
Akka Documentation

Release 1.2

Scalable Solutions AB

September 19, 2011

CONTENTS

1	Introduction	1
1.1	What is Akka?	1
1.2	Why Akka?	2
1.3	Getting Started	3
1.4	Getting Started Tutorial (Scala): First Chapter	6
1.5	Getting Started Tutorial (Scala with Eclipse): First Chapter	16
1.6	Getting Started Tutorial (Java): First Chapter	29
1.7	Use-case and Deployment Scenarios	42
1.8	Examples of use-cases for Akka	43
2	General	45
2.1	Akka and the Java Memory Model	45
2.2	Configuration	46
2.3	Event Handler	50
2.4	SLF4J	52
3	Common utilities	53
3.1	Scheduler	53
3.2	Duration	53
4	Scala API	55
4.1	Actors (Scala)	55
4.2	Typed Actors (Scala)	65
4.3	ActorRegistry (Scala)	68
4.4	Futures (Scala)	70
4.5	Dataflow Concurrency (Scala)	75
4.6	Agents (Scala)	79
4.7	Software Transactional Memory (Scala)	82
4.8	Transactors (Scala)	91
4.9	Remote Actors (Scala)	95
4.10	Serialization (Scala)	107
4.11	Fault Tolerance Through Supervisor Hierarchies (Scala)	121
4.12	Dispatchers (Scala)	129
4.13	Routing (Scala)	133
4.14	FSM	138
4.15	HTTP	146
4.16	HTTP Security	155
4.17	Testing Actor Systems	160
4.18	Tutorial: write a scalable, fault-tolerant, network chat server and client (Scala)	172
5	Java API	184
5.1	Actors (Java)	184
5.2	Typed Actors (Java)	190
5.3	ActorRegistry (Java)	194

5.4	Futures (Java)	195
5.5	Dataflow Concurrency (Java)	200
5.6	Software Transactional Memory (Java)	203
5.7	Transactors (Java)	213
5.8	Remote Actors (Java)	217
5.9	Serialization (Java)	227
5.10	Fault Tolerance Through Supervisor Hierarchies (Java)	230
5.11	Dispatchers (Java)	239
5.12	Routing (Java)	244
5.13	Guice Integration	245
6	Information for Developers	247
6.1	Building Akka	247
6.2	Developer Guidelines	249
6.3	Documentation Guidelines	250
6.4	Team	252
7	Project Information	253
7.1	Migration Guides	253
7.2	Release Notes	265
7.3	Scaladoc API	281
7.4	Documentation for Other Versions	281
7.5	Issue Tracking	282
7.6	Licenses	283
7.7	Sponsors	286
7.8	Support	286
7.9	Mailing List	286
7.10	Downloads	286
7.11	Source Code	286
7.12	Maven Repository	286
8	Additional Information	288
8.1	Add-on Modules	288
8.2	Articles & Presentations	288
8.3	Benchmarks	290
8.4	Here is a list of recipies for all things Akka	291
8.5	External Sample Projects	291
8.6	Projects using the removed Akka Persistence modules	295
8.7	Companies and Open Source projects using Akka	295
8.8	Third-party Integrations	299
8.9	Other Language Bindings	300
8.10	Feature Stability Matrix	300
9	Links	301
	Python Module Index	302
	Index	303

INTRODUCTION

1.1 What is Akka?

Simpler Scalability, Fault-Tolerance, Concurrency & Remoting through Actors

We believe that writing correct concurrent, fault-tolerant and scalable applications is too hard. Most of the time it's because we are using the wrong tools and the wrong level of abstraction. Akka is here to change that. Using the Actor Model together with Software Transactional Memory we raise the abstraction level and provide a better platform to build correct concurrent and scalable applications. For fault-tolerance we adopt the Let it crash/Embrace failure model which have been used with great success in the telecom industry to build applications that self-heals, systems that never stop. Actors also provides the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications. Akka is Open Source and available under the Apache 2 License.

Download from <http://akka.io/downloads/>

1.1.1 Akka implements a unique hybrid

- *Actors (Java)*, which gives you:
 - Simple and high-level abstractions for concurrency and parallelism.
 - Asynchronous, non-blocking and highly performant event-driven programming model.
 - Very lightweight event-driven processes (create ~6.5 million actors on 4GB RAM).
- *Fault Tolerance Through Supervisor Hierarchies (Java)* through supervisor hierarchies with “let-it-crash” semantics. Excellent for writing highly fault-tolerant systems that never stop, systems that self-heal.
- *Software Transactional Memory (Java)* (STM). (Distributed transactions coming soon).
- *Transactors (Java)*: combine actors and STM into transactional actors. Allows you to compose atomic message flows with automatic retry and rollback.
- *Remote Actors (Java)*: highly performant distributed actors with remote supervision and error management.
- *Java API* and *Scala API*

1.1.2 Akka can be used in two different ways

- As a library: used by a web app, to be put into ‘WEB-INF/lib’ or as a regular JAR on your classpath.
- As a microkernel: stand-alone kernel, embedding a servlet container and all the other modules.

See the *Use-case and Deployment Scenarios* for details.

1.2 Why Akka?

1.2.1 What features can the Akka platform offer, over the competition?

Akka is an unified runtime and programming model for:

- Scale up (Concurrency)
- Scale out (Remoting)
- Fault tolerance

One thing to learn and admin, with high cohesion and coherent semantics.

Akka is a very scalable piece of software, not only in the performance sense, but in the size of applications it is useful for. The core of Akka, akka-actor, is very small and easily dropped into an existing project where you need asynchronicity and lockless concurrency without hassle.

You can choose to include only the parts of akka you need in your application and then there's the whole package, the Akka Microkernel, which is a standalone container to deploy your Akka application in. With CPUs growing more and more cores every cycle, Akka is the alternative that provides outstanding performance even if you're only running it on one machine. Akka also supplies a wide array of concurrency-paradigms, allowing for users to choose the right tool for the job.

The integration possibilities for Akka Actors are immense through the Apache Camel integration. We provide Software Transactional Memory concurrency control through the excellent Multiverse project, and have integrated that with Actors, creating Transactors for coordinated concurrent transactions. We have Agents and Dataflow concurrency as well.

1.2.2 What's a good use-case for Akka?

(Web, Cloud, Application) Services - Actors lets you manage service failures (Supervisors), load management (back-off strategies, timeouts and processing-isolation), both horizontal and vertical scalability (add more cores and/or add more machines). Think payment processing, invoicing, order matching, datacrunching, messaging. Really any highly transactional systems like banking, betting, games.

Here's what some of the Akka users have to say about how they are using Akka: <http://stackoverflow.com/questions/4493001/good-use-case-for-akka>

1.2.3 Cloudy Akka

And that's all in the ApacheV2-licensed open source project. On top of that we have a commercial product called Cloudy Akka which provides the following features:

1. Management through Dashboard, JMX and REST
2. Monitoring through Dashboard, JMX and SNMP
3. Dapper-style tracing of messages across components and remote nodes
4. A configurable alert system
5. Real-time statistics
6. Very low overhead monitoring agents (should always be on in production)
7. Consolidation of statistics and logging information to a single node
8. Data analysis through Hadoop
9. Storage of statistics data for later processing
10. Provisioning and rolling upgrades through a dashboard

Read more [here](#).

1.3 Getting Started

Contents

- [Prerequisites](#)
- [Download](#)
- [Modules](#)
- [Using a release distribution](#)
 - [Microkernel](#)
- [Using a build tool](#)
- [Using Akka with Maven](#)
- [Using Akka with SBT](#)
- [Using Akka with Eclipse](#)
- [Using Akka with IntelliJ IDEA](#)
- [Build from sources](#)
- [Need help?](#)

The best way to start learning Akka is to try the Getting Started Tutorial, which comes in several flavours depending on you development environment preferences:

- *Getting Started Tutorial (Java): First Chapter* for Java development, either
 - as standalone project, running from the command line,
 - or as Maven project and running it from within Maven
- *Getting Started Tutorial (Scala): First Chapter* for Scala development, either
 - as standalone project, running from the command line,
 - or as SBT (Simple Build Tool) project and running it from within SBT
- *Getting Started Tutorial (Scala with Eclipse): First Chapter* for Scala development with Eclipse

The Getting Started Tutorial describes everything you need to get going, and you don't need to read the rest of this page if you study the tutorial. For later look back reference this page describes the essential parts for getting started with different development environments.

1.3.1 Prerequisites

Akka requires that you have [Java 1.6](#) or later installed on you machine.

1.3.2 Download

There are several ways to download Akka. You can download the full distribution with microkernel, which includes all modules. You can download just the core distribution. Or you can use a build tool like Maven or SBT to download dependencies from the Akka Maven repository.

1.3.3 Modules

Akka is split up into two different parts:

- Akka - The core modules. Reflects all the sections under [Scala API](#) and [Java API](#).
- Akka Modules - The microkernel and add-on modules, described in [Add-on Modules](#).

Akka is very modular and has many JARs for containing different features. The core distribution has seven modules:

- akka-actor-1.2 – Standard Actors
- akka-typed-actor-1.2 – Typed Actors
- akka-remote-1.2 – Remote Actors
- akka-stm-1.2 – STM (Software Transactional Memory), transactors and transactional datastructures
- akka-http-1.2 – Akka Mist for continuation-based asynchronous HTTP and also Jersey integration
- akka-slf4j-1.2 – SLF4J Event Handler Listener
- akka-testkit-1.2 – Toolkit for testing Actors

We also have Akka Modules containing add-on modules outside the core of Akka.

- akka-kernel-1.2 – Akka microkernel for running a bare-bones mini application server (embeds Jetty etc.)
- akka-amqp-1.2 – AMQP integration
- akka-camel-1.2 – Apache Camel Actors integration (it's the best way to have your Akka application communicate with the rest of the world)
- akka-camel-typed-1.2 – Apache Camel Typed Actors integration
- akka-scalaz-1.2 – Support for the Scalaz library
- akka-spring-1.2 – Spring framework integration
- akka-osgi-dependencies-bundle-1.2 – OSGi support

How to see the JARs dependencies of each Akka module is described in the [Dependencies](#) section. Worth noting is that akka-actor has zero external dependencies (apart from the scala-library.jar JAR).

1.3.4 Using a release distribution

Download the release you need, Akka core or Akka Modules, from <http://akka.io/downloads> and unzip it.

Microkernel

The Akka Modules distribution includes the microkernel. To run the microkernel:

- Set the AKKA_HOME environment variable to the root of the Akka distribution.
- To start the kernel use the scripts in the bin directory and deploy all samples applications from ./deploy dir.

More information is available in the documentation of the Microkernel in [Add-on Modules](#).

1.3.5 Using a build tool

Akka can be used with build tools that support Maven repositories. The Akka Maven repository can be found at <http://akka.io/repository> and Typesafe provides <http://repo.typesafe.com/typesafe/releases/> that proxies several other repositories, including akka.io.

