Introducción a Akka

David Vallejo

 @dvnavarro

Índice

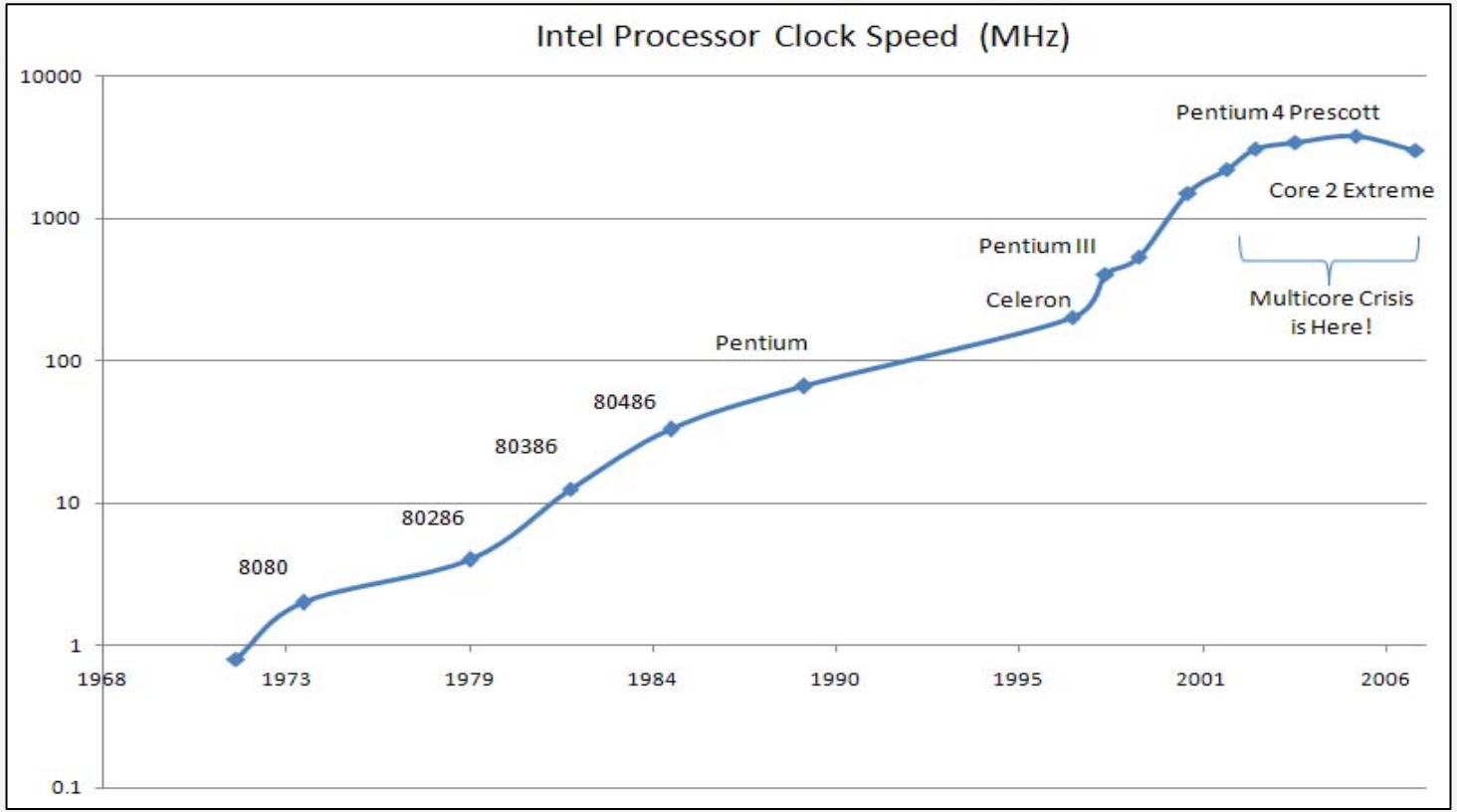
1. Motivación
2. Modelo de Actores
3. Features avanzadas
4. Testing
5. Conclusión


