs, shared-memory threads with locks, suffer from high memory consumption and context-switching overhead. Therefore, the interleaving of independent computations is often modeled in an event-driven style on these platforms. However, programming in an explicitly event-driven style is complicated and error-prone, because it involves an inversion of control [41,13].

In previous work [24], we developed *event-based actors* which let one program event-driven systems without inversion of control. Event-based actors support the same operations as thread-based actors, except that the receive operation cannot return normally to the thread that invoked it. Instead the entire continuation of such an actor has to be a part of the receive operation. This makes it possible to model a suspended actor by a continuation closure, which is usually much cheaper than suspending a thread.

In this paper we present a unification of thread-based and event-based actors. An actor can suspend with a full thread stack (`receive`) or it can suspend with just a continuation closure (`react`). The first form of suspension corresponds to thread-based, the second form to event-based programming. The new system combines the benefits of both models.

☆ A preliminary version of the paper appears in the proceedings of COORDINATION 2007, LNCS 4467, June 2007.

* Corresponding address: EPFL, Station 14, 1015 Lausanne, Switzerland. Tel.: +41 21 693 6483; fax: +41 21 693 6660.
*E-mail address:* philipp.haller@epfl.ch (P. Haller).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.09.019

Threads support blocking operations such as system I/O, and can be executed on multiple processor cores in parallel. Event-based computation, on the other hand, is more lightweight and scales to larger numbers of actors. We also present a set of combinators that allows a flexible composition of these actors.

The presented scheme has been implemented in the *Scala Actors* library.[1] It requires neither special syntax nor compiler support. A library-based implementation has the advantage that it can be flexibly extended and adapted to new needs. In fact, the presented implementation is the result of several previous iterations. However, to be easy to use, the library draws on several of Scala's advanced abstraction capabilities; notably partial functions and pattern matching [19].

The user experience gained so far indicates that the library makes concurrent programming in a JVM-based system much more accessible than previous techniques. The reduced complexity of concurrent programming is influenced by the following factors.

- Since accessing an actor's mailbox is race-free by design, message-based concurrency is potentially more secure than shared-memory concurrency with locks. We believe that message-passing with pattern matching is also more convenient in many cases.
- Actors are lightweight. On systems that support 5000 simultaneously active VM threads, over 1,200,000 actors can be active simultaneously. Users are thus relieved from writing their own code for thread-pooling.
- Actors are fully inter-operable with normal VM threads. Every VM thread is treated like an actor. This makes the advanced communication and monitoring capabilities of actors available even for normal VM threads.

Our integration of a high-level actor-based programming model, providing strong invariants and lightweight concurrency, with existing threading models of mainstream VM platforms is unique to the best of our knowledge. We believe that our approach offers a qualitative improvement in the development of concurrent software for multi-core systems.

The rest of this paper is structured as follows. Section 2 introduces our programming model and explains how it can be implemented as a Scala library. In Section 3 we present an extension to our programming model that allows us to unify thread-based and event-based models of concurrency under a single abstraction of actors. We also provide an overview and important details of the implementation of the Scala Actors library. Section 4 illustrates the core primitives of Scala Actors using larger examples. Section 5 introduces channels for type-safe and private communication. By means of a case study (Section 6) we show how our unified programming model can be applied to programming advanced web applications. Experimental results are presented in Section 7. Section 8 discusses related and future work, and Section 9 concludes.

## 2. The Scala Actors library

In the following, we introduce the fundamental concepts underlying our programming model and explain how various constructs are implemented in Scala. Section 2.1 shows how first-class message handlers support the extension of actors with new behavior.

**Actors.** The Scala Actors library provides a concurrent programming model based on *actors*. An actor [28,2] is a concurrent process that communicates with other actors by exchanging messages. Communication is *asynchronous*; messages are buffered in an actor's *mailbox*. An actor may respond to an asynchronous message by creating new actors, sending messages to known actors (including itself), or changing its behavior. The behavior specifies how the actor responds to the next message that it receives.

**Actors in Scala.** Our implementation of actors in Scala adopts the basic communication primitives virtually unchanged from Erlang [4]. The expression `a ! msg` sends message `msg` to actor `a` (asynchronously). 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 (see Fig. 1 for an example of a message pattern). If no pattern matches, the actor suspends.

New actors can be created in two ways. In the first alternative, we define a new class that extends the `Actor` trait.[2] The actor's behavior is defined by its `act` method. For example, an actor executing `body` can be created as follows:

```
class MyActor extends Actor {
  def act() { body } }
```

[1] Available as part of the Scala distribution at http://www.scala-lang.org/.

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

```
// base version
val orderMngr = actor {
  while (true) receive {
    case Order(s, item) =>
      val o =
        handleOrder(s, item)
      s ! Ack(o)
    case Cancel(s, o) =>
      if (o.pending) {
        cancelOrder(o)
        s ! Ack(o)
      } else s ! NoAck
    case x => junk += x
  }
}
val customer = actor {
  orderMngr!Order(self, it)
  receive {
    case Ack(o) => ...
  }
}
```

```
// version with reply and !?
val orderMngr = actor {
  while (true) receive {
    case Order(item) =>
      val o =
        handleOrder(sender, item)
      reply(Ack(o))
    case Cancel(o) =>
      if (o.pending) {
        cancelOrder(o)
        reply(Ack(o))
      } else reply(NoAck)
    case x => junk += x
  }
}
val customer = actor {
  orderMngr!?Order(it) match {
    case Ack(o) => ...
  }
}
```

**Fig. 1.** Example: orders and cancellations.

Note that after creating an instance of the `MyActor` class the actor has to be started by calling its `start` method. The second alternative for creating an actor is as follows. The expression `actor {body}` creates a new actor which runs the code in `body`. Inside `body`, the expression `self` is used to refer to the currently executing actor. This "inline" definition of an actor is often more concise than defining a new class. Finally, we note that every Java thread is also an actor, so even the main thread can execute `receive`.[3]

The example in Fig. 1 demonstrates the usage of all constructs introduced so far. First, we define an `orderMngr` actor that tries to receive messages inside an infinite loop. The `receive` operation waits for two kinds of messages. The `Order(s, 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 `s`. The `Cancel(s, 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 last pattern `x` in the `receive` of `orderMngr` is a variable pattern which matches any message. Variable patterns allow to remove messages from the mailbox that are normally not understood ("junk"). We also define a customer actor which places an order and waits for the acknowledgment of the order manager before proceeding. Since spawning an actor (using `actor`) is asynchronous, the defined actors are executed concurrently.

Note that in the above example we have to do some repetitive work to implement request/reply-style communication. In particular, the sender is explicitly included in every message. As this is a frequently recurring pattern, our library has special support for it. Messages always carry the identity of the sender with them. This enables the following additional operations:

- `a !? msg` sends `msg` to `a`, waits for a reply and returns it.
- `sender` refers to the actor that sent the message that was last received by `self`.
- `reply(msg)` replies with `msg` to `sender`.
- `a forward msg` sends `msg` to `a`, using the current `sender` instead of `self` as the sender identity.

With these additions, the example can be simplified as shown on the right-hand side of Fig. 1. In addition to the operations above, an actor may explicitly designate another actor as the reply destination of a message send. The expression `a.send(msg, b)` sends `msg` to `a` where actor `b` is the reply destination. This means that when `a` receives `msg`, `sender` refers to `b`; therefore, any reply from `a` is sent directly to `b`. This allows certain forwarding patterns to be expressed without creating intermediate actors [45].

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 following, we look "under the covers" to find out how each construct is defined and implemented. The implementation of concurrent processing is discussed in Section 3.3.

[3] Using `self` outside of an actor definition creates a dynamic proxy object which provides an actor identity to the current thread, thereby making it capable of receiving messages from other actors.

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, we define `!` as a method in the `Actor` trait:

```
trait Actor {
  val mailbox = new Queue[Any]
  def !(msg: Any): Unit = ...
  ...
}
```

The method does two things. First, it enqueues the message argument in the receiving actor's mailbox which is represented as a field of type `Queue[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. In Scala, the pattern matching expression inside braces is treated 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 }
```

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 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 `receive` construct is realized as a method (of the `Actor` trait) that takes a partial function as an argument.