1.3.6 Using Akka with Maven

Information about how to use Akka with Maven, including how to create an Akka Maven project from scratch, can be found in the [Getting Started Tutorial \(Java\): First Chapter](#).

Summary of the essential parts for using Akka with Maven:

1. Add this repository to your pom.xml:

```
<repository>
  <id>typesafe</id>
  <name>Typesafe Repository</name>
  <url>http://repo.typesafe.com/typesafe/releases/</url>
</repository>
```

2. Add the Akka dependencies. For example, here is the dependency for Akka Actor 1.2:

```
<dependency>
  <groupId>se.scalablesolutions.akka</groupId>
  <artifactId>akka-actor</artifactId>
  <version>1.2</version>
</dependency>
```

1.3.7 Using Akka with SBT

Information about how to use Akka with SBT, including how to create an Akka SBT project from scratch, can be found in the *Getting Started Tutorial (Scala): First Chapter*.

Summary of the essential parts for using Akka with SBT:

SBT installation instructions on <https://github.com/harrah/xsbt/wiki/Setup>

build.sbt file:

```
name := "My Project"

version := "1.0"

scalaVersion := "2.9.1"

resolvers += "Typesafe Repository" at "http://repo.typesafe.com/typesafe/releases/"

libraryDependencies += "se.scalablesolutions.akka" % "akka-actor" % "1.2"
```

1.3.8 Using Akka with Eclipse

Information about how to use Akka with Eclipse, including how to create an Akka Eclipse project from scratch, can be found in the *Getting Started Tutorial (Scala with Eclipse): First Chapter*.

Setup SBT project and then use `sbteclipse` to generate Eclipse project.

1.3.9 Using Akka with IntelliJ IDEA

Setup SBT project and then use `sbt-idea` to generate IntelliJ IDEA project.

1.3.10 Build from sources

Akka uses Git and is hosted at [Github](https://github.com).

- Akka: clone the Akka repository from <http://github.com/jboner/akka>
- Akka Modules: clone the Akka Modules repository from <http://github.com/jboner/akka-modules>

Continue reading the page on *Building Akka*

1.3.11 Need help?

If you have questions you can get help on the [Akka Mailing List](#).

You can also ask for [commercial support](#).

Thanks for being a part of the Akka community.

1.4 Getting Started Tutorial (Scala): First Chapter

1.4.1 Introduction

Welcome to the first tutorial on how to get started with Akka and Scala. We assume that you already know what Akka and Scala are and will now focus on the steps necessary to start your first project.

There are two variations of this first tutorial:

- creating a standalone project and run it from the command line
- creating a SBT (Simple Build Tool) project and running it from within SBT

Since they are so similar we will present them both.

The sample application that we will create is using actors to calculate the value of Pi. Calculating Pi is a CPU intensive operation and we will utilize Akka Actors to write a concurrent solution that scales out to multi-core processors. This sample will be extended in future tutorials to use Akka Remote Actors to scale out on multiple machines in a cluster.

We will be using an algorithm that is called “embarrassingly parallel” which just means that each job is completely isolated and not coupled with any other job. Since this algorithm is so parallelizable it suits the actor model very well.

Here is the formula for the algorithm we will use:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots = \frac{\pi}{4}.$$

In this particular algorithm the master splits the series into chunks which are sent out to each worker actor to be processed. When each worker has processed its chunk it sends a result back to the master which aggregates the total result.

1.4.2 Tutorial source code

If you don't want to type in the code and/or set up an SBT project then you can check out the full tutorial from the Akka GitHub repository. It is in the `akka-tutorials/akka-tutorial-first` module. You can also browse it online [here](#), with the actual source code [here](#).

To check out the code using Git invoke the following:

```
$ git clone git://github.com/jboner/akka.git
```

Then you can navigate down to the tutorial:

```
$ cd akka/akka-tutorials/akka-tutorial-first
```

1.4.3 Prerequisites

This tutorial assumes that you have Java 1.6 or later installed on your machine and `java` on your `PATH`. You also need to know how to run commands in a shell (ZSH, Bash, DOS etc.) and a decent text editor or IDE to type in the Scala code.

You need to make sure that `$JAVA_HOME` environment variable is set to the root of the Java distribution. You also need to make sure that the `$JAVA_HOME/bin` is on your `PATH`:

```
$ export JAVA_HOME=..root of java distribution..
$ export PATH=$PATH:$JAVA_HOME/bin
```

You can test your installation by invoking `java`:

```
$ java -version
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07-334-10M3326)
Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02-334, mixed mode)
```

1.4.4 Downloading and installing Akka

To build and run the tutorial sample from the command line, you have to download Akka. If you prefer to use SBT to build and run the sample then you can skip this section and jump to the next one.

Let's get the `akka-actors-1.2.zip` distribution of Akka from <http://akka.io/downloads/> which includes everything we need for this tutorial. Once you have downloaded the distribution unzip it in the folder you would like to have Akka installed in. In my case I choose to install it in `/Users/jboner/tools/`, simply by unzipping it to this directory.

You need to do one more thing in order to install Akka properly: set the `AKKA_HOME` environment variable to the root of the distribution. In my case I'm opening up a shell, navigating down to the distribution, and setting the `AKKA_HOME` variable:

```
$ cd /Users/jboner/tools/akka-actors-1.2
$ export AKKA_HOME='pwd'
$ echo $AKKA_HOME
/Users/jboner/tools/akka-actors-1.2
```

The distribution looks like this:

```
$ ls -l
config
doc
lib
src
```

- In the `config` directory we have the Akka conf files.
- In the `doc` directory we have the documentation, API, doc JARs, and also the source files for the tutorials.
- In the `lib` directory we have the Scala and Akka JARs.
- In the `src` directory we have the source JARs for Akka.

The only JAR we will need for this tutorial (apart from the `scala-library.jar` JAR) is the `akka-actor-1.2.jar` JAR in the `lib/akka` directory. This is a self-contained JAR with zero dependencies and contains everything we need to write a system using Actors.

Akka is very modular and has many JARs for containing different features. The core distribution has seven modules:

- `akka-actor-1.2.jar` – Standard Actors
- `akka-typed-actor-1.2.jar` – Typed Actors
- `akka-remote-1.2.jar` – Remote Actors

- akka-stm-1.2.jar – STM (Software Transactional Memory), transactors and transactional datastructures
- akka-http-1.2.jar – Akka Mist for continuation-based asynchronous HTTP and also Jersey integration
- akka-slf4j-1.2.jar – SLF4J Event Handler Listener for logging with SLF4J
- akka-testkit-1.2.jar – Toolkit for testing Actors

We also have Akka Modules containing add-on modules outside the core of Akka. You can download the Akka Modules distribution from <http://akka.io/downloads/>. It contains Akka core as well. We will not be needing any modules there today, but for your information the module JARs are these:

- akka-kernel-1.2.jar – Akka microkernel for running a bare-bones mini application server (embeds Jetty etc.)
- akka-amqp-1.2.jar – AMQP integration
- akka-camel-1.2.jar – Apache Camel Actors integration (it's the best way to have your Akka application communicate with the rest of the world)
- akka-camel-typed-1.2.jar – Apache Camel Typed Actors integration
- akka-scalaz-1.2.jar – Support for the Scalaz library
- akka-spring-1.2.jar – Spring framework integration
- akka- osgi-dependencies-bundle-1.2.jar – OSGi support

1.4.5 Downloading and installing Scala

To build and run the tutorial sample from the command line, you have to install the Scala distribution. If you prefer to use SBT to build and run the sample then you can skip this section and jump to the next one.

Scala can be downloaded from <http://www.scala-lang.org/downloads>. Browse there and download the Scala 2.9.1 release. If you pick the `tgz` or `zip` distribution then just unzip it where you want it installed. If you pick the IzPack Installer then double click on it and follow the instructions.

You also need to make sure that the `scala-2.9.1/bin` (if that is the directory where you installed Scala) is on your `PATH`:

```
$ export PATH=$PATH:scala-2.9.1/bin
```

You can test your installation by invoking `scala`:

```
$ scala -version
Scala code runner version 2.9.1.final -- Copyright 2002-2011, LAMP/EPFL
```

Looks like we are all good. Finally let's create a source file `Pi.scala` for the tutorial and put it in the root of the Akka distribution in the `tutorial` directory (you have to create it first).

Some tools require you to set the `SCALA_HOME` environment variable to the root of the Scala distribution, however Akka does not require that.

1.4.6 Downloading and installing SBT

SBT, short for 'Simple Build Tool' is an excellent build system written in Scala. It uses Scala to write the build scripts which gives you a lot of power. It has a plugin architecture with many plugins available, something that we will take advantage of soon. SBT is the preferred way of building software in Scala and is probably the easiest way of getting through this tutorial. If you want to use SBT for this tutorial then follow the following instructions, if not you can skip this section and the next.

To install SBT and create a project for this tutorial it is easiest to follow the instructions on <https://github.com/harrah/xsbt/wiki/Setup>.

Now we need to create our first Akka project. You could add the dependencies manually to the build script, but the easiest way is to use Akka's SBT Plugin, covered in the next section.

1.4.7 Creating an Akka SBT project

If you have not already done so, now is the time to create an SBT project for our tutorial. You do that by adding the following content to `build.sbt` file in the directory you want to create your project in:

```
name := "My Project"

version := "1.0"

scalaVersion := "2.9.1"

resolvers += "Typesafe Repository" at "http://repo.typesafe.com/typesafe/releases/"

libraryDependencies += "se.scalablesolutions.akka" % "akka-actor" % "1.2"
```

Create a directory `src/main/scala` in which you will store the Scala source files.

Not needed in this tutorial, but if you would like to use additional Akka modules beyond `akka-actor`, you can add these as `libraryDependencies` in `build.sbt`. Note that there must be a blank line between each. Here is an example adding `akka-remote` and `akka-stm`:

```
libraryDependencies += "se.scalablesolutions.akka" % "akka-actor" % "1.2"

libraryDependencies += "se.scalablesolutions.akka" % "akka-remote" % "1.2"

libraryDependencies += "se.scalablesolutions.akka" % "akka-stm" % "1.2"
```

So, now we are all set.

SBT itself needs a whole bunch of dependencies but our project will only need one; `akka-actor-1.2.jar`. SBT will download that as well.

1.4.8 Start writing the code

Now it's about time to start hacking.

We start by creating a `Pi.scala` file and adding these import statements at the top of the file:

```
package akka.tutorial.first.scala

import akka.actor.{Actor, PoisonPill}
import Actor._
import akka.routing.{Routing, CyclicIterator}
import Routing._
import akka.dispatch.Dispatchers

import java.util.concurrent.CountDownLatch
```

If you are using SBT in this tutorial then create the file in the `src/main/scala` directory.

If you are using the command line tools then create the file wherever you want. I will create it in a directory called `tutorial` at the root of the Akka distribution, e.g. in `$AKKA_HOME/tutorial/Pi.scala`.