MOTIVACIÓN

Ley de Moore

- No es fácil crear procesadores más potentes
- Existe una limitación física a la hora de incorporar más transistores a los procesadores
- Se piensa en utilizar varios procesadores en paralelo
- A principios de la década pasada aparecen los multicores

Ley de Moore


Intel Processor Clock Speed (MHz)
10000
1000
100
10
1
0.1
1968
1973
1979
1984
1990
1995
2001
2006
8080
80286
80386
80486
Pentium
Celeron
Pentium III
Pentium 4 Prescott
Core 2 Extreme
Multicore Crisis is Here!

Concurrencia y Paralelismo

- Programar en sistemas multicore implica concurrencia
- Condiciones de carrera
- Ahora programar es más complicado

MODELO DE ACTORES

¿Qué es un Actor?

- Caja negra que reacciona a determinados mensajes
- Puede tener (o no) un estado interno
- No se puede acceder al estado de un actor desde fuera
- Procesa solo un mensaje cada vez

¿Por qué esto funciona?

REGLA FUNDAMENTAL

"***Solo*** *se procesa un mensaje a la vez*"

Los actores son mutex

Pueden tener estado sin condiciones de carrera!!

¿Cómo definimos un Actor?

- Trait Actor
- Método receive Comportamiento del actor

```
type Receive = PartialFunction[Any, Unit]
```

```
def receive: Receive = {
    case message => doSomething()
}
```

Akka System

- Contexto en el que viven los actores
- Necesario para poder definir un actor

```
val system = ActorSystem("my-first-system")
```

ActorRef

- No podemos acceder al interior del actor
- Trabajaremos con referencias a actores

```
val myActor: ActorRef =
    system.actorOf(Props[MyFirstActor],"my-first-actor")
```

- Útil, por ejemplo, para pasar actores como un parámetro más

Envío de mensajes

- Enviar un mensaje a un actor y olvidarse de él:

```
childActor ! "Be careful!"
```

- Enviar mensajes esperando respuesta:

```
childActor ? "Be careful!"
```

Devuelve un futuro.

Hello World en Akka

```
class MyFirstActor extends Actor {

  def receive = {
    case _ => println("Hello World!")
  }

}
```

Hello World en Akka

```
object HelloWorldBoot extends App {

  val system = ActorSystem("my-first-system")
  val myActor: ActorRef =
    system.actorOf(Props[MyFirstActor],"my-first-actor")

  myActor ! "Start!"

  system.shutdown()
}
```

FEATURES AVANZADAS

Envío de mensajes y Sender

Existe una variable implícita que devuelve la referencia del actor que envió un mensaje. Se llama sender.

```
sender ! "Nice to meet you!"
```

Puedo enviar un mensaje a un actor modificando el sender por otro ActorRef conocido:

```
childActor.tell("Be careful!", motherActor)
```

De esta manera podemos hacer que un actor conteste a un tercero.

Estado en actores

- Los actores pueden tener “estado”
  - Mediante valores definidos en el interior del actor
  - Mediante comportamientos => Máquinas de estado

Estado mediante valores

```
class StateActor extends Actor {

  var state: Int = 0

  def receive = ???

}
```

Estado en actores mediante valores

- Estamos usando vars!!!!
- Pero, recordemos dos cosas:
  - No se puede acceder desde fuera al estado de un actor
  - Un actor funciona como un mutex. No hay condiciones de carrera
- DON'T PANIC! :)

Estado en actores mediante valores

- Usar variables no rompe la pureza si su uso está localizado
- Podemos encontrar ejemplos en la librería de Scala:

```
def isEmpty: Boolean = {
   var result = true
   breakable {
     for (x <- this) {
       result = false
       break
     }
   }
   result
 }
```

En los actores ocurre lo mismo. La variable no sale fuera

Estado en actores

Ejercicio: definir un actor que acumule los números que se le pasen por mensaje.

