Industrial-grade Concurrency with Akka

presented by Robbie Strickland

Who am I?

Robbie Strickland
Managing Architect, E3 Greentech
rostrickland@gmail.com
github.com/rstrickland
linkedin.com/in/robbiestrickland

Agenda

In scope

- Getting started
- How Akka actors work
- Supervisors
- Work-stealing
- Camel integration
- Handling HTTP requests
- Using the microkernel

Out of scope

- Actor design patterns
- Remote actors (deprecated in 2.0)
- Java API
- Comparison of Scala actor implementations
- Akka's STM

What is Akka?

- Distributed actor & STM implementation
- APIs for Scala & Java
- Written in Scala
- Now a part of the Typesafe offering
- Open source

Why Akka?

there are 4 (!!) Scala actor implementations

Key differentiators

- Robust supervisor structure - failure is embraced
- Built-in STM implementation
- Supports clustering of actors across multiple nodes
- Work-stealing among similar actors
- Nearly every part of the framework is configurable
- Lightweight microkernel enables easy deployment
- Full Camel support makes integration almost trivial
- Durable actor mailbox support
- Java API means your legacy code can easily integrate

The Basics

How to get up and running

Version status

- Version 1.2 is current
- Version 2.0 coming soon
  - Remote actors replaced by transparent clustered actors
  - Uses ZooKeeper for cluster coordination
  - Adds durable actor mailboxes

Runtime options

Akka Microkernel

- Lightweight, self-contained environment
- Simple to configure and deploy
- Embedded Jetty with JAX-RS support

Runtime options

Standalone application

- Just import the Akka classes
- Easily run from SBT, IDE, etc.

External container

- e.g. Tomcat, JEE container, etc.
- Directly integrate with existing application

The actor model

- Concurrency model as an alternative to threads
- Many actors can exist on a single thread
- Resource utilization is trivial for idle actors
- Every actor has a mailbox & processes messages as it receives them
- Actors are inherently asynchronous
- Similar actors can be pooled to allow for concurrent processing

Akka Actors

Actor lifecycle

Created - ActorRef instantiated but cannot receive messages
Started - ready to receive messages
Shut down - can no longer receive messages

Hooks exist to insert your custom code before and after each life cycle change event

Simple example - using an actor

```
val myActor = actorOf[MySimpleActor]   // actor created
myActor.start   // actor started
myActor ! "test"    // send fire and forget message to actor
myActor.stop    // actor shut down (not needed in most cases)
```

Simple example - inside the actor

```
class MySimpleActor extends Actor {
  def receive = {
    case msg:String => println(msg)
    case _ => println("Unknown message type")
  }
}
```

Two-way communications

- Actor replies to sender using reference to self.channel
- Two possible reply methods:
  - self.channel ! msg - throws an exception if sender does not exist (e.g. sender is not an actor)
  - self.channel tryTell msg - returns true if message sent successfully

Two-way communications

```
class MySimpleActor extends Actor {
  def receive = {
    case msg:String => self.channel ! msg + " received"
    case _ => self.channel tryTell "Unknown message type"
  }
}
```

Two-way communications

Send and receive future

```
val res = myActor ? msg  // res is a Future
```

Send and receive eventually

```
val res = (myActor ? msg).as[String]  // res is an Option[String]
```

Two-way communications

```
val myActor = actorOf[MySimpleActor]
myActor.start
val response = (myActor ? "test").as[String].getOrElse("")
println("Response = " + response)
```

Let's run some code!

Actor supervision

- Supervisors can manage groups of actors
- Permits graceful handling of actor failures
- Two failure strategies:
  - All for one - if any actor fails all are restarted
  - One for one - only the failed actor is restarted
- Supervisors can be ordered into hierarchies to provide for robust failure handling

Actor supervision

```
val myActor1 = actorOf[MySimpleActor].start
val myActor2 = actorOf[MySimpleActor].start

val supervisorConfig =
  SupervisorConfig(AllForOneStrategy(List(classOf[Throwable]), 3, 1000), Nil)
val supervisor = Supervisor(supervisorConfig)

supervisor.link(myActor1)
supervisor.link(myActor2)
```

Let's run some code!