1.4.9 Creating the messages

The design we are aiming for is to have one `Master` actor initiating the computation, creating a set of `Worker` actors. Then it splits up the work into discrete chunks, and sends these chunks to the different workers in a round-robin fashion. The master waits until all the workers have completed their work and sent back results for

aggregation. When computation is completed the master prints out the result, shuts down all workers and then itself.

With this in mind, let's now create the messages that we want to have flowing in the system. We need three different messages:

- `Calculate` – sent to the `Master` actor to start the calculation
- `Work` – sent from the `Master` actor to the `Worker` actors containing the work assignment
- `Result` – sent from the `Worker` actors to the `Master` actor containing the result from the worker's calculation

Messages sent to actors should always be immutable to avoid sharing mutable state. In scala we have 'case classes' which make excellent messages. So let's start by creating three messages as case classes. We also create a common base trait for our messages (that we define as being sealed in order to prevent creating messages outside our control):

```
sealed trait PiMessage

case object Calculate extends PiMessage

case class Work(start: Int, nrOfElements: Int) extends PiMessage

case class Result(value: Double) extends PiMessage
```

1.4.10 Creating the worker

Now we can create the worker actor. This is done by mixing in the `Actor` trait and defining the `receive` method. The `receive` method defines our message handler. We expect it to be able to handle the `Work` message so we need to add a handler for this message:

```
class Worker extends Actor {
  def receive = {
    case Work(start, nrOfElements) =>
      self.reply(Result(calculatePiFor(start, nrOfElements)) // perform the work
  }
}
```

As you can see we have now created an `Actor` with a `receive` method as a handler for the `Work` message. In this handler we invoke the `calculatePiFor(...)` method, wrap the result in a `Result` message and send it back to the original sender using `self.reply`. In Akka the sender reference is implicitly passed along with the message so that the receiver can always reply or store away the sender reference for future use.

The only thing missing in our `Worker` actor is the implementation on the `calculatePiFor(...)` method. While there are many ways we can implement this algorithm in Scala, in this introductory tutorial we have chosen an imperative style using a for comprehension and an accumulator:

```
def calculatePiFor(start: Int, nrOfElements: Int): Double = {
  var acc = 0.0
  for (i <- start until (start + nrOfElements))
    acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
  acc
}
```

1.4.11 Creating the master

The master actor is a little bit more involved. In its constructor we need to create the workers (the `Worker` actors) and start them. We will also wrap them in a load-balancing router to make it easier to spread out the work evenly between the workers. Let's do that first:

```
// create the workers
val workers = Vector.fill(nrOfWorkers)(actorOf[Worker].start())

// wrap them with a load-balancing router
val router = Routing.loadBalancerActor(CyclicIterator(workers)).start()
```

As you can see we are using the `actorOf` factory method to create actors, this method returns as an `ActorRef` which is a reference to our newly created actor. This method is available in the `Actor` object but is usually imported:

```
import akka.actor.Actor.actorOf
```

There are two versions of `actorOf`; one of them taking a actor type and the other one an instance of an actor. The former one (`actorOf[MyActor]`) is used when the actor class has a no-argument constructor while the second one (`actorOf(new MyActor(...))`) is used when the actor class has a constructor that takes arguments. This is the only way to create an instance of an `Actor` and the `actorOf` method ensures this. The latter version is using call-by-name and lazily creates the actor within the scope of the `actorOf` method. The `actorOf` method instantiates the actor and returns, not an instance to the actor, but an instance to an `ActorRef`. This reference is the handle through which you communicate with the actor. It is immutable, serializable and location-aware meaning that it “remembers” its original actor even if it is sent to other nodes across the network and can be seen as the equivalent to the Erlang actor’s PID.

The actor’s life-cycle is:

- Created – `Actor.actorOf[MyActor]` – can **not** receive messages
- Started – `actorRef.start()` – can receive messages
- Stopped – `actorRef.stop()` – can **not** receive messages

Once the actor has been stopped it is dead and can not be started again.

Now we have a router that is representing all our workers in a single abstraction. If you paid attention to the code above, you saw that we were using the `nrOfWorkers` variable. This variable and others we have to pass to the `Master` actor in its constructor. So now let’s create the master actor. We have to pass in three integer variables:

- `nrOfWorkers` – defining how many workers we should start up
- `nrOfMessages` – defining how many number chunks to send out to the workers
- `nrOfElements` – defining how big the number chunks sent to each worker should be

Here is the master actor:

```
class Master(
  nrOfWorkers: Int, nrOfMessages: Int, nrOfElements: Int, latch: CountDownLatch)
  extends Actor {

  var pi: Double = _
  var nrOfResults: Int = _
  var start: Long = _

  // create the workers
  val workers = Vector.fill(nrOfWorkers)(actorOf[Worker].start())

  // wrap them with a load-balancing router
  val router = Routing.loadBalancerActor(CyclicIterator(workers)).start()

  def receive = { ... }

  override def preStart() {
    start = System.currentTimeMillis
  }

  override def postStop() {
    // tell the world that the calculation is complete
  }
}
```

```
println(
  "\n\tPi estimate: \t\t%s\n\tCalculation time: \t%s millis"
  .format(pi, (System.currentTimeMillis - start)))
latch.countDown()
}
}
```

A couple of things are worth explaining further.

First, we are passing in a `java.util.concurrent.CountDownLatch` to the `Master` actor. This latch is only used for plumbing (in this specific tutorial), to have a simple way of letting the outside world knowing when the master can deliver the result and shut down. In more idiomatic Akka code, as we will see in part two of this tutorial series, we would not use a latch but other abstractions and functions like `Channel`, `Future` and `?` to achieve the same thing in a non-blocking way. But for simplicity let's stick to a `CountDownLatch` for now.

Second, we are adding a couple of life-cycle callback methods; `preStart` and `postStop`. In the `preStart` callback we are recording the time when the actor is started and in the `postStop` callback we are printing out the result (the approximation of Pi) and the time it took to calculate it. In this call we also invoke `latch.countDown` to tell the outside world that we are done.

But we are not done yet. We are missing the message handler for the `Master` actor. This message handler needs to be able to react to two different messages:

- `Calculate` – which should start the calculation
- `Result` – which should aggregate the different results

The `Calculate` handler is sending out work to all the `Worker` actors and after doing that it also sends a `Broadcast(PoisonPill)` message to the router, which will send out the `PoisonPill` message to all the actors it is representing (in our case all the `Worker` actors). `PoisonPill` is a special kind of message that tells the receiver to shut itself down using the normal shutdown method; `self.stop`. We also send a `PoisonPill` to the router itself (since it's also an actor that we want to shut down).

The `Result` handler is simpler, here we get the value from the `Result` message and aggregate it to our `pi` member variable. We also keep track of how many results we have received back, and if that matches the number of tasks sent out, the `Master` actor considers itself done and shuts down.

Let's capture this in code:

```
// message handler
def receive = {
  case Calculate =>
    // schedule work
    for (i <- 0 until nrOfMessages) router ! Work(i * nrOfElements, nrOfElements)

    // send a PoisonPill to all workers telling them to shut down themselves
    router ! Broadcast(PoisonPill)

    // send a PoisonPill to the router, telling him to shut himself down
    router ! PoisonPill

  case Result(value) =>
    // handle result from the worker
    pi += value
    nrOfResults += 1
    if (nrOfResults == nrOfMessages) self.stop()
}
```

1.4.12 Bootstrap the calculation

Now the only thing that is left to implement is the runner that should bootstrap and run the calculation for us. We do that by creating an object that we call `Pi`, here we can extend the `App` trait in Scala, which means that we will be able to run this as an application directly from the command line.

The `Pi` object is a perfect container module for our actors and messages, so let's put them all there. We also create a method `calculate` in which we start up the `Master` actor and wait for it to finish:

```
object Pi extends App {

  calculate(nrOfWorkers = 4, nrOfElements = 10000, nrOfMessages = 10000)

  ... // actors and messages

  def calculate(nrOfWorkers: Int, nrOfElements: Int, nrOfMessages: Int) {

    // this latch is only plumbing to know when the calculation is completed
    val latch = new CountDownLatch(1)

    // create the master
    val master = actorOf(
      new Master(nrOfWorkers, nrOfMessages, nrOfElements, latch)).start()

    // start the calculation
    master ! Calculate

    // wait for master to shut down
    latch.await()
  }
}
```

That's it. Now we are done.

But before we package it up and run it, let's take a look at the full code now, with package declaration, imports and all:

```
package akka.tutorial.first.scala

import akka.actor.{Actor, PoisonPill}
import Actor._
import akka.routing.{Routing, CyclicIterator}
import Routing._

import java.util.concurrent.CountDownLatch

object Pi extends App {

  calculate(nrOfWorkers = 4, nrOfElements = 10000, nrOfMessages = 10000)

  // =====
  // ===== Messages =====
  // =====
  sealed trait PiMessage
  case object Calculate extends PiMessage
  case class Work(start: Int, nrOfElements: Int) extends PiMessage
  case class Result(value: Double) extends PiMessage

  // =====
  // ===== Worker =====
  // =====
  class Worker extends Actor {

    // define the work
    def calculatePiFor(start: Int, nrOfElements: Int): Double = {
      var acc = 0.0
      for (i <- start until (start + nrOfElements))
        acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
      acc
    }
  }
```



```

def receive = {
  case Work(start, nrOfElements) =>
    self reply Result(calculatePiFor(start, nrOfElements)) // perform the work
}

// =====
// ===== Master =====
// =====

class Master(
  nrOfWorkers: Int, nrOfMessages: Int, nrOfElements: Int, latch: CountdownLatch)
  extends Actor {

  var pi: Double = _
  var nrOfResults: Int = _
  var start: Long = _

  // create the workers
  val workers = Vector.fill(nrOfWorkers)(actorOf[Worker].start())

  // wrap them with a load-balancing router
  val router = Routing.loadBalancerActor(CyclicIterator(workers)).start()

  // message handler
  def receive = {
    case Calculate =>
      // schedule work
      //for (start <- 0 until nrOfMessages) router ! Work(start, nrOfElements)
      for (i <- 0 until nrOfMessages) router ! Work(i * nrOfElements, nrOfElements)

      // send a PoisonPill to all workers telling them to shut down themselves
      router ! Broadcast(PoisonPill)

      // send a PoisonPill to the router, telling him to shut himself down
      router ! PoisonPill

    case Result(value) =>
      // handle result from the worker
      pi += value
      nrOfResults += 1
      if (nrOfResults == nrOfMessages) self.stop()
  }

  override def preStart() {
    start = System.currentTimeMillis
  }

  override def postStop() {
    // tell the world that the calculation is complete
    println(
      "\n\tPi estimate: \t\t%s\n\tCalculation time: \t%s millis"
        .format(pi, (System.currentTimeMillis - start)))
    latch.countDown()
  }
}

// =====
// ===== Run it =====
// =====

def calculate(nrOfWorkers: Int, nrOfElements: Int, nrOfMessages: Int) {

  // this latch is only plumbing to know when the calculation is completed

```

```

val latch = new CountdownLatch(1)

// create the master
val master = actorOf(
  new Master(nrOfWorkers, nrOfMessages, nrOfElements, latch)).start()

// start the calculation
master ! Calculate

// wait for master to shut down
latch.await()
}
}

```

1.4.13 Run it as a command line application

If you have not typed in (or copied) the code for the tutorial as `$AKKA_HOME/tutorial/Pi.scala` then now is the time. When that's done open up a shell and step in to the Akka distribution (`cd $AKKA_HOME`).