```
def receive[R](f: PartialFunction[Any, R]): R
```

The implementation of `receive` proceeds roughly as follows. First, messages in the mailbox are scanned in the order they appear. If `receive`'s argument `f` 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 `actor` and `self` constructs are realized as methods defined by the `Actor` *object*. Objects have exactly one instance at runtime, and their methods are similar to static methods in Java.

```
object Actor {
  def self: Actor ...
  def actor(body: => Unit): Actor ...
  ...
}
```

Note that Scala has different namespaces for types and terms. For instance, the name `Actor` is used both for the object above (a term) and the trait which is the result type of `self` and `actor` (a type). In the definition of the `actor` method, the argument body defines the behavior of the newly created actor. It is a closure returning the unit value. The leading => in its type indicates that it is passed by name.

There is also some other functionality in Scala's actor library 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 timeout 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].

### *2.1. Extending actor behavior*

The fact that message handlers are first-class partial function values can be used to make actors extensible with new behaviors. A general way to do this is to have classes provide actor behavior using methods, so that subclasses can override them.

[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`.

```
class Buffer(N: Int) extends Actor {
  val buf = new Array[Int](N)
  var in = 0; var out = 0; var n = 0
  def reaction: PartialFunction[Any, Unit] = {
    case Put(x) if n < N =>
      buf(in) = x; in = (in + 1) % N; n = n + 1; reply()
    case Get if n > 0 =>
      val r = buf(out); out = (out + 1) % N; n = n - 1; reply(r)
  }
  def act(): Unit = while (true) receive(reaction)
}
class Buffer2(N: Int) extends Buffer(N) {
  override def reaction: PartialFunction[Any, Unit] =
    super.reaction orElse {
      case Get2 if n > 1 =>
        out = (out + 2) % N; n = n - 2
        reply (buf(out-2), buf(out-1))
    }
}
```

**Fig. 2.** Extending actors with new behavior.

Fig. 2 shows an example. The `Buffer` class extends the `Actor` trait to define actors that implement bounded buffers containing at most `N` integers. We omit a discussion of the array-based implementation (using the `buf` array and a number of integer variables) since it is completely standard; instead, we focus on the actor-specific parts. First, consider the definition of the `act` method. Inside an infinite loop it invokes `receive` passing the result of the `reaction` method. This method returns a partial function that defines actions associated with the `Put(x)` and `Get` message patterns. As a result, instances of the `Buffer` class are actors that repeatedly wait for `Put` or `Get` messages.

Assume we want to extend the behavior of buffer actors, so that they also respond to `Get2` messages, thereby removing two elements at once from the buffer. The `Buffer2` class below shows such an extension. It extends the `Buffer` class, thereby overriding its `reaction` method. The new method returns a partial function which combines the behavior of the superclass with a new action associated with the `Get2` message pattern. Using the `orElse` combinator we obtain a partial function that is defined as `super.reaction` except that it is additionally defined for `Get2`. The definition of the `act` method is inherited from the superclass which results in the desired overall behavior.

## 3. Unified actor model and implementation

Traditionally, programming models for concurrent processes are either thread-based or event-based. We review their complementary strengths and weaknesses in Section 3.1. Scala Actors unify both programming models, allowing programmers to trade efficiency for flexibility in a fine-grained way. We present our unified, actor-based programming model in Section 3.2. Section 3.3 provides an overview as well as important details of the implementation of the Scala Actors library. Finally, Section 3.4 introduces a set of combinators that allows one to compose actors in a modular way.

### *3.1. Threads vs. events*

Concurrent processes such as actors can be implemented using one of two implementation strategies:

- Thread-based implementation: The behavior of a concurrent process is defined by implementing a thread-specific method. The execution state is maintained by an associated thread stack (see, *e.g.*, [30]).
- Event-based implementation: The behavior is defined by a number of (non-nested) event handlers which are called from inside an event loop. The execution state of a concurrent process is maintained by an associated record or object (see, *e.g.*, [44]).

Often, the two implementation strategies imply different programming models. Thread-based models are usually easier to use, but less efficient (context switches, memory consumption) [37], whereas event-based models are usually more efficient, but very difficult to use in large designs [41].

Most event-based models introduce an *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.*, signaling a pressed button). In the process, *event handlers* are installed in the execution environment. 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". Because of inversion of control, switching from a thread-based to an event-based model normally requires a global re-write of the program [10,13].

### 3.2. Unified actor model

The main idea of our programming model is to allow an actor to wait for a message using two different operations, called `receive` and `react`, respectively. Both operations try to remove a message from the current actor's mailbox given a partial function that specifies a set of message patterns (see Section 2). However, the semantics of `receive` corresponds to thread-based programming, whereas the semantics of `react` corresponds to event-based programming. In the following we discuss the semantics of each operation in more detail.

#### 3.2.1. The `receive` operation

The `receive` operation has the following type:

```
def receive[R](f: PartialFunction[Any, R]): R
```

If there is a message in the current actor's mailbox that matches one of the cases specified in the partial function `f`, the result of applying `f` to that message is returned. Otherwise, the current thread is suspended; this allows the receiving actor to resume execution normally when receiving a matching message. Note that `receive` retains the complete call stack of the receiving actor; the actor's behavior is therefore a sequential program which corresponds to thread-based programming.

#### 3.2.2. The `react` operation

The `react` operation has the following type:

```
def react(f: PartialFunction[Any, Unit]): Nothing
```

Note that `react` has return type `Nothing`. In Scala's type system a method has return type `Nothing` iff it never returns normally. This means that the action specified in `f` that corresponds to the matching message is the last code that the current actor executes. The semantics of `react` closely resembles event-based programming: the current actor registers the partial function `f` which corresponds to a set of event handlers, and then releases the underlying thread. When receiving a matching message the actor's execution is resumed by invoking the registered partial function. In other words, when using `react`, the argument partial function has to contain the rest of the current actor's computation (its *continuation*) since calling `react` never returns. In Section 3.4 we introduce a set of combinators that hide these explicit continuations.

### 3.3. Implementation

Before discussing the implementation it is useful to clarify some terminology. In the following Section 3.3.1 and Section 3.3.2 we refer to an actor that is unable to continue (*e.g.*, because it is waiting for a message) as being *suspended*. Note that this notion is independent of a specific concurrency model, such as threads. However, it is often necessary to indicate whether an actor is suspended in an event-based or in a thread-based way. We refer to an actor that is suspended in a `react` as being *detached* (since in this case the actor is detached from any other thread). In contrast, an actor that is suspended in a `receive` is called *blocked* (since in this case the underlying worker thread is blocked). More generally, we use the term blocking as a shortcut for *thread-blocking*.

#### 3.3.1. Implementation overview

In our framework, multiple actors are executed on multiple threads for two reasons:

(1) Executing concurrent code in parallel may result in speed-ups on multi-processors and multi-core processors.
(2) Executing two interacting actors on different threads allows actors to invoke blocking operations without affecting the progress of other actors.

Certain operations provided by our library introduce concurrency, namely spawning an actor using `actor`, and asynchronously sending a message using the `!` operator. We call these operations *asynchronous operations*. Depending on the current load of the system, asynchronous operations may be executed in parallel. Invoking an asynchronous operation creates a task that is submitted to a thread pool for execution. More specifically, a task is generated in the following three cases:

(1) Spawning a new actor using `actor {body}` generates a task that executes `body`.
(2) Sending a message to an actor suspended in a `react` that enables it to continue generates a task that processes the message.
(3) Calling `react` where a message can be immediately removed from the mailbox generates a task that processes the message.

The basic idea of our implementation is to use a thread pool to execute actors, and to *resize* the thread pool whenever it is necessary to support general thread operations. If actors use only operations of the event-based model, the size of the thread pool can be fixed. This is different if some of the actors use blocking operations such as `receive` or system I/O. In the case where every worker thread is occupied by a blocked actor and there are pending tasks, the thread pool has to grow.

For example, consider a thread pool with a single worker thread, executing a single actor *a*. Assume *a* first spawns a new actor *b*, and then waits to receive a message from *b* using the thread-based receive operation. Spawning *b* creates a new task that is submitted to the thread pool for execution. Execution of the new task is delayed until *a* releases the worker thread. However, when *a* suspends, the worker thread is *blocked*, thereby leaving the task unprocessed indefinitely. Consequently, *a* is never resumed since the only task that could resume it (by sending it a message) is never executed. The system is deadlocked.

In our library, system-induced deadlocks are avoided by increasing the size of the thread pool whenever necessary. It is necessary to add another worker thread whenever there is a pending task and all worker threads are blocked. In this case, the pending task(s) are the only computations that could possibly unblock any of the worker threads (*e.g.*, by sending a message to a suspended actor). To do this, a scheduler thread (which is separate from the worker threads of the thread pool) periodically checks if there is a task in the task queue and all worker threads are blocked. In that case, a new worker thread is added to the thread pool that processes any remaining tasks.

### *3.3.2. Implementation details*

A detached actor (*i.e.*, suspended in a `react` call) is not represented by a blocked thread but by a closure that captures the actor's continuation. This closure is executed once a message is sent to the actor that matches one of the message patterns specified in the `react`. When an actor detaches, its continuation closure is stored in a `continuation` field of the `Actor` trait:

```
trait Actor {
  var continuation: PartialFunction[Any, Unit]
  val mailbox = new Queue[Any]
  def !(msg: Any): Unit = ...
  def react(f: PartialFunction[Any, Unit]): Nothing = ...
  ...
}
```

An actor's continuation is represented as a partial function of type `PartialFunction[Any, Unit]`. When invoking an actor's continuation we pass the message that enables the actor to resume as an argument. The idea is that an actor only detaches when `react` fails to remove a matching message from the mailbox. This means that a detached actor is always resumed by sending it a message that it is waiting for. This message is passed when invoking the continuation. We represent the continuation as a *partial function* rather than a function to be able to test whether a message that is sent to an actor enables it to continue. This is explained in more detail below.

The `react` method saves the continuation closure whenever the receiving actor has to suspend (and therefore detaches):

```
def react(f: PartialFunction[Any, Unit]): Nothing =
  synchronized {
    mailbox.dequeueFirst(f.isDefinedAt) match {
      case Some(msg) =>
        schedule(new Task({ () => f(msg) }))
      case None =>
        continuation = f
        isDetached = true
        waitingFor = f.isDefinedAt
    }
    throw new SuspendActorException
  }
```

Recall that a partial function, such as `f`, is usually represented as a block with a list of patterns and associated actions. If a message can be removed from the mailbox (tested using `dequeueFirst`) the action associated with the matching pattern is scheduled for execution by calling the `schedule` operation. It is passed a task which contains a delayed computation that applies `f` to the received message, thereby executing the associated action. Tasks and the `schedule` operation are discussed in more detail below.

If no message can be removed from the mailbox, we save `f` as the continuation of the receiving actor. Since `f` contains the complete execution state we can resume the execution at a later point when a matching message is sent to the actor. The instance variable `isDetached` is used to tell whether the actor is detached (as opposed to blocked in a `receive`). If it is, the value stored in the `continuation` field is a valid execution state. The instance variable `waitingFor` stores a function of type `Any => Boolean` that is used to test whether a newly sent message enables the actor to continue. It is needed in addition to the `continuation` field since the latter cannot be used when the actor is blocked in a `receive`.

Finally, by throwing a special exception, control is transferred to the point in the control flow where the current actor was started or resumed. Since actors are always executed as part of tasks, the `SuspendActorException` is only caught inside task bodies.

Tasks are represented as instances of the following class (simplified):

```
class Task(cont: () => Unit) {
  def run() {
    try { cont() } // invoke continuation
    catch { case _: SuspendActorException =>
      // do nothing }
  } }
```

The constructor of the `Task` class takes a continuation of type `() => Unit` as its single argument. The class has a single `run` method that wraps an invocation of the continuation in an exception handler. The exception handler catches exceptions of type `SuspendActorException` which are thrown whenever an actor detaches. The body of the exception handler is empty since the necessary bookkeeping, such as saving the actor's continuation, has already been done at the point where the exception was thrown.

Sending a message to an actor involves checking whether the actor is waiting for the message, and, if so, resuming the actor according to the way in which it suspended (*i.e.*, using `receive` or `react`):

```
def !(msg: Any): Unit = synchronized {
  if (waitingFor(msg)) {
    waitingFor = (x: Any) => false
    if (isDetached) {
      isDetached = false
      schedule(new Task({ () => continuation(msg) }))
    } else
      resume() // thread-based resume
  } else mailbox += msg
}
```

When sending a message to an actor that it does not wait for (*i.e.*, the actor is not suspended or its continuation is not defined for the message), the message is simply enqueued in the actor's mailbox. Otherwise, the internal state of the actor is changed to reflect the fact that it is no longer waiting for a message. Then, we test whether the actor is detached; in this case we schedule a new task that applies the actor's continuation to the newly received message. The continuation was saved when the actor detached the last time. If the actor is not detached (which means it is blocked in a `receive`), it is resumed by notifying its underlying thread which is blocked on a call to `wait`.

Spawning an actor using `actor {body}` generates a task that executes `body` as part of a new actor:

```
def actor(body: => Unit): Actor = {
  val a = new Actor {
    def act() = body }
  schedule(new Task({ () => a.act() }))
  a }
```

The `actor` function takes a delayed expression (indicated by the leading `=>`) that evaluates to `Unit` as its single argument. After instantiating a new `Actor` with the given `body`, we create a new task that is passed a continuation that simply executes the actor's body. Note that the actor may detach later on (*e.g.*, by waiting in a `react`), in which case execution of the task is finished early, and the rest of the actor's body is run as part of a new continuation which is created when the actor is resumed subsequently.

The `schedule` operation submits tasks to a thread pool for execution. A simple implementation strategy would be to put new tasks into a global queue that all worker threads in the pool access. However, we found that a global task queue becomes a serious bottle neck when a program creates short tasks with high frequency (especially if such a program is executed on multiple hardware threads). To remove this bottle neck, each worker thread has its own local task queue. When a worker thread generates a new task, *e.g.*, when a message send enables the receiver to continue, the (sending) worker puts it into its local queue. This means that a receiving actor is *often* executed on the same thread as the sender. This is not always the case, because *work stealing* balances the work load on multiple worker threads (which ultimately leads to parallel execution of tasks) [6]. This means that idle worker threads with empty task queues look into the queues of other workers for tasks to execute. However, accessing the local task queue is much faster than accessing the global task queue thanks to sophisticated non-blocking algorithms [31]. In our framework the global task queue is used to allow non-worker threads (any JVM thread) to invoke asynchronous operations.

As discussed before, our thread pool has to grow whenever there is a pending task and all worker threads are blocked. Unfortunately, on the JVM there is no safe way for library code to find out if a thread is blocked. Therefore, we implemented a conservative heuristic that approximates the predicate "all worker threads blocked". The approximation uses a time-stamp of the last "library activity". If the time-stamp is not recent enough (*i.e.*, it has not changed since a multiple of scheduler runs),

```
class InOrder(n: IntTree) extends Producer[Int] {
  def produceValues() {
    traverse(n)
  }
  def traverse(n: IntTree) {
    if (n != null) {
      traverse(n.left)
      produce(n.elem)
      traverse(n.right)
    }
  }
}
```

**Fig. 3.** Producer that generates all values in a tree in in-order.

the predicate is assumed to hold, *i.e.*, it is assumed that all worker threads are blocked. We maintain a global time-stamp that is updated on every call to send, receive etc.

### 3.4. Composing actor behavior

Without extending the unified actor model, defining an actor that executes several given functions in sequence is not possible in a modular way.

For example, consider the two methods below:

```
def awaitPing = react { case Ping => }
def sendPong = sender ! Pong
```

It is not possible to sequentially compose `awaitPing` and `sendPong` as follows:

```
actor { awaitPing; sendPong }
```

Since `awaitPing` ends in a call to `react` which never returns, `sendPong` would never get executed. One way to work around this restriction is to place the continuation into the body of `awaitPing`:

```
def awaitPing = react { case Ping => sendPong }
```

However, this violates modularity. Instead, our library provides an `andThen` combinator that allows actor behavior to be composed sequentially. Using `andThen`, the body of the above actor can be expressed as follows:

```
awaitPing andThen sendPong
```

`andThen` is implemented by installing a hook function in the actor. This function is called whenever the actor terminates its execution. Instead of exiting, the code of the second body is executed. Saving and restoring the previous hook function permits chained applications of `andThen`.

The `Actor` object also provides a `loop` combinator. It is implemented in terms of `andThen`:

```
def loop(body: => Unit) = body andThen loop(body)
```

Hence, the body of `loop` can end in an invocation of `react`. Similarly, we can define a `loopWhile` combinator that terminates the actor when a provided guard evaluates to `false`.

## 4. Examples

In this section we discuss two larger examples. These examples serve two purposes. First, they show how our unified programming model can be used to make parts of a threaded program event-based with minimal changes to an initial actor-based program. Second, they demonstrate the use of the combinators introduced in Section 3.4 to turn a complex program using non-blocking I/O into a purely event-driven program while maintaining a clear threaded code structure.

### 4.1. Producers and iteration

In the first example, we are going to write an abstraction of *producers* that provide a standard iterator interface to retrieve a sequence of produced values. Producers are defined by implementing an abstract `produceValues` method that calls a `produce` method to generate individual values. Both methods are inherited from a `Producer` class. For example, Fig. 3 shows the definition of a producer that generates the values contained in a tree in in-order.

Fig. 4 shows an implementation of producers in terms of two actors, a *producer* actor, and a *coordinator* actor. The producer runs the `produceValues` method, thereby sending a sequence of values, wrapped in `Some` messages, to the

```
class Producer[T] {
  def produce(x: T) {
    coordinator ! Some(x)
  }
  val producer = actor {
    produceValues
    coordinator ! None
  }
  ...
}
```

```
val coordinator = actor {
  while (true) receive {
    case Next => receive {
      case x: Option[_] =>
        reply(x)
    }
  }
}
```

**Fig. 4.** Implementation of the producer and coordinator actors.

```
val coordinator = actor {
  loop { react {
    // ... as in Figure 4
  }}}
```

**Fig. 5.** Implementation of the coordinator actor using `react`.

coordinator. The sequence is terminated by a `None` message. The coordinator synchronizes requests from clients and values coming from the producer.

It is possible to economize one thread in the producer implementation. As shown in Fig. 5, this can be achieved by changing the call to `receive` in the coordinator actor into a call to `react` and using the `loop` combinator instead of the `while` loop. By calling `react` in its outer loop, the coordinator actor allows the scheduler to detach it from its worker thread when waiting for a `Next` message. This is desirable since the time between client requests might be arbitrarily long. By detaching the coordinator, the scheduler can re-use the worker thread and avoid creating a new one.

### *4.2. Pipes and asynchronous I/O*

In this example, a pair of processes exchanges data over a FIFO pipe. Such a pipe consists of a sink and a source channel that are used for writing to the pipe and reading from the pipe, respectively. The two processes communicate over the pipe as follows. One process starts out writing some data to the sink while the process at the other end reads it from the source. Once all of the data has been transmitted, the processes exchange roles and repeat this conversation.

To make this example more realistic and interesting at the same time, we use non-blocking I/O operations. A process that wants to write data has to register its interest in writing together with an event handler; when the I/O subsystem can guarantee that the next write operation will not block (*e.g.*, because of enough buffer space), it invokes this event handler.

The data should be processed concurrently; it is therefore not sufficient to put all the program logic into the event handlers that are registered with the I/O subsystem. Moreover, we assume that a process may issue blocking calls while processing the received data; processing the data inside an event handler could therefore block the entire I/O subsystem, which has to be avoided. Instead, the event handlers have to either notify a thread or an actor, or submit a task to a thread pool for execution.

In the following, we first discuss a solution that uses threads to represent the end points of a pipe. After that, we present an event-based implementation and compare it to the threaded version. Finally, we discuss a solution that uses Scala Actors. The solutions are compared with respect to synchronization and code structure.

We use a number of objects and methods whose definitions are omitted because they are not interesting for our discussion. First, processes have a reference `sink` to an I/O channel. The channel provides a `write` method that writes the contents of a buffer to the channel. The non-blocking I/O API is used as follows. The user implements an event handler which is a class with a single method that executes the I/O operation (and possibly other code). This event handler is registered with an I/O event dispatcher `disp` together with a channel; the dispatcher invokes an event handler when the corresponding (read or write) event occurs on the channel that the handler registered with. Each event handler is only registered until it has been invoked. Therefore, an event handler has to be registered with the dispatcher for each event that it should handle.

#### *4.2.1. Thread-based pipes*

In the first solution that we discuss, each end point of a pipe is implemented as a thread. Fig. 6 shows the essential parts of the implementation. The `run` method of the `Proc` class on the left-hand side shows the body of a process thread. First, we test whether the process should start off writing or reading. The `writeData` and `readData` operations are executed in the according order. After the writing process has written all its data, it has to synchronize with the reading process, so that the processes can safely exchange roles. This is necessary to avoid the situation where both processes have registered a handler for the same kind of I/O event. In this case, a process might wait indefinitely for an event because it was dispatched to the other process. We use a simple barrier of size 2 for synchronization: a thread invoking `await` on the `exch` barrier is blocked until a second thread invokes `exch.await`. The `writeData` method is shown on the right-hand side of Fig. 6

```
class Proc(write: Boolean,
           exch: Barrier)
  extends Thread {
  ...
  override def run() {
    if (write) writeData
    else readData
    exch.await
    if (write) readData
    else writeData
  } }
```

```
def writeData {
  fill(buf)
  disp.register(sink,
                writeHnd)
  var finished = false
  while (!finished) {
    dataReady.await
    dataReady.reset
    if (bytesWritten==32*1024)
      finished = true
    else {
      if (!buf.hasRemaining)
        fill(buf)
      disp.register(sink,
                    writeHnd)
    } } }
val writeHnd = new WriteHandler {
  def handleWrite() {
    bytesWritten +=
      sink.write(buf)
    dataReady.await
  } }
```

**Fig. 6.** Thread-based pipes.

```
class Proc(write: Boolean,
           pool: Executor)
{
  ...
  var last = false
  if (write) writeData
  else readData
  ...
  def writeData {
    fill(buf)
    disp.register(...)
  }
}
```

```
val task = new Runnable {
  def run() {
    if (bytesWritten==32*1024) {
      if (!last) {
        last = true; readData
      }
    } else {
      if (!buf.hasRemaining)
        fill(buf)
      disp.register(sink,
                    writeHnd)
    }
  }
}
val writeHnd = new WriteHandler {
  def handleWrite() {
    bytesWritten +=
      sink.write(buf)
    pool.execute(task)
  }
}
```

**Fig. 7.** Event-driven pipes.

(the `readData` method is analogous). First, it fills a buffer with data using the `fill` method. After that, it registers the `writeHnd` handler for write events on the `sink` with the I/O event dispatcher (`writeHnd` is discussed below). After that, the process enters a loop. First, it waits on the `dataReady` barrier until the write event handler has completed the next write operation. When the thread resumes, it first resets the `dataReady` barrier to the state where it has not been invoked, yet. The thread exits the loop when it has written 32 kB of data. Otherwise, it refills the buffer if it has been completed, and re-registers the event handler for the next write operation. The `writeHnd` event handler implements a single method `handleWrite` that writes data stored in `buf` to the `sink`, thereby counting the number of bytes written. After that, it notifies the concurrently running writer thread by invoking `await` on the `dataReady` barrier.

#### *4.2.2. Event-driven pipes*

Fig. 7 shows an event-driven version that is functionally equivalent to the previous threaded program. The process constructor which is the body of the `Proc` class shown on the left-hand side, again, tests whether the process starts out

```
class Proc(write: Boolean,
           other: Actor)
  extends Actor {
  ...
  def act() {
    { if (write)
        writeData
      else
        readData
    } andThen {
      other ! Exchange
      react {
        case Exchange =>
          if (write)
            readData
          else
            writeData }
    } }
}
```