Estado en actores

Posible solución:
AdderActor

```
class AdderActor extends Actor {
  var accum = 0
  def receive = {
    case Add(n: Int) => accum = accum + n
    case PrintAccum => println(s"Accum: $accum")
  }
}
case class Add(n: Int)
case object PrintAccum
```

Comportamientos

- Un actor puede tener uno o varios comportamientos: **behaviours**
- Cada comportamiento se define como un método receive
- Es posible cambiar de comportamientos con el metodo context. become:

```
def ping: Receive = {
  case _ =>
    sender ! "Ping"
    context.become(pong)
}
```

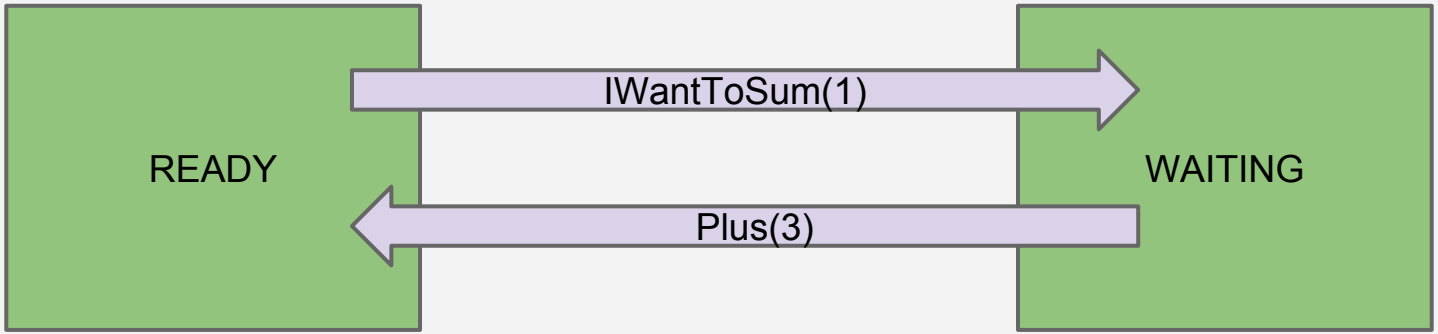
```
def pong: Receive = {
  case _ =>
    sender ! "Pong"
    context.become(ping)
}
```

Comportamientos

Ejercicio: definir un actor que, cuando se le pase un entero, cambie de comportamiento para esperar un segundo, y devolver la suma de ambos.

Comportamientos

Posible solución: CalculActor.scala



Routing

- Permite enviar un mensaje a un conjunto de actores como si fuesen un único ActorRef

- Se puede adoptar cualquier política

```
val router: ActorRef =

  context.actorOf(RoundRobinPool(5).props(Props[HelloWorldActor]), "router")
```