First we need to compile the source file. That is done with Scala's compiler `scalac`. Our application depends on the `akka-actor-1.2.jar` JAR file, so let's add that to the compiler classpath when we compile the source:

```
$ scalac -cp lib/akka/akka-actor-1.2.jar tutorial/Pi.scala
```

When we have compiled the source file we are ready to run the application. This is done with `java` but yet again we need to add the `akka-actor-1.2.jar` JAR file to the classpath, and this time we also need to add the Scala runtime library `scala-library.jar` and the classes we compiled ourselves:

```

$ java \
  -cp lib/scala-library.jar:lib/akka/akka-actor-1.2.jar:. \
  akka.tutorial.first.scala.Pi
AKKA_HOME is defined as [/Users/jboner/tools/akka-actors-1.2]
loading config from [/Users/jboner/tools/akka-actors-1.2/config/akka.conf].

Pi estimate:          3.1435501812459323
Calculation time:     858 millis

```

Yippee! It is working.

If you have not defined the `AKKA_HOME` environment variable then Akka can't find the `akka.conf` configuration file and will print out a `Can't load akka.conf` warning. This is ok since it will then just use the defaults.

1.4.14 Run it inside SBT

If you used SBT, then you can run the application directly inside SBT. First you need to compile the project:

```

$ sbt
> compile
...

```

When this is done we can run our application directly inside SBT:

```

> run
...
Pi estimate:          3.1435501812459323
Calculation time:     942 millis

```

Yippee! It is working.

If you have not defined an the `AKKA_HOME` environment variable then Akka can't find the `akka.conf` configuration file and will print out a `Can't load akka.conf` warning. This is ok since it will then just use the defaults.

1.4.15 Conclusion

We have learned how to create our first Akka project using Akka's actors to speed up a computation-intensive problem by scaling out on multi-core processors (also known as scaling up). We have also learned to compile and run an Akka project using either the tools on the command line or the SBT build system.

If you have a multi-core machine then I encourage you to try out different number of workers (number of working actors) by tweaking the `nrOfWorkers` variable to for example; 2, 4, 6, 8 etc. to see performance improvement by scaling up.

Now we are ready to take on more advanced problems. In the next tutorial we will build on this one, refactor it into more idiomatic Akka and Scala code, and introduce a few new concepts and abstractions. Whenever you feel ready, join me in the Getting Started Tutorial: Second Chapter.

Happy hakking.

1.5 Getting Started Tutorial (Scala with Eclipse): First Chapter

1.5.1 Introduction

Welcome to the first tutorial on how to get started with [Akka](#) and [Scala](#). We assume that you already know what Akka and Scala are and will now focus on the steps necessary to start your first project. We will be using [Eclipse](#), and the [Scala plugin for Eclipse](#).

The sample application that we will create is using actors to calculate the value of Pi. Calculating Pi is a CPU intensive operation and we will utilize Akka Actors to write a concurrent solution that scales out to multi-core processors. This sample will be extended in future tutorials to use Akka Remote Actors to scale out on multiple machines in a cluster.

We will be using an algorithm that is called “embarrassingly parallel” which just means that each job is completely isolated and not coupled with any other job. Since this algorithm is so parallelizable it suits the actor model very well.

Here is the formula for the algorithm we will use:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots = \frac{\pi}{4}.$$

In this particular algorithm the master splits the series into chunks which are sent out to each worker actor to be processed. When each worker has processed its chunk it sends a result back to the master which aggregates the total result.

1.5.2 Tutorial source code

If you don't want to type in the code and/or set up an SBT project then you can check out the full tutorial from the Akka GitHub repository. It is in the `akka-tutorials/akka-tutorial-first` module. You can also browse it online [here](#), with the actual source code [here](#).

1.5.3 Prerequisites

This tutorial assumes that you have Java 1.6 or later installed on your machine and `java` on your `PATH`. You also need to know how to run commands in a shell (ZSH, Bash, DOS etc.) and a recent version of Eclipse (at least 3.6 - Helios).

If you want to run the example from the command line as well, you need to make sure that `$JAVA_HOME` environment variable is set to the root of the Java distribution. You also need to make sure that the `$JAVA_HOME/bin` is on your `PATH`:

```
$ export JAVA_HOME=..root of java distribution..
$ export PATH=$PATH:$JAVA_HOME/bin
```

You can test your installation by invoking `java`:

```
$ java -version
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07-334-10M3326)
Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02-334, mixed mode)
```

1.5.4 Downloading and installing Akka

To build and run the tutorial sample from the command line, you have to download Akka. If you prefer to use SBT to build and run the sample then you can skip this section and jump to the next one.

Let's get the `akka-actors-1.2.zip` distribution of Akka from <http://akka.io/downloads/> which includes everything we need for this tutorial. Once you have downloaded the distribution unzip it in the folder you would like to have Akka installed in. In my case I choose to install it in `/Users/jboner/tools/`, simply by unzipping it to this directory.

You need to do one more thing in order to install Akka properly: set the `AKKA_HOME` environment variable to the root of the distribution. In my case I'm opening up a shell, navigating down to the distribution, and setting the `AKKA_HOME` variable:

```
$ cd /Users/jboner/tools/akka-actors-1.2
$ export AKKA_HOME='pwd'
$ echo $AKKA_HOME
/Users/jboner/tools/akka-actors-1.2
```

The distribution looks like this:

```
$ ls -l
config
doc
lib
src
```

- In the `config` directory we have the Akka conf files.
- In the `doc` directory we have the documentation, API, doc JARs, and also the source files for the tutorials.
- In the `lib` directory we have the Scala and Akka JARs.
- In the `src` directory we have the source JARs for Akka.

The only JAR we will need for this tutorial (apart from the `scala-library.jar` JAR) is the `akka-actor-1.2.jar` JAR in the `lib/akka` directory. This is a self-contained JAR with zero dependencies and contains everything we need to write a system using Actors.

Akka is very modular and has many JARs for containing different features. The core distribution has seven modules:

- `akka-actor-1.2.jar` – Standard Actors
- `akka-typed-actor-1.2.jar` – Typed Actors

- akka-remote-1.2.jar – Remote Actors
- akka-stm-1.2.jar – STM (Software Transactional Memory), transactors and transactional datastructures
- akka-http-1.2.jar – Akka Mist for continuation-based asynchronous HTTP and also Jersey integration
- akka-slf4j-1.2.jar – SLF4J Event Handler Listener for logging with SLF4J
- akka-testkit-1.2.jar – Toolkit for testing Actors

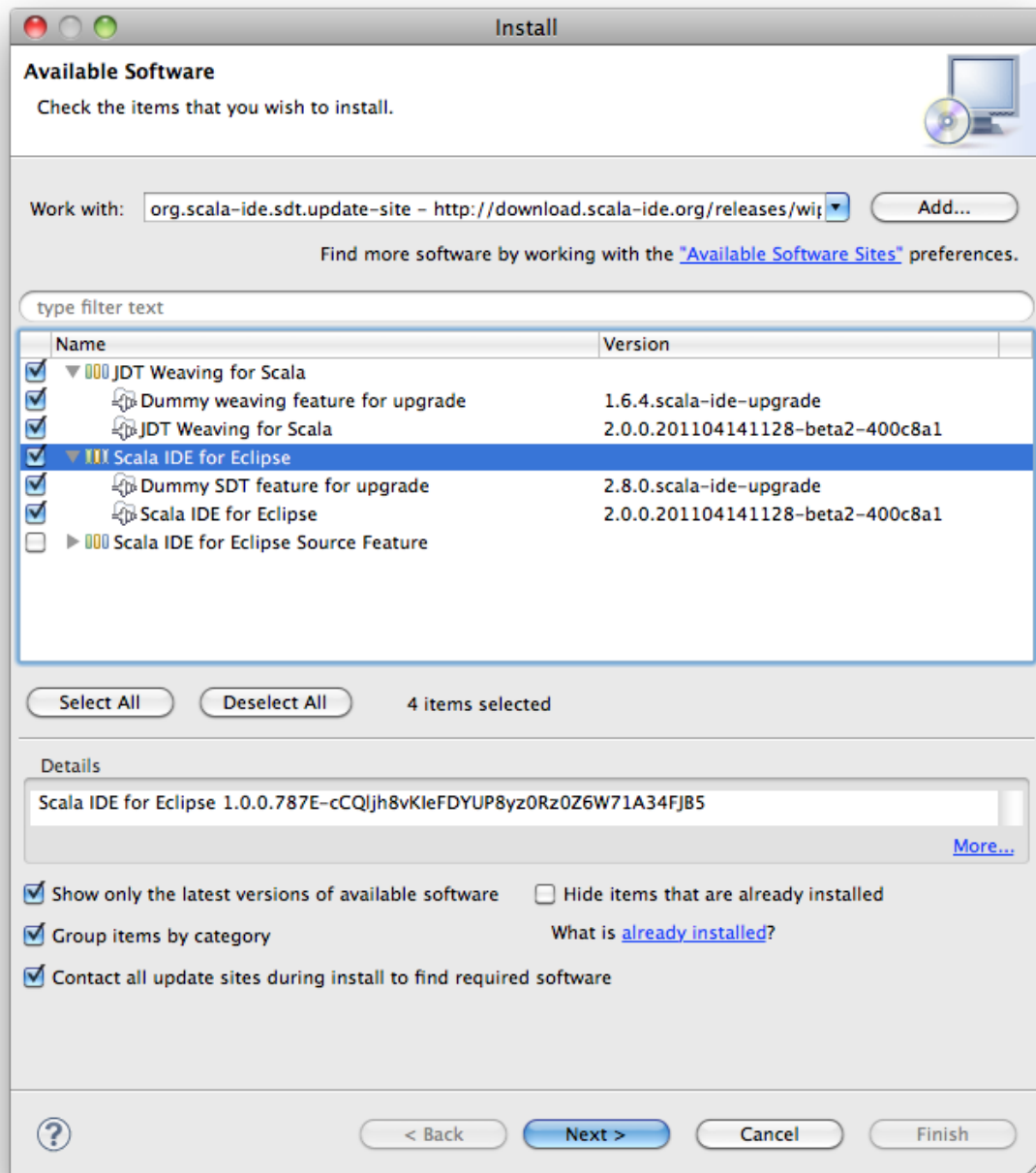
We also have Akka Modules containing add-on modules outside the core of Akka. You can download the Akka Modules distribution from <http://akka.io/downloads/>. It contains Akka core as well. We will not be needing any modules there today, but for your information the module JARs are these:

- akka-kernel-1.2.jar – Akka microkernel for running a bare-bones mini application server (embeds Jetty etc.)
- akka-amqp-1.2.jar – AMQP integration
- akka-camel-1.2.jar – Apache Camel Actors integration (it's the best way to have your Akka application communicate with the rest of the world)
- akka-camel-typed-1.2.jar – Apache Camel Typed Actors integration
- akka-scalaz-1.2.jar – Support for the Scalaz library
- akka-spring-1.2.jar – Spring framework integration
- akka- osgi-dependencies-bundle-1.2.jar – OSGi support

1.5.5 Downloading and installing the Scala IDE for Eclipse

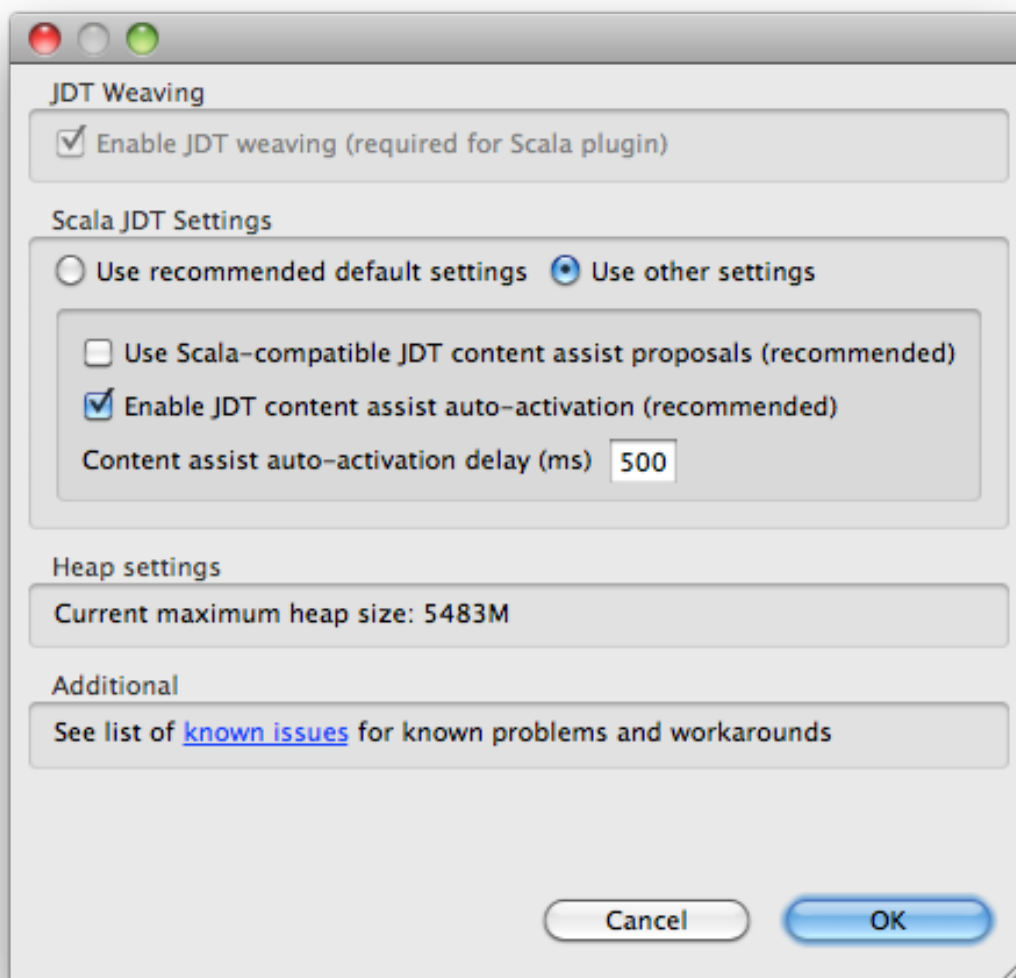
If you want to use Eclipse for coding your Akka tutorial, you need to install the Scala plugin for Eclipse. This plugin comes with its own version of Scala, so if you don't plan to run the example from the command line, you don't need to download the Scala distribution (and you can skip the next section).

You can install this plugin using the regular update mechanism. First choose a version of the IDE from <http://download.scala-ide.org>. We recommend you choose 2.0.x, which comes with Scala 2.9. Copy the corresponding URL and then choose Help/Install New Software and paste the URL you just copied. You should see something similar to the following image.



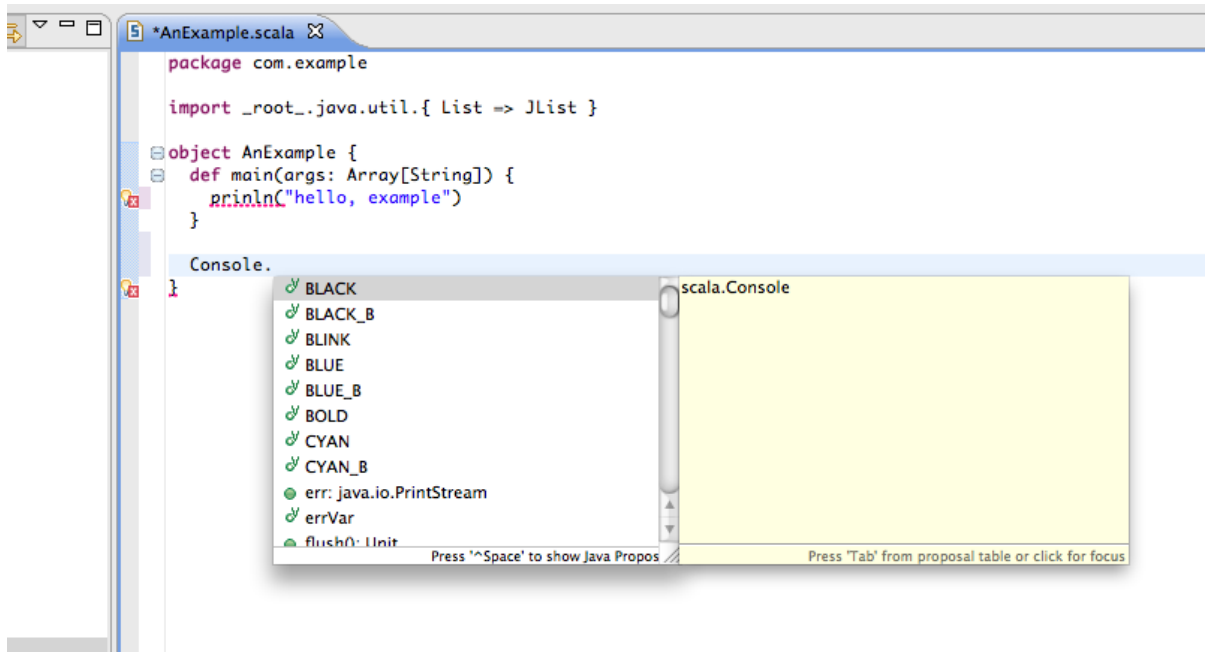
Make sure you select both the JDT Weaving for Scala and the Scala IDE for Eclipse plugins. The other plugin is optional, and contains the source code of the plugin itself.

Once the installation is finished, you need to restart Eclipse. The first time the plugin starts it will open a diagnostics window and offer to fix several settings, such as the delay for content assist (code-completion) or the shown completion proposal types.



Accept the recommended settings, and follow the instructions if you need to increase the heap size of Eclipse.

Check that the installation succeeded by creating a new Scala project (File/New>Scala Project), and typing some code. You should have content-assist, hyperlinking to definitions, instant error reporting, and so on.



You are ready to code now!

1.5.6 Downloading and installing Scala

To build and run the tutorial sample from the command line, you have to install the Scala distribution. If you prefer to use Eclipse to build and run the sample then you can skip this section and jump to the next one.

Scala can be downloaded from <http://www.scala-lang.org/downloads>. Browse there and download the Scala 2.9.1 release. If you pick the `tgz` or `zip` distribution then just unzip it where you want it installed. If you pick the IzPack Installer then double click on it and follow the instructions.

You also need to make sure that the `scala-2.9.1/bin` (if that is the directory where you installed Scala) is on your PATH:

```
$ export PATH=$PATH:scala-2.9.1/bin
```

You can test your installation by invoking `scala`:

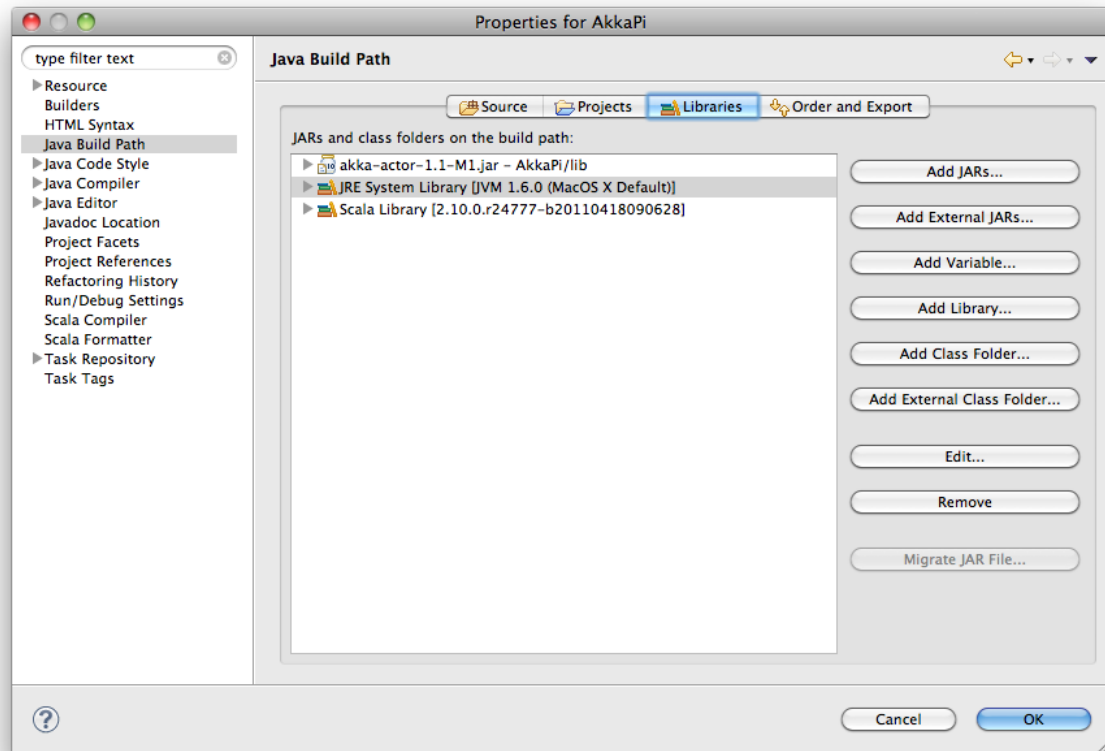
```
$ scala -version
Scala code runner version 2.9.1.final -- Copyright 2002-2011, LAMP/EPFL
```

Looks like we are all good. Finally let's create a source file `Pi.scala` for the tutorial and put it in the root of the Akka distribution in the `tutorial` directory (you have to create it first).