```
def writeData {
  fill(buf)
  disp.register(sink,
                writeHnd)
  var bytesWritten = 0
  loopWhile(bytesWritten<32*1024)
    react { case Written(num) =>
      bytesWritten += num
      if (bytesWritten==32*1024)
        exit()
      else {
        if (!buf.hasRemaining)
          fill(buf)
        disp.register(sink,
                      writeHnd)
      } }
}
val writeHnd =
  new WriteHandler {
    def handleWrite() {
      val num =
        sink.write(buf)
      proc!Written(num)
    } }
```

**Fig. 8.** Actor-based pipes.

writing or reading. However, based on this test only *one* of the two I/O operations is called. The reason is that each I/O operation, such as `writeData`, registers an event handler with the I/O subsystem, and then returns immediately. The event handler for the second operation may only be installed when the last handler of the previous operation has run. Therefore, we have to decide inside the event handler of the write operation whether we want to read subsequently or not. The `last` field keeps track of this decision across all event handler invocations. If `last` is `false`, we invoke `readData` after `writeData` has finished (and *vice versa*); otherwise, the sequence of I/O operations is finished. The definition of an event handler for write events is shown on the right-hand side of Fig. 7 (read events are handled in an analogous manner). As before, the `writeHnd` handler implements the `handleWrite` method that writes data from `buf` to the `sink`, thereby counting the number of bytes written. To do the concurrent processing the handler submits a task to a thread pool for execution. The definition of this task is shown above. Inside the task we first test whether all data has been written; if so, the next I/O operation (in this case, `readData`) is invoked depending on the field `last` that we discussed previously. If the complete contents of `buf` has been written, it is refilled. Finally, the task re-registers the `writeHnd` handler to process the next event.

Compared to thread-based programming, the event-driven style obscures the control flow. For example, consider the `writeData` method. It does some work, and then registers an event handler. However, it is not clear what the operational effect of `writeData` is. Moreover, what happens after `writeData` has finished its actual work? To find out, we have to look inside the code of the registered event handler. This is still not sufficient, since also the submitted task influences the control flow. In summary, the program logic is implicit, and has to be recovered in a tedious way. Moreover, state has to be maintained across event handlers and tasks. In languages that do not support closures this often results in manual stack management [1].

#### *4.2.3. Actor-based pipes*

Fig. 8 shows the same program using Scala Actors. The `Proc` class extends the `Actor` trait; its `act` method specifies the behavior of an end point. The body of the `act` method is similar to the process body of the thread-based version. There are two important differences. First, control flow is specified using the `andThen` combinator. This is necessary since `writeData` (and `readData`) may suspend using `react`. Without using `andThen`, parts of the actor's continuation not included in the argument closure of the suspending `react` would be "lost". Basically, `andThen` appends the closure on its right-hand side to whatever continuation is saved during the execution of the closure on its left-hand side. Second, end point actors exchange messages to synchronize when switching roles from writing to reading (and *vice versa*). The `writeData` method is similar to its thread-based counterpart. The `while` loop is replaced by the `loopWhile` combinator since inside the loop the actor may suspend using `react`. At the beginning of each loop iteration the actor waits for a `Written` message signaling the completion of a write event handler. The number of bytes written is carried inside the message which allows us to make `bytesWritten` a local variable; in the thread-based version it is shared among the event handler and the process. The remainder of `writeData` is the same as in the threaded version. The `writeHnd` handler used in the actor-based program is similar to the thread-based version, except that it notifies its process using an asynchronous message send. Note that, in

general, the event handler is run on a thread which is different from the worker threads used by our library to execute actors (the I/O subsystem might use its own thread pool, for example). To make the presented scheme work, it is therefore crucial that arbitrary threads may send messages to actors.

**Conclusion.** Compared to the event-driven program, the actor-based version improves on the code structure in the same way as the thread-based version. Passing result values as part of messages makes synchronization slightly clearer and reduces the number of global variables compared to the thread-based program. However, in Section 7 we show that an event-based implementation of a benchmark version of the pipes example is much more efficient and scalable than a purely thread-based implementation. Our unified actor model allows us to implement the pipes example in a purely event-driven way while maintaining the clear code structure of an equivalent thread-based program.

## 5. Channels and selective communication

In the programming model that we have described so far, actors are the only entities that can send and receive messages. Moreover, the receive operation ensures *locality*, *i.e.*, only the owner of the mailbox can receive messages from it. Therefore, race conditions when accessing the mailbox are avoided by design. Types of messages are flexible: they are usually recovered through pattern matching. Ill-typed messages are ignored instead of raising compile-time or run-time errors. In this respect, our library implements a dynamically-typed embedded domain-specific language.

However, to take advantage of Scala's rich static type system, we need a way to permit strongly-typed communication among actors. For this, we use channels which are parameterized with the types of messages that can be sent to and received from it, respectively. Moreover, the visibility of channels can be restricted according to Scala's scoping rules. That way, communication between sub-components of a system can be hidden. We distinguish input channels from output channels. Actors are then treated as a special case of output channels:

```
trait Actor extends OutputChannel[Any] { ... }
```

The possibility for an actor to have multiple input channels raises the need to selectively communicate over these channels. Up until now, we have shown how to use `receive` to remove messages from an actor's mailbox. We have not yet shown how messages can be received from multiple input channels. Instead of adding a new construct, we generalize `receive` to work over multiple channels.

For example, a model of a component of an integrated circuit can receive values from both a control and a data channel using the following syntax:

```
receive {
  case DataCh ! data => ...
  case CtrlCh ! cmd => ...
}
```

## 6. Case study

In this section we show how our unified actor model addresses some of the challenges of programming web applications. In the process, we review event- and thread-based solutions to common problems, such as blocking I/O operations. Our goal is then to discuss potential benefits of our unified approach. Advanced web applications typically pose at least the following challenges to the programmer:

- *Blocking operations.* There is almost always some functionality that is implemented using blocking operations. Possible reasons are lack of suitable libraries (*e.g.*, for non-blocking socket I/O), or simply the fact that the application is built on top of a large code base that uses potentially blocking operations in some places. Typically, rewriting infrastructure code to use non-blocking operations is not an option.
- *Non-blocking operations.* On platforms such as the JVM, web application servers often provide some parts (if not all) of their functionality in the form of non-blocking APIs for efficiency. Examples are request handling, and asynchronous HTTP requests.
- *Race-free data structures.* Advanced web applications typically maintain user profiles for personalization. These profiles can be quite complex (some electronic shopping sites apparently track every item that a user visits). Moreover, a single user may be logged in on multiple machines, and issue many requests in parallel. This is common on web sites, such as those of electronic publishers, where single users represent whole organizations. It is therefore mandatory to ensure race-free accesses to a user's profile.

### 6.1. Thread-based approaches

VMs overlap computation and I/O by transparently switching among threads. Therefore, even if loading a user profile from disk blocks, only the current request is delayed. Non-blocking operations can be converted to blocking operations to support a threaded style of programming: after firing off a non-blocking operation, the current thread blocks until it is

notified by a completion event. However, threads do not come for free. On most mainstream VMs, the overhead of a large number of threads – including context switching and lock contention – can lead to serious performance degradation [44, 18]. Overuse of threads can be avoided by using bounded thread pools [30]. Shared resources such as user profiles have to be protected using synchronization operations. This is known to be particularly hard using shared-memory locks [32]. We also note that alternatives such as transactional memory [25,26], even though a clear improvement over locks, do not provide seamless support for I/O operations as of yet. Instead, most approaches require the use of compensation actions to revert the effects of I/O operations, which further complicate the code.

### 6.2. Event-based approaches

In an event-based model, the web application server generates events (network and I/O readiness, completion notifications etc.) that are processed by event handlers. A small number of threads (typically one per CPU) loop continuously removing events from a queue and dispatching them to registered handlers. Event handlers are required not to block since otherwise the event-dispatch loop could be blocked, which would freeze the whole application. Therefore, all operations that could potentially block, such as the user profile look-up, have to be transformed into non-blocking versions. Usually, this means executing them on a newly spawned thread, or on a thread pool, and installing an event handler that gets called when the operation is completed [38]. Usually, this style of programming entails an inversion of control that causes the code to lose its structure and maintainability [10,13].

### 6.3. Scala Actors

In our unified model, event-driven code can easily be wrapped to provide a more convenient interface that avoids inversion of control without spending an extra thread [24]. The basic idea is to decouple the thread that signals an event from the thread that handles it by sending a message that is buffered in an actor's mailbox. Messages sent to the same actor are processed atomically with respect to each other. Moreover, the programmer may explicitly specify in which order messages should be removed from its mailbox. Like threads, actors support blocking operations using implicit thread pooling as discussed in Section 3.3. Compared to a purely event-based approach, users are relieved from writing their own *ad hoc* thread pooling code. Since the internal thread pool can be global to the web application server, the thread pool controller can leverage more information for its decisions [44]. Finally, accesses to an actor's mailbox are race-free. Therefore, resources such as user profiles can be protected by modeling them as (thread-less) actors.

## 7. Experimental results

Optimizing performance across threads and events involves a number of non-trivial trade-offs. Therefore, we do not want to argue that our framework is better than event-based systems or thread-based systems or both. Instead, the following basic experiments show that the performance of our framework is comparable to those of both thread-based and event-based systems.

### 7.1. Message passing

In the first benchmark we measure 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. 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 3.00 GHz Intel Pentium 4 processor with 2048 MB memory, running Sun's Java HotSpot VM 1.5.0 under Linux 2.6.15 (SMP configuration). We set the JVM's maximum heap size to 1024 MB to provide for sufficient physical memory to avoid any disk activity. In each case we took the median of 5 runs. The execution times of three equivalent implementations written using (1) our actor library, (2) pure Java threads, and (3) SALSA (version 1.1.1), a Java-based actor language [40], respectively, are compared. We run the actor-based version using three different configurations. The first and second configurations fix the number of pool threads to 4 and 1, respectively. The third configuration runs without a thread pool; it is not equivalent with event-based actors, though, since we still create tasks that are put into a queue which is processed by a single thread. Therefore, the third configuration avoids the overhead of work stealing and thread pool resizing.

Fig. 9 shows the number of token passes per second (throughput) depending on the ring size. Note that both scales are logarithmic. For less than 1500 processes, pure Java threads are on average 3 times faster than actors that manage a pool with 4 threads. The overhead that stems from managing the thread pool contributes a factor of 2.6 (threads are on average 14% faster than actors without the runtime system overhead). Using more than a single pool thread increases throughput by about 18%; This performance gain is due to the two hyper threads of the CPU. For 1500 or more processes, the throughput of threads breaks in and reaches a minimum of about 23,000 tokens per second at 4500 processes. At this point, and up until 20,000 processes, throughput of the default actor configuration with 4 pool threads remains constant at about 39,000 tokens per seconds. The VM is unable to create a ring with 5500 threads as it runs out of heap memory. In contrast, using

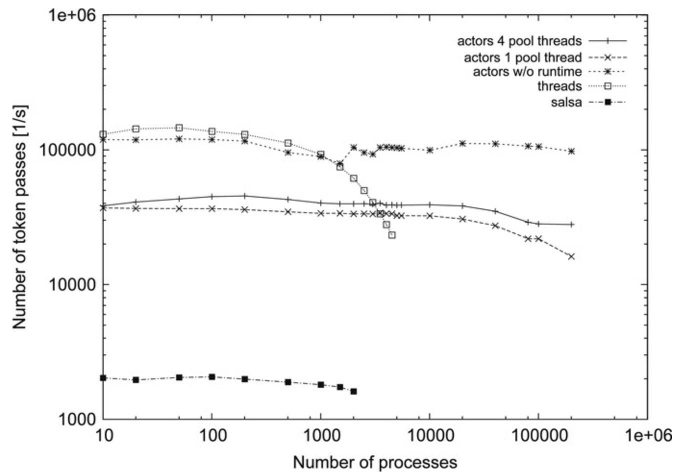


**Fig. 9.** Throughput (number of token passes per second) for a fixed number of 10 tokens.

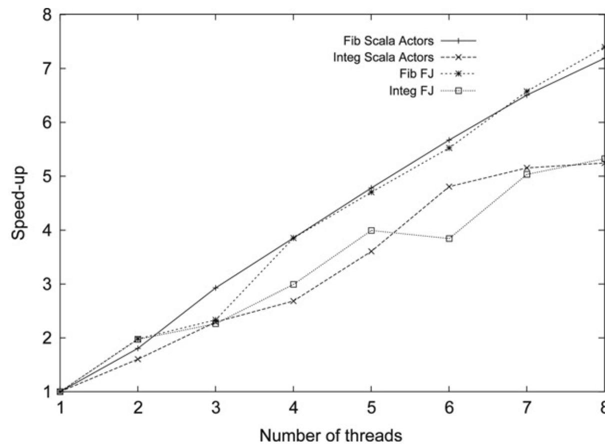


**Fig. 10.** Speed-up for Fibonacci and Integration benchmarks.

Scala Actors the ring can be operated with as many as 600,000 processes (since every queue is also an actor this amounts to 1,200,000 simultaneously active actors). Throughput of Scala Actors is on average over 20 times higher than that of SALSA. When creating 2500 processes using SALSA, the VM runs out of heap memory.

**Conclusion.** Purely event-based actors are competitive with native Java threads, even for a small number of processes. However, the overhead of our runtime system that involves work stealing and thread-pool monitoring/resizing can be significant when the number of processes is small. When running with only 10 processes, (multi-threaded) actors are 3.4 times slower than pure Java threads. In this benchmark threads use simple monitor-style synchronization, *i.e.*, `synchronized` methods where signaling is done between threads accessing the same monitor. Apparently, this style of synchronization is well-supported on modern JVMs. Only for a relatively large number of threads ($>=$ 1000) the overhead of context switching becomes significant (contention is low since each monitor is accessed by only two threads).

### 7.2. Multi-core scalability

In the second experiment, we are interested in the speed-up that is gained by adding processor cores to a system. The following tests were run on a multi-processor with 4 dual-core AMD Opteron 64-Bit processors (2.8 GHz each) with 16 GB memory, running Sun's Java HotSpot 64-Bit Server VM 1.5.0 under Linux 2.6.16. In each case we took the median of 5 runs. We ran direct translations of the Fibonacci (Fib) and Gaussian integration (Integ) programs distributed with Lea's high-performance fork/join framework for Java (FJ) [31]. The speed-ups as shown in Fig. 10 are linear as expected since the programs run almost entirely in parallel.