Dispatchers

- Dispatchers determine how messages are delivered to actors
- Four types:
  - **Thread-based** - creates one thread per actor
  - **Event-based** - actors share threads based on large number of config options
  - **Priority event-based** - same as above with ability to specify priority of messages
  - **Work-stealing** - same as event-based except reallocates work to idle actors

Dispatchers

Define dispatcher in a companion object so all instances get the dispatcher:

```
object MyActor {
  val dispatcher =
    Dispatchers.newExecutorBasedEventDrivenDispatcher("MyActor").build
}

class MyActor extends Actor {
  self.dispatcher = MyActor.dispatcher
  ...
}
```

Work-stealing

- More accurately, work donating
- Overloaded actor will donate messages to idle actors
- Does not donate to non-idle actors to avoid messages being "stolen" more than once
- Adds minimal performance overhead

Let's run some code!

Camel Integration

What is Camel?

- Apache project
- Abstracts protocol from implementation
- Allows for plug-and-play protocol swapping
- Makes integration with existing infrastructures trivial
- Supports a vast array of protocols
- Uses Producer/Consumer architecture
- Producers create messages on an endpoint
- Consumers respond to messages on an endpoint

Camel with Akka

- Include necessary dependencies in your project:
  - Akka-Camel component
  - Camel core component
  - Protocol-specific Camel component
  - Actual protocol implementation
- Mix in Consumer/Producer traits to your actors
- Define the Camel endpoint URI
- Start sending messages

Example build.sbt using jetty

```
libraryDependencies ++= Seq(
  "se.scalablesolutions.akka" % "akka-actor" % "1.2",
  "se.scalablesolutions.akka" % "akka-camel" % "1.2",
  "org.apache.camel" % "camel" % "2.8.2",
  "org.apache.camel" % "camel-jetty" % "2.8.2",
  "org.eclipse.jetty" % "jetty-webapp" % "8.0.0.M2" % "test",
  "org.mortbay.jetty" % "servlet-api" % "3.0.20100224" % "provided"
)
```

Example producer

```
class CamelProducer extends Actor with Producer {
  def endpointUri = "jetty:http://localhost:6200/test"
}
```

... that's it!

Example consumer

```
class CamelConsumer extends Actor with Consumer {
  def endpointUri = "jetty:http://localhost:6200/test"

  def receive = {
    case msg: Message =>
      val msgStr = msg.getBodyAs(classOf[String])
      self.channel ! msgStr + " received"
  }
}
```

Let's run some code!

Handling HTTP requests with Akka

Akka Mist

- Async, non-blocking HTTP
- HTTP server backed by actor pool
- Runs inside microkernel using embedded Jetty
- Simple implementation:
  - Mix in Endpoint trait & define URI hook
  - Specify request handler actor
  - Create boot class
  - Add boot class to akka.conf

Mist - example endpoint

```
class MistActor extends Actor with Endpoint {
  self.dispatcher = Endpoint.Dispatcher

  def hook(uri:String) = uri startsWith "/mistTest"
  def provide(uri:String) = actorOf[MistActorService].start

  override def preStart = registry.actorsFor(classOf[RootEndpoint]).head !
      Endpoint.Attach(hook, provide)

  def receive = handleHttpRequest
}
```

Mist - example handler

```
class MistActorService extends Actor {
  def receive = {

    case get:Get =>
      def default(any: Any) = ""
      val msg = get.getParameterOrElse("msg", default)
      get OK msg + " received"

    case other:RequestMethod => other NotAllowed "unsupported request"
  }
}
```

Mist - example boot class

```
class Boot {
  val factory = SupervisorFactory(SupervisorConfig(
                    OneForOneStrategy(List(classOf[Exception]), 3, 100),
                    Supervise(actorOf[RootEndpoint], Permanent) ::
                    Supervise(actorOf[MistActorService], Permanent) :: Nil))

  factory.newInstance.start
}
```

Let's run some code!

Other Akka Goodies

i.e. stuff to explore later if interested

Additional features

- Software Transactional Memory
- Actor upgrade - actors can change behavior on the fly
- Typed Actors - turns regular POJOs into actors
- Transactors - combines actors with STM
- Routing - ActorPool, LoadBalancer, etc
- Finite State Machine - out-of-the box FSM implementation
- Play! framework integration
- Pinky REST/MVC integration

Add-on modules

- ▫AMQP integration
- Scalaz support
- Spring integration
- OSGi support

That's it!

Contact info again

Robbie Strickland
Managing Architect, E3 Greentech
rostrickland@gmail.com
github.com/rstrickland
linkedin.com/in/robbiestrickland