Some tools require you to set the `SCALA_HOME` environment variable to the root of the Scala distribution, however Akka does not require that.

1.5.7 Creating an Akka project in Eclipse

If you have not already done so, now is the time to create an Eclipse project for our tutorial. Use the `New Scala Project` wizard and accept the default settings. Once the project is open, we need to add the akka libraries to the *build path*. Right click on the project and choose `Properties`, then click on `Java Build Path`. Go to `Libraries` and click on `Add External Jars...`, then navigate to the location where you installed akka and choose `akka-actor.jar`. You should see something similar to this:



Using SBT in Eclipse

If you are an [SBT](#) user, you can follow the [Downloading and installing SBT](#) instruction and additionally install the `sbteclipse` plugin. This adds support for generating Eclipse project files from your SBT project. You need to install the plugin as described in the [README](#) of `sbteclipse`

Then run the `eclipse` target to generate the Eclipse project:

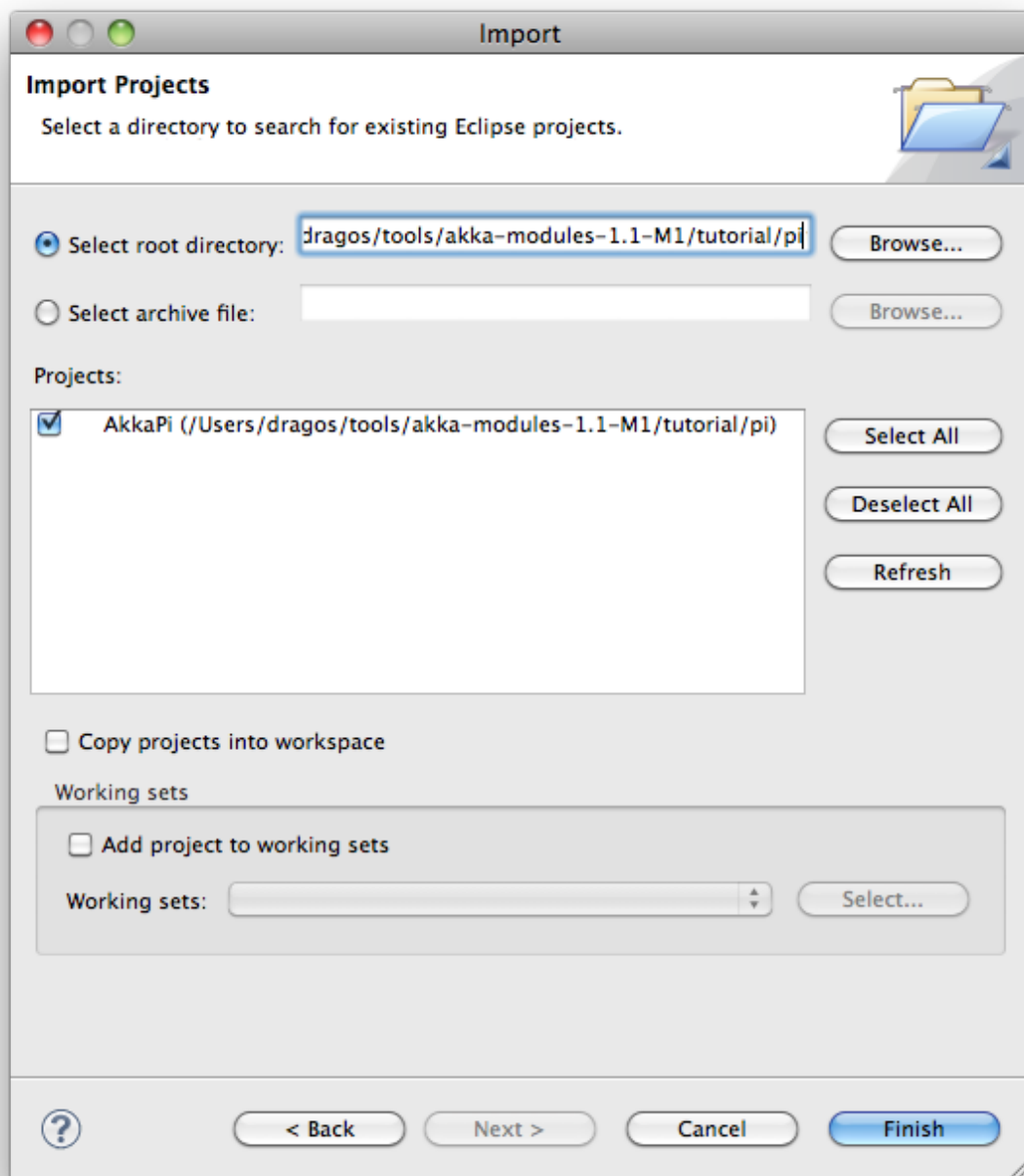
```
$ sbt
> eclipse
```

The options `create-src` and `with-sources` are useful:

```
$ sbt
> eclipse create-src with-sources
```

- `create-src` to create the common source directories, e.g. `src/main/scala`, `src/main/test`
- `with-sources` to create source attachments for the library dependencies

Next you need to import this project in Eclipse, by choosing `Eclipse/Import...` Existing Projects into Workspace. Navigate to the directory where you defined your SBT project and choose import:



Now we have the basis for an Akka Eclipse application, so we can..

1.5.8 Start writing the code

The design we are aiming for is to have one `Master` actor initiating the computation, creating a set of `Worker` actors. Then it splits up the work into discrete chunks, and sends these chunks to the different workers in a round-robin fashion. The master waits until all the workers have completed their work and sent back results for aggregation. When computation is completed the master prints out the result, shuts down all workers and then itself.

With this in mind, let's now create the messages that we want to have flowing in the system.

1.5.9 Creating the messages

We start by creating a package for our application, let's call it `akka.tutorial.first.scala`. We start by creating case classes for each type of message in our application, so we can place them in a hierarchy, call it `PiMessage`. Right click on the package and choose `New Scala Class`, and enter `PiMessage` for the name of the class.

We need three different messages:

- `Calculate` – sent to the `Master` actor to start the calculation
- `Work` – sent from the `Master` actor to the `Worker` actors containing the work assignment
- `Result` – sent from the `Worker` actors to the `Master` actor containing the result from the worker's calculation

Messages sent to actors should always be immutable to avoid sharing mutable state. In Scala we have 'case classes' which make excellent messages. So let's start by creating three messages as case classes. We also create a common base trait for our messages (that we define as being sealed in order to prevent creating messages outside our control):

```
package akka.tutorial.first.scala

sealed trait PiMessage

case object Calculate extends PiMessage

case class Work(start: Int, nrOfElements: Int) extends PiMessage

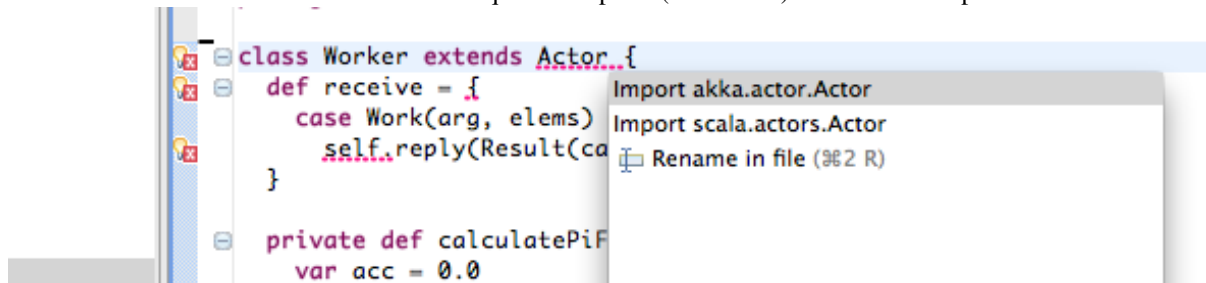
case class Result(value: Double) extends PiMessage
```

1.5.10 Creating the worker

Now we can create the worker actor. Create a new class called `Worker` as before. We need to mix in the `Actor` trait and defining the `receive` method. The `receive` method defines our message handler. We expect it to be able to handle the `Work` message so we need to add a handler for this message:

```
class Worker extends Actor {
  def receive = {
    case Work(start, nrOfElements) =>
      self reply Result(calculatePiFor(start, nrOfElements)) // perform the work
  }
}
```

The `Actor` trait is defined in `akka.actor` and you can either import it explicitly, or let Eclipse do it for you when it cannot resolve the `Actor` trait. The quick fix option (`Ctrl-F1`) will offer two options:



Choose the Akka Actor and move on.

As you can see we have now created an `Actor` with a `receive` method as a handler for the `Work` message. In this handler we invoke the `calculatePiFor(...)` method, wrap the result in a `Result` message and send it back to the original sender using `self.reply`. In Akka the sender reference is implicitly passed along with the message so that the receiver can always reply or store away the sender reference for future use.

The only thing missing in our `Worker` actor is the implementation on the `calculatePiFor(..)` method. While there are many ways we can implement this algorithm in Scala, in this introductory tutorial we have chosen an imperative style using a `for` comprehension and an accumulator:

```
def calculatePiFor(start: Int, nrOfElements: Int): Double = {
  var acc = 0.0
  for (i <- start until (start + nrOfElements))
    acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
  acc
}
```

1.5.11 Creating the master

Now create a new class for the master actor. The master actor is a little bit more involved. In its constructor we need to create the workers (the `Worker` actors) and start them. We will also wrap them in a load-balancing router to make it easier to spread out the work evenly between the workers. First we need to add some imports:

```
import akka.actor.{Actor, PoisonPill}
import akka.routing.{Routing, CyclicIterator}
import Routing._
import akka.dispatch.Dispatchers

import java.util.concurrent.CountDownLatch
```

and then we can create the workers:

```
// create the workers
val workers = Vector.fill(nrOfWorkers)(actorOf[Worker].start())

// wrap them with a load-balancing router
val router = Routing.loadBalancerActor(CyclicIterator(workers)).start()
```

As you can see we are using the `actorOf` factory method to create actors, this method returns as an `ActorRef` which is a reference to our newly created actor. This method is available in the `Actor` object but is usually imported:

```
import akka.actor.Actor.actorOf
```

There are two versions of `actorOf`; one of them taking a actor type and the other one an instance of an actor. The former one (`actorOf[MyActor]`) is used when the actor class has a no-argument constructor while the second one (`actorOf(new MyActor(..))`) is used when the actor class has a constructor that takes arguments. This is the only way to create an instance of an `Actor` and the `actorOf` method ensures this. The latter version is using call-by-name and lazily creates the actor within the scope of the `actorOf` method. The `actorOf` method instantiates the actor and returns, not an instance to the actor, but an instance to an `ActorRef`. This reference is the handle through which you communicate with the actor. It is immutable, serializable and location-aware meaning that it “remembers” its original actor even if it is sent to other nodes across the network and can be seen as the equivalent to the Erlang actor’s PID.