### 7.3. I/O performance

The following benchmark scenario is similar to those used in the evaluation of other thread implementations [42,33]. We aim to simulate the effects of a large number of mostly-idle client connections. For this purpose, we create a large number

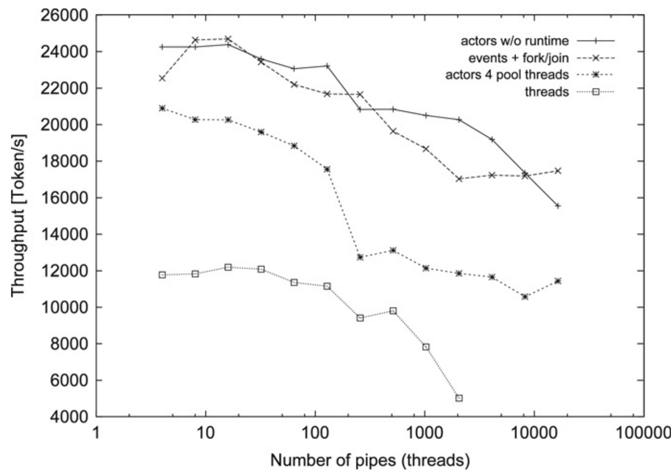


**Fig. 11.** Network scalability benchmark on single-processor.

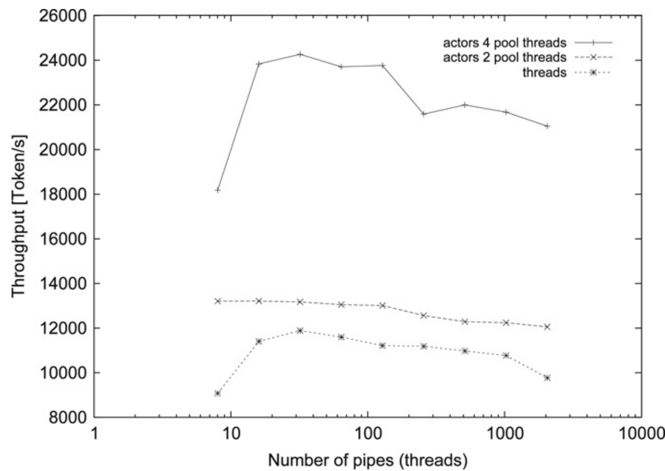


**Fig. 12.** Network scalability benchmark on multi-processor.

of FIFO pipes and measure the throughput of concurrently passing a number of tokens through them. If the number of pipes is less than 128, the number of tokens is one quarter of the number of pipes; otherwise, exactly 128 tokens are passed concurrently. The idle end points are used to model slow client links. After a token has been passed from one process to another, the processes at the two end points of the pipe exchange roles, and repeat this conversation.

Fig. 11 shows the performance of implementations based on events, threads, and actors under load. The programs used to obtain these results are slightly extended versions of those discussed in Section 4.2. They were run on a 1.60 GHz Intel Pentium M with 512 MB memory, running Sun's Java HotSpot Server VM 1.5.0 under Linux 2.6.17. In each case, we took the average of 5 runs. In the first version, end points are implemented as actors that are run on a single thread without the overhead of the runtime system that we discussed in Section 3.3. The second version uses a purely event-based implementation; concurrent tasks are run on a lightweight fork/join execution environment [31]. The third program is basically the same as the first one, except that actors run in "unified mode" using the runtime system which manages 4 pool threads by default. The overhead of unified actors compared to purely event-based actors ranges between 8% (4 pipes) and 70% (256 pipes). In the last version, end points are implemented as standard JVM threads. In this example, events are on average 120% faster than threads, unified actors are on average 66% faster.

Fig. 12 shows the performance of the actor-based and thread-based programs when run on a multi-processor machine with 4 hardware threads (2 hyper-threaded Intel Xeon 3.06 GHz with 2 GB memory, running Linux 2.6.11 SMP). When the number of non-idle threads increases from 2 to 4 in the thread-based version, throughput increases by about 26%. After that the additional threads basically induce only additional overhead. In the version that uses our unified model, growing the thread pool from 2 to 4 workers increases throughput on average by 74%. Even though the overhead of the runtime system for our unified actor model can be as high as 70% on a single CPU, this overhead is quickly amortized when the program is run on a multi-processor system.

**Conclusion.** The synchronization patterns found in the network scalability benchmark are more likely to be found in real-world programs than those of the previous benchmarks. Simple monitor-style synchronization is not sufficient; instead,

barriers and blocking queues are used to synchronize end points and event handlers. Moreover, threads are likely to be blocked more often, waiting for I/O events. This situation enables actors to amortize their overhead and to reverse the result of the message passing benchmark. For 64 pipes or less, the throughput of actors is on average 69% higher than the throughput of threads. Presumably, thread notification (`notify` on the JVM) is relatively expensive. On a multi-processor system with 4 hardware threads, only actors enable a significant speed-up in this benchmark.

## 8. Related and future work

Lauer and Needham [29] note in their seminal work that threads and events are dual to each other. They suggest that any choice of either one of them should therefore be based on the underlying platform. Almost two decades later, Ousterhout [37] argues that threads are a bad idea not only because they often perform poorly, but also because they are hard to use. More recently, von Behren and others [41] point out that even though event-driven programs often outperform equivalent threaded programs, they are too difficult to write. The two main reasons are: first, the interactive logic of a program is fragmented across multiple event handlers (or classes, as in the state design pattern [21]). Second, control flow among handlers is expressed implicitly through manipulation of shared state [10]. In the Capriccio system [42], static analysis and compiler techniques are employed to transform a threaded program into a cooperatively-scheduled event-driven program with the same behavior.

There are several other approaches that avoid the above control inversion. However, they have either limited scalability,[5] or they lack support of blocking operations. Termite Scheme [23] integrates Erlang's programming model into Scheme. Scheme's first-class continuations are exploited to express process migration. However, their system apparently does not support multiple processor cores. All published benchmarks were run in a single-core setting. Responders [10] provide an event-loop abstraction as a Java language extension. Since their implementation spends a VM thread per event-loop, scalability is limited on standard JVMs. SALSA [40] is a Java-based actor language that has a similar limitation (each actor runs on its own thread). In addition, message passing performance suffers from the overhead of reflective method calls. Timber [5] is an object-oriented and functional programming language designed for real-time embedded systems. It offers message passing primitives for both synchronous and asynchronous communication between concurrent *reactive objects*. In contrast to our programming model, reactive objects are not allowed to call operations that might block indefinitely. Frugal objects [22] (FROBs) are distributed reactive objects that communicate through typed events. FROBs are basically actors with an event-based computation model. Similar to reactive objects in Timber, FROBs may not call blocking operations.

In languages like Haskell and Scala, the continuation monad can also be used to implement lightweight concurrency [11]. In fact, it is possible to define a monadic interface for the actors that we present in this paper; however, a thorough discussion is beyond the scope of this paper. Li and Zdancewic [33] use the continuation monad to combine events and threads in a Haskell-based system for writing high-performance network services. However, they require blocking system calls to be wrapped in non-blocking operations. In our library actors subsume threads, which makes this wrapping unnecessary; essentially, the programmer is relieved from writing custom thread-pooling code.

The actor model has also been integrated into various Smalltalk systems. Actalk [7] is an actor library for Smalltalk-80 that does not support multiple processor cores. Actra [39] extends the Smalltalk/V VM to provide lightweight processes. In contrast, we implement lightweight actors on unmodified VMs.

Other concurrent programming languages and systems also use actors or actor-like abstractions. ProActive [8] is a middleware for programming distributed Grid applications. Its main abstractions are a form of deterministic active objects [9] that communicate using asynchronous method calls and futures. ProActive allows distributing active objects on a large number of machines in a Grid. In contrast, Scala Actors address the trade-off between threads and events for achieving highly scalable multithreading on a single machine. AmbientTalk [16] provides actors based on communicating event loops [35]. AmbientTalk implements a protocol mapping [15] that allows native (Java) threads to interact with actors while preserving non-blocking communication among event-loops. However, the mapping relies on the fact that each actor is always associated with its own VM thread, whereas Scala's actors can be thread-less. While AmbientTalk offers a bridge between two distinct concurrency models, Scala Actors provide a single unified concurrency model.

In Section 7 we show that our actor implementation scales to a number of actors that is two orders of magnitude larger than what purely thread-based systems such as SALSA support. Moreover, results suggest that our model scales with the number of processor cores in a system. Our unified actor model provides seamless support for blocking operations. Therefore, existing thread-blocking APIs do not have to be wrapped in non-blocking operations. Unlike approaches such as Actra our implementation provides lightweight actor abstractions on unmodified (Java) VMs.

Our library was inspired to a large extent by Erlang's elegant programming model. Erlang [4] is a dynamically-typed functional programming language designed for programming real-time control systems. The combination of lightweight isolated processes, asynchronous message passing with pattern matching, and controlled error propagation has been proven to be very effective [3,36]. One of our main contributions lies in the integration of Erlang's programming model into a full-fledged object-oriented and functional language. Moreover, by lifting compiler magic into library code we achieve

[5] We use the term scalability to refer to the number of concurrent processes; in some related domains, *e.g.*, distributed programming, scalability is often measured in number of machines.

