Programming Paradigms for Concurrency
Lecture 10 – The Actor Paradigm

Based on a course on
Principles of Reactive Programming
by Martin Odersky, Erik Meijer, Roland Kuhn

Modified by
Thomas Wies
New York University
Limitations of Shared Memory Concurrency

• locks are the “goto statements” of concurrency
  
  locks do not compose; reasoning about programs becomes (even more) difficult

• shared memory typically implies **physically shared memory**
  
  programs do not *scale out* to distributed architectures

• OS threads are **resource-hungry** and context-switching is expensive
  
  number of threads = number of available cores
  ≠ number of logical tasks
Message Passing Concurrency

• no shared memory (in its pure form)
  + some classes of concurrency errors avoided by design
  + natural programming model for distributed architectures
  - sometimes less efficient on shared-memory architectures: data must be copied before sending

• all synchronization between processes is explicit
  + reasoning about program behavior is simplified
  - “it’s harder to parallelize a sequential program using MP”

• higher level of abstraction
  + decouple computation tasks from physical threads
    -> event-driven programming
Message Passing Paradigms

Two important categories of MP paradigms:

1. Actor or agent-based paradigms
   - unique receivers: messages are sent directly from one process to another

2. Channel-based paradigms
   - multiple receivers: messages are sent to channels that are shared between processes

We will focus on the actor paradigm.
The Actor Paradigm

Actors are the object-oriented approach to concurrency

“everything is an actor”

actor = object + logical thread
A Brief History of Actors

- Hewitt, Bishop, Steiger 1973: actor model
- Agha 1986: actor languages and semantics
- Armstrong et al. 1990s: Erlang language
- Haller, Odersky 2006: Scala actors
- Boner 2009: Akka actors
The Akka Actor Trait

type Receive = PartialFunction[Any,Unit]

trait Actor {
   def receive: Receive
   ...
}

The Actor type describes the behavior of an actor, i.e., how it reacts to received messages.
A Simple Actor

class Counter extends Actor {
  var count = 0
  def receive = {
    case "incr" => count += 1
  }
}

Use pattern matching to dispatch incoming messages
Sending Messages

class Counter extends Actor {
    var count = 0
    def receive = {
        case "incr" => count += 1
        case ("get", customer: ActorRef) =>
            customer ! count
    }
}
trait Actor {
  implicit val self: ActorRef
  def sender: ActorRef
  ...
}

abstract class ActorRef {
  def !(msg: Any)(implicit sender: ActorRef = Actor.noSender): Unit
  def tell(msg: Any, sender: ActorRef) = this.!(msg)(sender)
  ...
}
Using sender

class Counter extends Actor {
    var count = 0
    def receive = {
        case "incr" => count += 1
        case "get" => sender ! count
    }
}
Changing an Actor’s Behavior

class ActorContext {
   def become(behavior: Receive, discardOld: Boolean = true): Unit
   def unbecome(): Unit
   ...
}

trait Actor {
   implicit val context: ActorContext
   ...
}
Changing an Actor’s Behavior

class Counter extends Actor {
    def counter(n: Int) = {
        case "incr" => context.become(counter(n + 1))
        case "get" => sender ! n
    }
    def receive = counter(0)
}
Important Lessons to Remember

• Prefer context.become for different behaviors, with data local to each behavior
Creating and Stopping Actors

class ActorContext {
    def actorOf(p: Props, name: String): ActorRef
    def stop(a: ActorRef): Unit
    ...
}

trait Actor {
    val self: ActorRef
    ...
}

Actors are created by other actors. Typically, stop is called with self as argument.
A Simple Actor Application

class Main extends Actor {
    val counter = context.actorOf(Props[Counter], "counter")

    counter ! "incr"
    counter ! "incr"
    counter ! "incr"
    counter ! "get"

    def receive = {
        case count: Int =>
            println(s"count was $count")
            context.stop(self)
    }
}
Internal Computation of Actors