The actor’s life-cycle is:

- Created – `Actor.actorOf[MyActor]` – can **not** receive messages
- Started – `actorRef.start()` – can receive messages
- Stopped – `actorRef.stop()` – can **not** receive messages

Once the actor has been stopped it is dead and can not be started again.

Now we have a router that is representing all our workers in a single abstraction. If you paid attention to the code above, you saw that we were using the `nrOfWorkers` variable. This variable and others we have to pass to the `Master` actor in its constructor. So now let’s create the master actor. We have to pass in three integer variables:

- `nrOfWorkers` – defining how many workers we should start up

- `nrOfMessages` – defining how many number chunks to send out to the workers
- `nrOfElements` – defining how big the number chunks sent to each worker should be

Here is the master actor:

```
class Master(
    nrOfWorkers: Int, nrOfMessages: Int, nrOfElements: Int, latch: CountDownLatch)
    extends Actor {

    var pi: Double = _
    var nrOfResults: Int = _
    var start: Long = _

    // create the workers
    val workers = Vector.fill(nrOfWorkers)(actorOf[Worker].start())

    // wrap them with a load-balancing router
    val router = Routing.loadBalancerActor(CyclicIterator(workers)).start()

    def receive = { ... }

    override def preStart() {
        start = System.currentTimeMillis
    }

    override def postStop() {
        // tell the world that the calculation is complete
        println(
            "\n\tPi estimate: \t\t%s\n\tCalculation time: \t%s millis"
                .format(pi, (System.currentTimeMillis - start)))
        latch.countDown()
    }
}
```

A couple of things are worth explaining further.

First, we are passing in a `java.util.concurrent.CountDownLatch` to the Master actor. This latch is only used for plumbing (in this specific tutorial), to have a simple way of letting the outside world knowing when the master can deliver the result and shut down. In more idiomatic Akka code, as we will see in part two of this tutorial series, we would not use a latch but other abstractions and functions like `Channel`, `Future` and `?` to achieve the same thing in a non-blocking way. But for simplicity let's stick to a `CountDownLatch` for now.

Second, we are adding a couple of life-cycle callback methods; `preStart` and `postStop`. In the `preStart` callback we are recording the time when the actor is started and in the `postStop` callback we are printing out the result (the approximation of Pi) and the time it took to calculate it. In this call we also invoke `latch.countDown` to tell the outside world that we are done.

But we are not done yet. We are missing the message handler for the `Master` actor. This message handler needs to be able to react to two different messages:

- Calculate – which should start the calculation
- Result – which should aggregate the different results

The `Calculate` handler is sending out work to all the `Worker` actors and after doing that it also sends a `Broadcast(PoisonPill)` message to the router, which will send out the `PoisonPill` message to all the actors it is representing (in our case all the `Worker` actors). `PoisonPill` is a special kind of message that tells the receiver to shut itself down using the normal shutdown method; `self.stop`. We also send a `PoisonPill` to the router itself (since it's also an actor that we want to shut down).

The Result handler is simpler, here we get the value from the Result message and aggregate it to our pi member variable. We also keep track of how many results we have received back, and if that matches the number of tasks sent out, the Master actor considers itself done and shuts down.

Let's capture this in code:

```
// message handler
def receive = {
  case Calculate =>
    // schedule work
    for (i <- 0 until nrOfMessages) router ! Work(i * nrOfElements, nrOfElements)

    // send a PoisonPill to all workers telling them to shut down themselves
    router ! Broadcast(PoisonPill)

    // send a PoisonPill to the router, telling him to shut himself down
    router ! PoisonPill

  case Result(value) =>
    // handle result from the worker
    pi += value
    nrOfResults += 1
    if (nrOfResults == nrOfMessages) self.stop()
}
```

1.5.12 Bootstrap the calculation

Now the only thing that is left to implement is the runner that should bootstrap and run the calculation for us. We do that by creating an object that we call `Pi`, here we can extend the `App` trait in Scala, which means that we will be able to run this as an application directly from the command line or using the Eclipse Runner.

The `Pi` object is a perfect container module for our actors and messages, so let's put them all there. We also create a method `calculate` in which we start up the `Master` actor and wait for it to finish:

```
object Pi extends App {

  calculate(nrOfWorkers = 4, nrOfElements = 10000, nrOfMessages = 10000)

  ... // actors and messages

  def calculate(nrOfWorkers: Int, nrOfElements: Int, nrOfMessages: Int) {

    // this latch is only plumbing to know when the calculation is completed
    val latch = new CountDownLatch(1)

    // create the master
    val master = actorOf(
      new Master(nrOfWorkers, nrOfMessages, nrOfElements, latch)).start()

    // start the calculation
    master ! Calculate

    // wait for master to shut down
    latch.await()
  }
}
```

That's it. Now we are done.

1.5.13 Run it from Eclipse

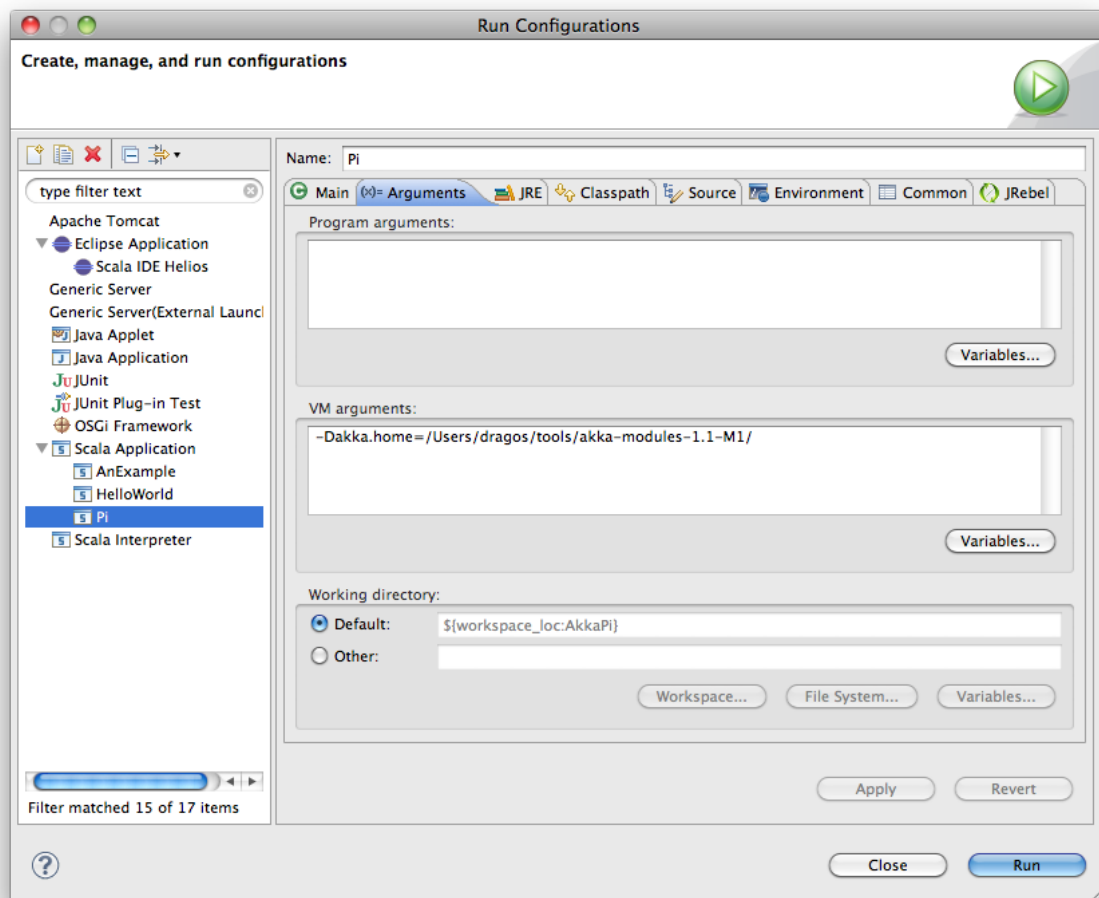
Eclipse builds your project on every save when `Project/Build Automatically` is set. If not, bring your project up to date by clicking `Project/Build Project`. If there are no compilation errors, you can right-click in the editor where `Pi` is defined, and choose `Run as... /Scala application`. If everything works fine, you should see:

```
AKKA_HOME is defined as [/Users/jboner/tools/akka-actors-1.2]
loading config from [/Users/jboner/tools/akka-actors-1.2/config/akka.conf].
```

```
Pi estimate:      3.1435501812459323
Calculation time: 858 millis
```

If you have not defined an the `AKKA_HOME` environment variable then Akka can't find the `akka.conf` configuration file and will print out a `Can't load akka.conf` warning. This is ok since it will then just use the defaults.

You can also define a new Run configuration, by going to Run/Run Configurations. Create a new Scala application and choose the tutorial project and the main class to be `akkatutorial.Pi`. You can pass additional command line arguments to the JVM on the `Arguments` page, for instance to define where `akka.conf` is:



Once you finished your run configuration, click `Run`. You should see the same output in the `Console` window. You can use the same configuration for debugging the application, by choosing `Run/Debug History` or just `Debug As`.

1.5.14 Conclusion

We have learned how to create our first Akka project using Akka's actors to speed up a computation-intensive problem by scaling out on multi-core processors (also known as scaling up). We have also learned to compile and run an Akka project using Eclipse.

If you have a multi-core machine then I encourage you to try out different number of workers (number of working actors) by tweaking the `nrOfWorkers` variable to for example; 2, 4, 6, 8 etc. to see performance improvement by scaling up.

Now we are ready to take on more advanced problems. In the next tutorial we will build on this one, refactor it into more idiomatic Akka and Scala code, and introduce a few new concepts and abstractions. Whenever you feel ready, join me in the Getting Started Tutorial: Second Chapter.

Happy hacking.

1.6 Getting Started Tutorial (Java): First Chapter

1.6.1 Introduction

Welcome to the first tutorial on how to get started with [Akka](#) and Java. We assume that you already know what Akka and Java are and will now focus on the steps necessary to start your first project.

There are two variations of this first tutorial:

- creating a standalone project and run it from the command line
- creating a Maven project and running it from within Maven

Since they are so similar we will present them both.

The sample application that we will create is using actors to calculate the value of Pi. Calculating Pi is a CPU intensive operation and we will utilize Akka Actors to write a concurrent solution that scales out to multi-core processors. This sample will be extended in future tutorials to use Akka Remote Actors to scale out on multiple machines in a cluster.