Scheduler

- Planificar envíos de mensajes a un actor.
- De forma periódica o puntual

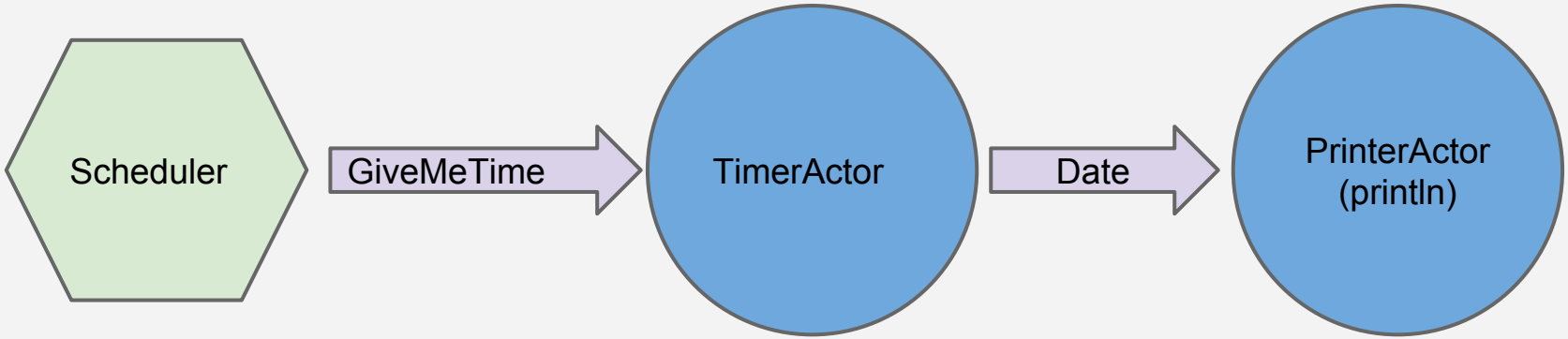
```
val timerActor = system.actorOf(Props[TimerActor], "timer-actor")

import system.dispatcher
import scala.concurrent.duration._
import scala.language.postfixOps

val timerServiceScheduler =
  system.scheduler.schedule(0 milliseconds, 1 seconds, timerActor, GiveMeTime)
```

Scheduler

Ejercicio: definir un actor que devuelva un String con la fecha, cada segundo.

Scheduler

Posible solución: TimerActor & PrinterActor



Ask Pattern

- Mediante “?” podemos esperar la respuesta de un actor
- Será devuelta como un futuro para permitir asincronía
- Hay que tener en cuenta dos aspectos importantes:
  - Timeout
  - Mapeo del futuro. Recibimos un Future[Any]

Ask Pattern

Ejemplo de Ask-Pattern:

```
implicit val timeout = Timeout(5 seconds) // needed for `?` below
val f: Future[Result] = (actorRef ? AreYouThere).mapTo[String]
```

Podemos redirigir el mensaje del futuro y enviarlo a otro actor:

```
f pipeTo otherActor
```

Supervisión

- Un actor puede crear actores en su contexto de ejecución
- Se establece una jerarquía de actores

```
class FirstActor extends Actor {
  val child = context.actorOf(Props[MyActor], name = "myChild")
  // plus some behavior ...
}
```

Supervisión

- Se puede definir una política de recuperación por si uno o varios hijos sufren un fallo de ejecución.

```
override val supervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
   case _: NullPointerException     => Restart
   case _: IllegalArgumentException => Stop
  }
```