- actors can
  - react to incoming messages
  - dynamically create other actors
  - send messages to other actors
  - dynamically change behavior
Evaluation Order of Actor Computations

• Actor-internal computation is single-threaded
  – messages are received sequentially
  – behavior change is effective before next message is processed
  – processing one message is an atomic operation

• Sending a message is similar to calling a synchronized method, except that it is non-blocking
Actors Encapsulate State

• no direct access to an actor’s internal state
• state is accessed indirectly through message passing
• message passing is
  – asynchronous
  – buffered (FIFO)
  – over unique-receiver channels (mailboxes)
  – restricted to “known” actor references
    • self
    • actors this created
    • references this received in messages
object BankAccount {
    case class Deposit(amount: BigInt) {
        require(amount > 0)
    }
    case class Withdraw(amount: BigInt) {
        require(amount > 0)
    }
    case object Done
    case object Failed
}
class BankAccount extends Actor {
  import BankAccount._

  var balance = BigInt(0)

  def receive = {
    case Deposit(amount) =>
      balance += amount; sender ! Done
    case Withdraw(amount) if amount <= balance =>
      balance -= amount; sender ! Done
    case _ => sender ! Failed
  }
}
object WireTransfer {
    case class Transfer(from: ActorRef,
                         to: ActorRef, amount: BigInt)
    case object Done
    case object Failed
}
class WireTransfer extends Actor {
  import WireTransfer._

  def receive = {
    case Transfer(from, to, amount) =>
      from ! BankAccount.Withdraw(amount)
      context.become(awaitWithdraw(to, amount, sender))
  }

  def awaitWithdraw ...
class WireTransfer extends Actor {

    def awaitWithdraw(to: ActorRef, amount: BigInt, client: ActorRef): Receive = {
        case BankAccount.Done =>
            to ! BankAccount.Deposit(amount)
            context.become(awaitDeposit(client))
        case BankAccount.Failed =>
            client ! Failed
            context.stop(self)
    }

    def awaitDeposit ...
Wire Transfer

class WireTransfer extends Actor {

    def awaitDeposit(client: ActorRef): Receive = {
        case BankAccount.Done =>
            client ! Done
            context.stop(self)
    }
}
A Simple Web Crawler

**Goal:** write a simple web crawler that

- makes an HTTP request for a given URL
- parses the returned HTTP body to collect all links to other URLs
- recursively follows those links up to a given depth
- all links encountered should be returned.
Basic Design

Client ➔ Receptionist
- Get(url)
- Result(url, urls)

Receptionist ➔ Controller
- Crawl(url, d)
- Result(urls)

Controller ➔ Getter
- Get(url, d)
- Link(url, d)
- Done

Getter ➔ Getter
- Getter

getter
- Getter
- Getter
- Getter
Plan of Action

• write web client which asynchronously turns a URL into an HTTP body (based on com.ning.http.client)

• write a Getter actor for processing the body

• write a Controller which spawns Getters for all links encountered

• write a Receptionist managing one Controller per request.
The Web Client

val client = new AsyncHttpClient

def get(url: String): String = {
  val response = client.prepareGet(url).execute().get
  if (response.getStatusCode < 400)
    response.getResponseBodyExcerpt(131072)
  else throw BadStatus(response.getStatusCode)
}

Blocks the caller until the web server has replied
⇒ actor is deaf to other requests, e.g., cancellation
⇒ priority inversion: current thread is blocked
A short Digression to Monads

- Monads allow you to encapsulate side-effects such as
  - state mutation
  - IO
  - exceptions
  - latency

- We look at two of Scala's monads:
  - Try: encapsulates exceptions
  - Future: encapsulates exceptions and latency
Implicit Exception Handling

def divide: Int =
  val dividend =
    Console.readLine("Enter an Int to divide:\n").toInt
  val divisor =
    Console.readLine("Enter an Int to divide by:\n").toInt
  dividend/divisor