compatibility with standard, unmodified JVMs. To Erlang's programming model we add new forms of composition as well as *channels*, which permit strongly-typed and secure inter-actor communication.

The idea to implement lightweight concurrent processes using continuations has been explored many times [43,27,12]. However, none of the existing techniques are applicable to VMs such as the JVM because (1) the security model restricts accessing the run-time stack, and (2) heap-based stacks break interoperability with existing code.

The approach used to implement thread management in the Mach 3.0 kernel [17] is at least conceptually similar to ours. When a thread blocks in the kernel, either it preserves its register state and stack and resumes by restoring this state, or it preserves a pointer to a continuation function that is called when the thread is resumed. Instead of function pointers we use closures that automatically lift referenced stack variables on the heap avoiding explicit state management in many cases. Moreover, we save a richer continuation in form of a *partial function*. A partial function allows testing whether it is defined for a given value. We use this test to decide whether an actor should be resumed upon receiving a message.

There is a rich body of work on building fast web servers, using events or a combination of events and threads (for example SEDA [44]). However, a comprehensive discussion of this work is beyond the scope of this paper.

**Ongoing and future work.** The Scala Actors library includes a runtime system that provides basic support for remote (*i.e.*, inter-VM) actor communication. In ongoing work we are extending the framework with remote actor references that support volatile connections, similar to ambient references [14]. Integrating abstractions for fault-tolerant distributed programming (*e.g.*, [20,46]) into Scala Actors is an interesting area for future work.

## 9. Conclusion

In this paper we have shown how thread-based and event-based models of concurrency can be unified. The main idea of our unified programming model is an actor abstraction that provides two kinds of operations for receiving messages. The `receive` operation retains the complete call stack of the receiver while waiting for a message, while the `react` operation retains only a continuation closure. The first form of suspension corresponds to thread-based, the second form to event-based programming. As a result, our unified concurrency model combines the benefits of threads and events while abstracting commonalities. The presented ideas have been implemented in the Scala Actors library which provides actor-based concurrency supporting high-level communication through messages and pattern matching. We believe that our work closes an important gap between message-passing concurrency and popular VM platforms.

## References


[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur, Cooperative task management without manual stack management, in: Proc. USENIX, USENIX, 2002, pp. 289–302.
[2] Gul A. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, Massachusetts, 1986.
[3] Joe Armstrong, Erlang — a survey of the language and its industrial applications, in: Proc. INAP, October 1996, pp. 16–18.
[4] Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams, Concurrent Programming in Erlang, second edition, Prentice-Hall, 1996.
[5] A. Black, M. Carlsson, M. Jones, R. Kieburtz, J. Nordlander, Timber: A Programming Language for Real-Time Embedded Systems, 2002.
[6] R. Blumofe, et al., Cilk: An efficient multithreaded runtime system, Journal of Parallel and Distributed Computing 37 (1) (1996) 55–69.
[7] Jean-Pierre Briot, Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment, in: Proc. ECOOP, 1989, pp. 109–129.
[8] D. Caromel, C. Delbé, A. di Costanzo, M. Leyton, ProActive: An integrated platform for programming and running applications on grids and P2P systems, Computational Methods in Science and Technology 12 (1) (2006) 69–77.
[9] Denis Caromel, Ludovic Henrio, Bernard Paul Serpette, Asynchronous and deterministic objects, in: Proc. POPL, ACM, 2004, pp. 123–134.
[10] Brian Chin, Todd D. Millstein, Responders: Language support for interactive applications, in: Proc. ECOOP, in: LNCS, vol. 4067, Springer, July 2006, pp. 255–278.
[11] Koen Claessen, A poor man's concurrency monad, Journal of Functional Programming 9 (3) (1999) 313–323.
[12] Eric Cooper, Gregory Morrisett, Adding threads to standard ML, Report CMU-CS-90-186, Carnegie-Mellon University, December 1990.
[13] Ryan Cunningham, Eddie Kohler, Making events less slippery with eel, in: Proc. HotOS, USENIX, June 2005.
[14] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, Elisa Gonzalez Boix, Theo D'Hondt, Wolfgang De Meuter, Ambient references: Addressing objects in mobile networks, in: OOPSLA Companion, ACM, 2006, pp. 986–997.
[15] Tom Van Cutsem, Stijn Mostinckx, Wolfgang De Meuter, Linguistic symbiosis between actors and threads, in: Proc. ICDL, ACM, 2007, pp. 222–248.
[16] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, Wolfgang De Meuter, Ambient-oriented programming in AmbientTalk, in: Proc. ECOOP, in: LNCS, vol. 4067, 2006, pp. 230–254.
[17] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, Randall W. Dean, Using continuations to implement thread management and communication in operating systems, Operating Systems Review 25 (5) (1991) 122–136.
[18] Adam Dunkels, Björn Grönvall, Thiemo Voigt, Contiki — A lightweight and flexible operating system for tiny networked sensors, in: Proc. LCN, 2004, pp. 455–462.
[19] Burak Emir, Martin Odersky, John Williams, Matching objects with patterns, in: Erik Ernst (Ed.), Proc. ECOOP, in: LNCS, vol. 4609, Springer, 2007, pp. 273–298.
[20] John Field, Carlos A. Varela, Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments, in: Proc. POPL, ACM, 2005, pp. 195–208.
[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1995.
[22] Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, Jesper Spring, Frugal mobile objects, Technical Report, EPFL, 2005.
[23] Guillaume Germain, Marc Feeley, Stefan Monnier, Concurrency oriented programming in termite scheme, in: Proc. Workshop on Scheme and Functional Programming, September 2006.
[24] Philipp Haller, Martin Odersky, Event-based programming without inversion of control, in: Proc. JMLC, in: LNCS, vol. 4228, Springer, September 2006, pp. 4–22.
[25] Tim Harris, Keir Fraser, Language support for lightweight transactions, in: Proc. OOPSLA, 2003, pp. 388–402.
[26] Tim Harris, Simon Marlow, Simon L. Peyton Jones, Maurice Herlihy, Composable memory transactions, in: Proc. PPOPP, ACM, June 2005, pp. 48–60.
[27] Christopher T. Haynes, Daniel P. Friedman, Engines build process abstractions, in: Symp. Lisp and Functional Programming, ACM, August 1984, pp. 18–24.

[28] Carl Hewitt, Peter Bishop, Richard Steiger, A universal modular ACTOR formalism for artificial intelligence, in: IJCAI, 1973, pp. 235–245.
[29] Hugh C. Lauer, Roger M. Needham, On the duality of operating system structures, Operating Systems Review 13 (2) (1979) 3–19.
[30] Doug Lea, Concurrent Programming in Java, Addison Wesley, 1996.
[31] Doug Lea, A java fork/join framework, in: Java Grande, 2000, pp. 36–43.
[32] Edward A. Lee, The problem with threads. Technical Report UCB/EECS-2006-1, University of California, Berkeley, January 2006.
[33] Peng Li, Steve Zdancewic, Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives, in: Proc. PLDI, ACM, 2007, pp. 189–199.
[34] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1996.
[35] Mark S. Miller, E. Dean Tribble, Jonathan Shapiro, Concurrency among strangers — programming in E as plan coordination, in: Proc. TGC 2005, in: LNCS, vol. 3705, Springer-Verlag, 2005, pp. 195–229.
[36] J.H. Nyström, Philip W. Trinder, David J. King, Evaluating distributed functional languages for telecommunications software, in: Proc. Workshop on Erlang, ACM, August 2003, pp. 1–7.
[37] John Ousterhout, Why threads are A bad idea (for most purposes), Invited talk at USENIX, January 1996.
[38] Vivek S. Pai, Peter Druschel, Willy Zwaenepoel, Flash: An efficient and portable Web server, in: Proc. USENIX, June 1999, pp. 199–212.
[39] D.A. Thomas, W.R. Lalonde, J. Duimovich, M. Wilson, J. McAffer, B. Berry, Actra: A multitasking/multiprocessing smalltalk, ACM SIGPLAN Notices 24 (4) (1989) 87–90.
[40] Carlos Varela, Gul Agha, Programming dynamically reconfigurable open systems with SALSA, ACM SIGPLAN Notices 36 (12) (2001) 20–34.
[41] J. Robert von Behren, Jeremy Condit, Eric A. Brewer, Why events are a bad idea (for high-concurrency servers), in: Proc. Hot OS, USENIX, May 2003, pp. 19–24.
[42] J. Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric A. Brewer, Capriccio: Scalable threads for internet services, in: Proc. SOSP, 2003, pp. 268–281.
[43] Mitchell Wand, Continuation-based multiprocessing, in: LISP Conference, 1980, pp. 19–28.
[44] Matt Welsh, David E. Culler, Eric A. Brewer, SEDA: An architecture for well-conditioned, scalable internet services, in: Proc. SOSP, 2001, pp. 230–243.
[45] Akinori Yonezawa, Jean-Pierre Briot, Etsuya Shibayama, Object-oriented concurrent programming in ABCL/1, in: Proc. OOPSLA, 1986, pp. 258–268.
[46] Lukasz Ziarek, Philip Schatz, Suresh Jagannathan, Stabilizers: A modular checkpointing abstraction for concurrent functional programs, in: Proc. ICFP, ACM, 2006, pp. 136–147.