Akka

Quick Intro for hAkkers

What is it?

Toolkit for development of

- scalable
- distributed
- fault-tolerant

applications in Scala or Java

Modules Offered

- actors (typed / untyped / testkit)
- very slick Futures / Promises
- STM
- async-http
- camel
- spring
- cluster (upcoming in 2.0)
- … (to be filled, possibly by you!)

Project Details

- site at http://akka.io
- source at https://github.com/jboner/akka
- https://www.assembla.com/spaces/akka for tickets (need to “watch” the space)
- Apache License 2.0
- Contributor’s License Agreement needed for submissions

Hacking Guide

- Performance is King!
  - memory footprint
  - minimize allocations
  - avoid blocking operations
- Every feature or fix needs a corresponding test case
- Always update docs together with code
  - reStructured Text within git repo

Goals for Scalathon

- Hack demo application
  - get feedback on API & docs
  - get feedback on testkit
- contribute to improve experience for newcomers

Getting Started

- Strong encapsulation of Actor
  - only access through ActorRef
  - only constructible inside factory method
  - no calling of Actor methods from the outside, except through mailbox
- Communication via Channel[T]
  - allows sending
  - may be passed around

Actor Sample

```
import akka.actor._

class A extends Actor {
  def receive = {
    case “ping” => self reply “pong”
  }
}

class B(a: ActorRef) extends Actor {
  var last: UntypedChannel = NullChannel
  def receive = {
    case “ping” =>
      last = self.channel
      a ! “ping”
    case “pong” => last safe_! “pong”
  }
}
```

```
import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers
import akka.testkit.TestKit
import akka.util.duration._

class ABSpec extends WordSpec with MustMatchers
    with TestKit {

  “Some B” must {
    “return answer obtained from A” in {
      val a = actorOf[A].start()
      val b = actorOf(new B(a)).start()
      within(500 millis) {
        b ! “ping”
        expectMsg(“pong”)
      }
    }
  }
}
```

Future Sample

```scala
import akka.dispatch.Future
import Future.flow

val x = Future(calculateSomeExpensiveInt())
val y = Future(calculateSomeExpensiveFloat())
val z = flow {
  val f = someActor ? x()
  // someActor will reply at some point with a Float
  Some(f() / y())
} recover {
  case _: ArithmeticException => None
}

z onResult {
  case Some(x) => doSomething(x)
  case None => log.error(“calculation failed”)
}

val result = z.await.result // if you must really block
```

Conclusion

- Forget what you knew about “composing” concurrent programs
- View actors as fundamental building blocks encapsulating state
- Go wild!