What can go wrong here?
The Try Class

sealed abstract class Try[T] {
  abstract def isSuccess: Boolean
  abstract def isFailure: Boolean
  abstract def get: T
  abstract def flatMap[S](f: T => Try[S]): Try[S]
  abstract def map[S](f: T => S): Try[S]
  ...
}

case class Success[T](elem: T) extends Try[T]
case class Failure[T](t: Throwable) extends Try[T]
Try's Companion Object

object Try {
  def apply[T](body: => T) {
    try {
      Success(body)
    }
    catch {
      t => Failed(t)
    }
  }
}

Now we can wrap the result of a computation in a Try value:
val dividend =
  Try(Console.readLine("Enter an Int to divide:\n").toInt)
Implicit Exception Handling

import scala.util.{Try, Success, Failure}

def divide: Int =
  val dividend =
    Try(Console.readLine("Enter an Int to divide:
\n").toInt)
val divisor =
  Try(Console.readLine("Enter an Int to divide by:
\n").toInt)
val result =
Futures

A Future is an object holding a value which may become available at some point.

• This value is usually the result of some other computation.

• If the computation has completed with a value or with an exception, then the Future is **completed**.

• A Future can only be completed once.

Think of a Future as an asynchronous version of Try
The Future Trait

```scala
trait Awaitable[T] {
  abstract def ready(atMost: Duration): Unit
  abstract def result(atMost: Duration): T
}

trait Future[T] extends Awaitable[T] {
  abstract def onComplete[U](f: (Try[T]) => U)
      (implicit executor: ExecutionContext): Unit
  abstract def flatMap[S](f: T => Future[S]): Future[S]
  abstract def map[S](f: T => S): Future[S]
...
}

object Future {
  def apply[T](body: => T)
      (implicit executor: ExecutionContext): Future[T]
}
```
import scala.concurrent._
import ExecutionContext.Implicits.global
...
val usdQuote = Future { connection.getCurrentValue(USD) }
val eurQuote = Future { connection.getCurrentValue(EUR) }
val purchase = for {
  usd <- usdQuote
  eur <- eurQuote
  if isProfitable(usd, eur)
} yield connection.buy(amount, eur)

purchase onSuccess {
  case _ => println(s"Purchased EUR $amount")
}
import scala.concurrent.{Future, Promise}
import scala.concurrent.ExecutionContext.Implicits.global

val p = promise[T]
val producer = Future {
  val r = produceSomething()
  p.success(r)
  continueDoingSomethingUnrelated()
}
val f = p.future
val consumer = Future {
  startDoingSomething()
  f onSuccess {
    case r => doSomethingWithResult(r)
  }
}
The Web Client

val client = new AsyncHttpClient
def get(url: String): String = {
    val response = client.prepareGet(url).execute().get
    if (response.getStatusCode < 400)
        response.getResponseBodyExcerpt(131072)
    else throw BadStatus(response.getStatusCode)
}

Blocks the caller until the web server has replied
⇒ actor is deaf to other requests, e.g., cancellation
⇒ priority inversion: current thread is blocked
The Web Client

```scala
val client = new AsyncHttpClient
def get(url: String)(implicit exec: Executor):
    Future[String] = {
        val f = client.prepareGet(url).execute()
        val p = Promise[String]()
        f.addListener(new Runnable {
            def run = {
                val response = f.get
                if (response.getStatusCode < 400)
                    p.success(response.getResponseBodyExcerpt(131072))
                else p.failure(BadStatus(response.getStatusCode))
            }
        }, exec)
        p.future
    }
```
Important Lessons to Remember

• Prefer `context.become` for different behaviors, with data local to each behavior

