Actors that Unify Threads and Events

Philipp Haller, EPFL

joint work with Martin Odersky, EPFL
Implementing Concurrent Processes

1. Thread-based
   - Behavior = body of designated method
   - Execution state = thread stack
   - *Examples*: Java threads, POSIX threads

2. Event-based
   - Behavior = set of event handlers
   - Execution state = object shared by handlers
   - *Examples*: Java Swing, TinyOS
Threads
[Ousterhout96]

- Support multiple hardware cores (Good)
- Behavior = sequential program (Good)
- Heavyweight (Bad)
  - High memory consumption
    - Pre-allocated stacks
  - Lock contention bottleneck
- Synchronization using locks error-prone, not composable (Bad)
Events: Remedy?  
[vonBehren03]

- Lightweight (**Good**)
  - Multiple events interleaved on single thread
  - Low memory consumption
- Automatic synchronization (**Good**)
- No hardware support (**Bad**)
- Inversion of control (**Bad**)
  - Behavior != sequential program
Rest of this Talk

- Programming with Actors in Scala
- Unifying Threads and Events
  - Programming Model
  - Lightweight Execution Environment
- Composing Actors
- Selective Communication
- Experimental Results
Actors

- Model of concurrent processes introduced by Hewitt and Agha
- Upon reception of a message, an actor may
  - send messages to other actors
  - create new actors
  - change its behavior/state
- Most popular implementation: Erlang
- But: No widespread adoption in languages for standard VMs (e.g. JVM, CLR)
Actors in Scala

- Two principle operations (adopted from Erlang)
  ```scala
  actor ! message  // message send
  
  receive {
    case msgpat_1 => action_1
    ... 
    case msgpat_n => action_n
  }
  ```

- Send is asynchronous; messages are buffered in actor's *mailbox*

- `receive` waits for message that matches any of the patterns `msgpat_i`
Example: Producers

• Producers act like iterators, generate values *concurrently* with consumer:

```scala
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) } }
```

• Methods `produceValues` *(abstract)* and `produce` inherited from class `Producer`
Implementing Producers

Producers are implemented in terms of two actors.

1. The *producer* actor runs `produceValues`:

```scala
abstract class Producer[T] extends Iterator[T] {
  def produceValues: Unit
  def produce(x: T) {
    coordinator ! Some(x)
  }
  private val producer = actor {
    produceValues; coordinator ! None
  }
}
```
The *coordinator* actor synchronizes requests from clients and values from the producer.

```scala
val coordinator = actor {
  while (true) {
    receive {
      case Next =>
        receive {
          case x: Option[_] => client ! x
        }
    }
  }
}
```
Lightweight Execution Environment

Actors (many)

Task queue

Worker threads (few)
Creating Actors

```scala
actor {
  // body
}

// closure => T3

T1
T2
T3
```
Thread Mode: `receive`

1. Scan messages in mailbox
2. If no message matches any of the patterns, `suspend worker thread`
3. Otherwise, process first matching message

Actor remains active
Event Mode: react

1. Register message handler
2. Become passive (temporarily)

Actor becomes inactive

```scala
... react {
    case Msg(x) =>
        // handle msg
}
```
Suspend in Event Mode

Task Ti:

```scala
... react {
  case Msg(x) => // handle msg
}
```

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

Exception:

1. Unwinds stack of actor/worker thread
2. Finishes current task

// do nothing
Resume in Event Mode

Actor \( a \) waits for

\[
\{
    \text{case } \text{Msg}(x) \Rightarrow \text{\// handle msg}
\}
\]

Task \( Ti \):

\[
\ldots
\]

\[
a ! \text{Msg}(42)
\]

\[
\ldots
\]

\[
\text{Ti} + 2
\]

wt executes \( Ti \)

\[
\text{Ti} + 1
\]

\[
\text{Ti} + 2: \text{apply(Msg(42))}
\]
Thread Pool Resizing

A suspended in receive
{ case Msg(x) =>... }

B suspended in library (e.g. wait())

T1: ...
A ! Msg(x)
...

Executing T1 would unblock wt1!
Implementing Producers (3)

Economize one thread in Producer by changing `receive` in the coordinator actor to `react`

```scala
val coordinator = actor {
  loop {
    react {
      case Next =>
        react {
          case x: Option[_] => client ! x
        }
    }
  }
}
```
Composing Actors

- Composing event-driven code non-trivial
  - `react` may *unwind stack* at any point
  - Normal sequencing does not work
- Composition operators for common uses
  - `a andThen b` runs `a` followed by `b`
  - `def loop(body: => Unit) = body andThen loop(body)`
Channels

trait OutputChannel[-Msg] {
  def !(msg: Msg): Unit
  def forward(msg: Msg): Unit
}

trait InputChannel[+Msg] {
  receive[R](f: PartialFunction[Msg, R]): R
  react(f: PartialFunction[Msg, Unit]): Nothing
  ...
}

class Channel[Msg] extends InputChannel[Msg]
  with OutputChannel[Msg]

trait Actor extends OutputChannel[Any] {
  ...
}
Selective Communication

- Generalize receive/react:

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

- Composing alternatives using `orElse`:

```scala
receive {
  case DataCh ! data => ... 
  case CtrlCh ! cmd => ... 
} orElse super.reactions
```
Experimental Results

Number of token passes per second in ring of processes.
Conclusion

- Threads and events can be unified under an abstraction of actors
- `receive/react` allows programmers to trade-off efficiency for flexibility
- Implemented in **Scala Actors** library (http://www.scala-lang.org/)
- Real-world usage: *lift* web framework
Thread Pool Resizing (2)
(cf. SEDA [Welsh01])

- Sample task queue
- Add thread when queue length exceeds threshold (up to max. number of threads)
- Remove thread when idle for specified period of time
Experimental Results (2)

- Micro benchmarks run on 4-way dual-core Opteron machine (8 cores total)
- Compared to Doug Lea's FJTask framework for Java
def httpFetch(queryURL: String) = {
  val req = new XmlHttprequest
  req.addOnReadyStateChangeListener(new PropertyChangeListener() {
    override def propertyChange(evt: PropertyChangeEvent) {
      if (evt.getNewValue() == ReadyState.LOADED) {
        val response = req.getResponseText()
        httpParseResponse(response)
      }
    }
  })
  try {
    req.open(Method.GET, new URL(queryURL))
    req.send()
  } catch {
    case e: Throwable => ...
  }
}

Typical asynchronous HTTP document fetch
Inversion of Control

```
val res = evt...
```
Problems of Inversion of Control

• Hard to understand control-flow
  – reconstruct entire call-graph

• Manual stack management
  – handler code *not* defined where event is handled
  – local variables, parameters etc. not accessible

• Managing resources (files, sockets) becomes even harder
  – often long-lived, used in several event handlers
  – when is a missing close() a leak?
Blocking-style Code

Client

Server

val res = evt...

Message sends
Concurrency is Indispensable

- Software is concurrent
  - Interactive applications
  - Web services
  - Distributed software
- Hardware is concurrent
  - Hyper-threading
  - Multi-cores, Many-cores
  - Grid computing