Ask Pattern - Supervision

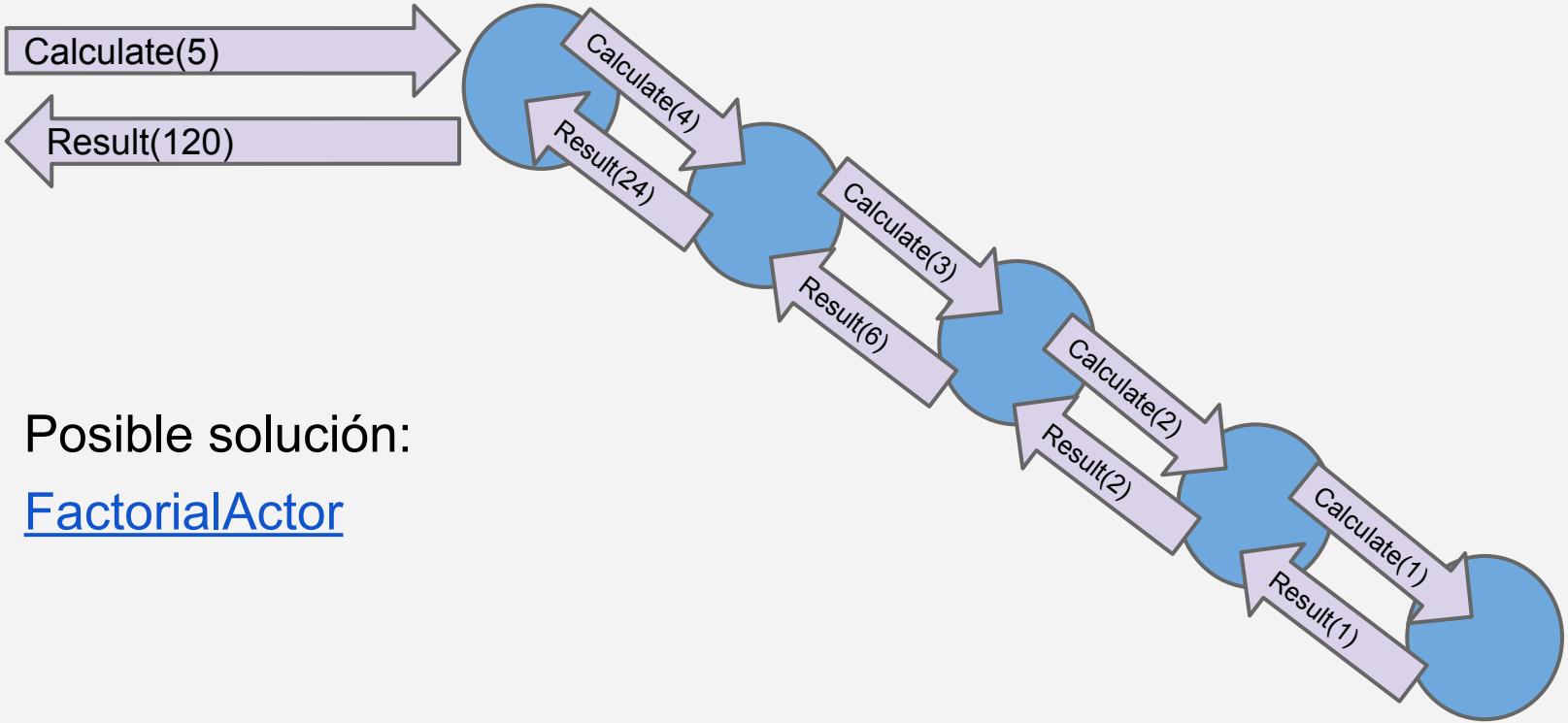
Ejercicio:

Crear un actor que calcule el factorial de un número. Para ello, un primer actor recibirá un mensaje con un entero n que será el número al que hay que calcular el factorial.

Ese actor, creará un hijo al que pedirá el factorial de n-1

El actor que tenga como propósito crear el factorial de 1 devolverá un 1 a su padre.

Ask Pattern - Supervision



Posible solución:

[FactorialActor]


TESTING

Akka-testkit

Proporciona funcionalidad para probar modelos de actores.

Dos conceptos básicos:

- TestActorRef
- ProbeActor

TestActorRef

Permite acceder al interior de un actor para hacer comprobaciones de estado:

```
val testActorRef = TestActorRef[AdderActor]
val testActor = testActorRef.underlyingActor

testActor.accum should be(0)
```

TestActorRef

Ejemplo: AdderActorSpec

TestProbe

Permite crear actores muy simples para realizar comprobaciones en ellos:

```
val probe1 = TestProbe()

val probe2 = TestProbe()

val actor = system.actorOf(Props[MyDoubleEcho])

actor ! ((probe1.ref, probe2.ref))

actor ! "hello"

probe1.expectMsg(500 millis, "hello")

probe2.expectMsg(500 millis, "hello")
```

TestProbe

Ejemplo: FactorialActorSpec


CONCLUSIONES

Conclusiones

- Paradigma Reactivo
- Beneficios de la programación funcional
- Útil en problemas distribuidos
- Difícil de testear

Conclusiones

- Futuros === Asincronía
- Flexible y dinámico
- “Estado” controlado


That's all Folks!

scalerablog.wordpress.com

@scalerablog