• An actor application is non-blocking – event-driven from top to bottom
Finding Links

```scala
def findLinks(body: String): Iterator[String] = {
  for {
    anchor <- A_TAG.findAllMatchIn(body)
    HREF_ATTR(dquot, quot, bare) <- anchor.subgroups
  } yield
    if (dquot != null) dquot
    else if (quot != null) quot
    else bare
}
```

```html
<html>
<head> ... </head>
<body>
 ... 
<a href="http://cs.nyu.edu"></a>
 ... 
</body>
</html>
```
The Getter Actor (1)

class Getter(url: String, depth: Int) extends Actor {
    implicit val exec = context.dispatcher.asInstanceOf[Executor with ExecutionContext]

    val future = WebClient.get(url)
    future onComplete {
        case Success(body) => self ! body
        case Failure(err) => self ! Status.Failure(err)
    }
    ...
}
class Getter(url: String, depth: Int) extends Actor {

  implicit val exec = context.dispatcher.asInstanceOf[Executor with ExecutionContext]

  val future = WebClient.get(url)
  future.pipeTo(self)

  ...
}

The Getter Actor (2)
class Getter(url: String, depth: Int) extends Actor {
    implicit val exec = context.dispatcher.asInstanceOf[Executor with ExecutionContext]

    WebClient get url pipeTo self
    ...
}
Important Lessons to Remember

• Prefer context.become for different behaviors, with data local to each behavior
• An actor application is non-blocking – event-driven from top to bottom
• Actors are run by a dispatcher – potentially shared – which can also run Futures
The Getter Actor (4)

class Getter(url: String, depth: Int) extends Actor {
...
  def receive = {
    case body: String =>
      for (link <- findLinks(body))
        context.parent ! Controller.Crawl(link, depth)
      stop()
    case _: Status.Failure => stop()
  }
  def stop() = {
    context.parent ! Done
    context.stop(self)
  }
}
Actor Logging

- Logging includes IO which can block indefinitely
- Akka’s logging delegates this task to dedicated actor
- supports system-wide levels of debug, info, warning, error
- set level, e.g., by using the setting akka.loglevel=DEBUG

```scala
class A extends Actor with ActorLogging {
  def receive = {
    case msg => log.debug("received message: {}", msg)
  }
}
```
class Controller extends Actor with ActorLogging {
  var cache = Set.empty[String]
  var children = Set.empty[ActorRef]
  def receive = {
    case Crawl(url, depth) =>
      log.debug("{} crawling {}", depth, url)
      if (!cache(url) && depth > 0)
        children += context.actorOf(
          Props(new Getter(url, depth - 1)))
      cache += url
    case Getter.Done =>
      children -= sender
      if (children.isEmpty) context.parent ! Result(cache)
  }
}
Important Lessons to Remember

- Prefer context.become for different behaviors, with data local to each behavior.
- An actor application is non-blocking — event-driven from top to bottom.
- Actors are run by a dispatcher — potentially shared — which can also run Futures.
- Prefer immutable data structures, since they can be shared between actors.
import scala.concurrent.duration._

class Controller extends Actor with ActorLogging {
    context.setReceiveTimeout(10 seconds)
    ...
    def receive = {
        case Crawl(...) => ...
        case Getter.Done => ...
        case ReceiveTimeout => children foreach (_ ! Getter.Abort)
    }
}

The receive timeout is reset by every received message.
class Getter(url: String, depth: Int) extends Actor {
    ...
    def receive = {
        case body: String =>
            for (link <- findLinks(body)) ...
                stop()
        case _: Status.Failure => stop()
        case Abort => stop()
    }
    def stop() = {
        context.parent ! Done
        context.stop(self)
    }
}
The Scheduler

Akka includes a timer service optimized for high volume, short durations, and frequent cancellations of events.

```scala
trait Scheduler {
  def scheduleOnce(delay: FiniteDuration, target: ActorRef, msg: Any) (
    implicit ec: ExecutionContext): Cancellable

  def scheduleOnce(delay: FiniteDuration)(block: => Unit) (
    implicit ec: ExecutionContext): Cancellable

  def scheduleOnce(delay: FiniteDuration, run: Runnable) (
    implicit ec: ExecutionContext): Cancellable

  // ... the same for repeating timers
}
```
Adding an Overall Timeout (1)

class Controller extends Actor with ActorLogging {
    import context.dispatcher
    var children = Set.empty[ActorRef]
    context.system.scheduler.scheduleOnce(10 seconds) {
        children foreach (_ ! Getter.Abort)
    }
    ...
}

This is not thread-safe!
- code is run by the scheduler in a different thread
- potential race condition on children
class Controller extends Actor with ActorLogging {
  import context.dispatcher
  var children = Set.empty[ActorRef]
  context.system.scheduler.scheduleOnce(10 seconds, self, Timeout)
  ...
  def receive = {
    ...
    case Timeout => children foreach (_ ! Getter.Abort)
  }
}
How Actors and Futures Interact (1)

Future composition methods invite to closing over the actor’s state:

```scala
class Cache extends Actor {
  var cache = Map.empty[String, String]
  def receive = {
    case Get(url) =>
      if (cache contains url) sender ! cache(url)
    else
      WebClient get url foreach { body =>
        cache += url -> body
        sender ! body
      }
  }
}
```
class Cache extends Actor {
    var cache = Map.empty[String, String]
    def receive = {
        case Get(url) =>
            if (cache contains url) sender ! cache(url)
            else
                WebClient
                    .get(url)
                    .map { Result(sender, url, _) }
                    .pipeTo(self)
        case Result(client, url, body) =>
            cache += url -> body
            client ! body
    }
}

Still leaking state!
class Cache extends Actor {
  var cache = Map.empty[String, String]
  def receive = {
    case Get(url) =>
      if (cache contains url) sender ! cache(url)
    else
      val client = sender
      WebClient.get(url) map (Result(client, url, _))
        pipeTo self
    case Result(client, url, body) =>
      cache += url -> body
      client ! body
  }
}
Important Lessons to Remember

• Prefer context.become for different behaviors, with data local to each behavior
• An actor application is non-blocking – event-driven from top to bottom
• Actors are run by a dispatcher – potentially shared – which can also run Futures
• Prefer immutable data structures, since they can be shared
• Do not refer to actor state from code running asynchronously
class Receptionist extends Actor {
    def receive = waiting

    def waiting: Receive = {
          // upon Get(url) start a crawl and become running
    }

    def running(queue: Vector[Job]): Receive = {
          // upon Get(url) append that to queue and keep running
          // upon Controller.Result(links) ship that to client
          // and run next job from queue (if any)
    }
}
case class Job(client: ActorRef, url: String)
val DEPTH = 2
var reqNo = 0

def runNext(queue: Vector[Job]): Receive = {
  reqNo += 1
  if (queue.isEmpty) waiting
  else {
    val controller = context.actorOf(Props[Controller], s"c$reqNo")
    controller ! Controller.Crawl(queue.head.url, DEPTH)
    running(queue)
  }
}
def enqueueJob(queue: Vector[Job]): Receive = {
    if (queue.size > 3) {
        sender ! Failed(job.url)
        running(queue)
    } else running(queue :+ job)
}
def waiting: Receive = {
    case Get(url) =>
        context.become(runNext(Vector(Job(sender, url))))
}

def running(queue: Vector[Job]): Receive = {
    case Controller.Result(links) =>
        val job = queue.head
        job.client ! Result(job.url, links)
        context.stop(sender)
        context.become(runNext(queue.tail))
    case Get(url) =>
        context.become(enqueueJob(queue, Job(sender, url)))
}
Important Lessons to Remember

• Prefer `context.become` for different behaviors, with data local to each behavior.
• An actor application is non-blocking – event-driven from top to bottom.
• Actors are run by a dispatcher – potentially shared – which can also run Futures.
• Prefer immutable data structures, since they can be shared.
• Do not refer to actor state from code running asynchronously.