We will be using an algorithm that is called “embarrassingly parallel” which just means that each job is completely isolated and not coupled with any other job. Since this algorithm is so parallelizable it suits the actor model very well.

Here is the formula for the algorithm we will use:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots = \frac{\pi}{4}.$$

In this particular algorithm the master splits the series into chunks which are sent out to each worker actor to be processed. When each worker has processed its chunk it sends a result back to the master which aggregates the total result.

1.6.2 Tutorial source code

If you don't want to type in the code and/or set up a Maven project then you can check out the full tutorial from the Akka GitHub repository. It is in the `akka-tutorials/akka-tutorial-first` module. You can also browse it online [here](#), with the actual source code [here](#).

To check out the code using Git invoke the following:

```
$ git clone git://github.com/jboner/akka.git
```

Then you can navigate down to the tutorial:

```
$ cd akka/akka-tutorials/akka-tutorial-first
```

1.6.3 Prerequisites

This tutorial assumes that you have Java 1.6 or later installed on your machine and `java` on your `PATH`. You also need to know how to run commands in a shell (ZSH, Bash, DOS etc.) and a decent text editor or IDE to type in the Java code.

You need to make sure that `$JAVA_HOME` environment variable is set to the root of the Java distribution. You also need to make sure that the `$JAVA_HOME/bin` is on your `PATH`:

```
$ export JAVA_HOME=..root of java distribution..
$ export PATH=$PATH:$JAVA_HOME/bin
```

You can test your installation by invoking `java`:

```
$ java -version
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07-334-10M3326)
Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02-334, mixed mode)
```

1.6.4 Downloading and installing Akka

To build and run the tutorial sample from the command line, you have to download Akka. If you prefer to use SBT to build and run the sample then you can skip this section and jump to the next one.

Let's get the `akka-actors-1.2.zip` distribution of Akka from <http://akka.io/downloads/> which includes everything we need for this tutorial. Once you have downloaded the distribution unzip it in the folder you would like to have Akka installed in. In my case I choose to install it in `/Users/jboner/tools/`, simply by unzipping it to this directory.

You need to do one more thing in order to install Akka properly: set the `AKKA_HOME` environment variable to the root of the distribution. In my case I'm opening up a shell, navigating down to the distribution, and setting the `AKKA_HOME` variable:

```
$ cd /Users/jboner/tools/akka-actors-1.2
$ export AKKA_HOME=`pwd`
$ echo $AKKA_HOME
/Users/jboner/tools/akka-actors-1.2
```

The distribution looks like this:

```
$ ls -l
config
doc
lib
src
```

- In the `config` directory we have the Akka conf files.
- In the `doc` directory we have the documentation, API, doc JARs, and also the source files for the tutorials.
- In the `lib` directory we have the Scala and Akka JARs.
- In the `src` directory we have the source JARs for Akka.

The only JAR we will need for this tutorial (apart from the `scala-library.jar` JAR) is the `akka-actor-1.2.jar` JAR in the `lib/akka` directory. This is a self-contained JAR with zero dependencies and contains everything we need to write a system using Actors.

Akka is very modular and has many JARs for containing different features. The core distribution has seven modules:

- `akka-actor-1.2.jar` – Standard Actors
- `akka-typed-actor-1.2.jar` – Typed Actors
- `akka-remote-1.2.jar` – Remote Actors
- `akka-stm-1.2.jar` – STM (Software Transactional Memory), transactors and transactional datastructures
- `akka-http-1.2.jar` – Akka Mist for continuation-based asynchronous HTTP and also Jersey integration

- akka-slf4j-1.2.jar – SLF4J Event Handler Listener for logging with SLF4J
- akka-testkit-1.2.jar – Toolkit for testing Actors

We also have Akka Modules containing add-on modules outside the core of Akka. You can download the Akka Modules distribution from <http://akka.io/downloads/>. It contains Akka core as well. We will not be needing any modules there today, but for your information the module JARs are these:

- akka-kernel-1.2.jar – Akka microkernel for running a bare-bones mini application server (embeds Jetty etc.)
- akka-amqp-1.2.jar – AMQP integration
- akka-camel-1.2.jar – Apache Camel Actors integration (it's the best way to have your Akka application communicate with the rest of the world)
- akka-camel-typed-1.2.jar – Apache Camel Typed Actors integration
- akka-scalaz-1.2.jar – Support for the Scalaz library
- akka-spring-1.2.jar – Spring framework integration
- akka- osgi-dependencies-bundle-1.2.jar – OSGi support

1.6.5 Downloading and installing Maven

Maven is an excellent build system that can be used to build both Java and Scala projects. If you want to use Maven for this tutorial then follow the following instructions, if not you can skip this section and the next.

First browse to <http://maven.apache.org/download.html> and download the 3.0.3 distribution.

To install Maven it is easiest to follow the instructions on <http://maven.apache.org/download.html#Installation>.

1.6.6 Creating an Akka Maven project

If you have not already done so, now is the time to create a Maven project for our tutorial. You do that by stepping into the directory you want to create your project in and invoking the `mvn` command:

```
$ mvn archetype:generate \
  -DgroupId=akka.tutorial.first.java \
  -DartifactId=akka-tutorial-first-java \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DinteractiveMode=false
```

Now we have the basis for our Maven-based Akka project. Let's step into the project directory:

```
$ cd akka-tutorial-first-java
```

Here is the layout that Maven created:

```
akka-tutorial-first-jboner
|-- pom.xml
'-- src
    |-- main
    |   '-- java
    |       '-- akka
    |           '-- tutorial
    |               '-- first
    |                   '-- java
    |                       '-- App.java
```

As you can see we already have a Java source file called `App.java`, let's now rename it to `Pi.java`.

We also need to edit the `pom.xml` build file. Let's add the dependency we need as well as the Maven repository it should download it from. The Akka Maven repository can be found at <http://akka.io/repository> and Typesafe

provides <http://repo.typesafe.com/typesafe/releases/> that proxies several other repositories, including akka.io. It should now look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <name>akka-tutorial-first-java</name>
  <groupId>akka.tutorial.first.java</groupId>
  <artifactId>akka-tutorial-first-java</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <url>http://akka.io</url>

  <dependencies>
    <dependency>
      <groupId>se.scalablesolutions.akka</groupId>
      <artifactId>akka-actor</artifactId>
      <version>1.2</version>
    </dependency>
  </dependencies>

  <repositories>
    <repository>
      <id>typesafe</id>
      <name>Typesafe Repository</name>
      <url>http://repo.typesafe.com/typesafe/releases/</url>
    </repository>
  </repositories>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

1.6.7 Start writing the code

Now it's about time to start hacking.

We start by creating a `Pi.java` file and adding these import statements at the top of the file:

```
package akka.tutorial.first.java;

import static akka.actor.ActorOf;
import static akka.actor.ActorOf.poissonPill;
import static java.util.Arrays.asList;

import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.actor.UntypedActorFactory;
```

```
import akka.routing.CyclicIterator;
import akka.routing.InfiniteIterator;
import akka.routing.Routing.Broadcast;
import akka.routing.UntypedLoadBalancer;

import java.util.concurrent.CountDownLatch;
```

If you are using Maven in this tutorial then create the file in the `src/main/java/akka/tutorial/first/java` directory.

If you are using the command line tools then create the file wherever you want. I will create it in a directory called `tutorial` at the root of the Akka distribution, e.g. in `$AKKA_HOME/tutorial/akka/tutorial/first/java/Pi.java`.

1.6.8 Creating the messages

The design we are aiming for is to have one `Master` actor initiating the computation, creating a set of `Worker` actors. Then it splits up the work into discrete chunks, and sends these chunks to the different workers in a round-robin fashion. The master waits until all the workers have completed their work and sent back results for aggregation. When computation is completed the master prints out the result, shuts down all workers and then itself.

With this in mind, let's now create the messages that we want to have flowing in the system. We need three different messages:

- `Calculate` – sent to the `Master` actor to start the calculation
- `Work` – sent from the `Master` actor to the `Worker` actors containing the work assignment
- `Result` – sent from the `Worker` actors to the `Master` actor containing the result from the worker's calculation

Messages sent to actors should always be immutable to avoid sharing mutable state. So let's start by creating three messages as immutable POJOs. We also create a wrapper `Pi` class to hold our implementation:

```
public class Pi {

    static class Calculate {}

    static class Work {
        private final int start;
        private final int nrOfElements;

        public Work(int start, int nrOfElements) {
            this.start = start;
            this.nrOfElements = nrOfElements;
        }

        public int getStart() { return start; }
        public int getNrOfElements() { return nrOfElements; }
    }

    static class Result {
        private final double value;

        public Result(double value) {
            this.value = value;
        }

        public double getValue() { return value; }
    }
}
```

1.6.9 Creating the worker

Now we can create the worker actor. This is done by extending in the `UntypedActor` base class and defining the `onReceive` method. The `onReceive` method defines our message handler. We expect it to be able to handle the `Work` message so we need to add a handler for this message:

```
static class Worker extends UntypedActor {

    // message handler
    public void onReceive(Object message) {
        if (message instanceof Work) {
            Work work = (Work) message;

            // perform the work
            double result = calculatePiFor(work.getStart(), work.getNrOfElements());

            // reply with the result
            getContext().replyUnsafe(new Result(result));

        } else throw new IllegalArgumentException("Unknown message [" + message + "]");
    }
}
```

As you can see we have now created an `UntypedActor` with a `onReceive` method as a handler for the `Work` message. In this handler we invoke the `calculatePiFor(..)` method, wrap the result in a `Result` message and send it back to the original sender using `getContext().replyUnsafe(..)`. In Akka the sender reference is implicitly passed along with the message so that the receiver can always reply or store away the sender reference for future use.

The only thing missing in our `Worker` actor is the implementation on the `calculatePiFor(..)` method:

```
// define the work
private double calculatePiFor(int start, int nrOfElements) {
    double acc = 0.0;
    for (int i = start * nrOfElements; i <= ((start + 1) * nrOfElements - 1); i++) {
        acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1);
    }
    return acc;
}
```

1.6.10 Creating the master

The master actor is a little bit more involved. In its constructor we need to create the workers (the `Worker` actors) and start them. We will also wrap them in a load-balancing router to make it easier to spread out the work evenly between the workers. Let's do that first:

```
static class Master extends UntypedActor {
    ...

    static class PiRouter extends UntypedLoadBalancer {
        private final InfiniteIterator<ActorRef> workers;

        public PiRouter(ActorRef[] workers) {
            this.workers = new CyclicIterator<ActorRef>(asList(workers));
        }

        public InfiniteIterator<ActorRef> seq() {
            return workers;
        }
    }

    public Master(...) {
```

```

...

// create the workers
final ActorRef[] workers = new ActorRef[nrOfWorkers];
for (int i = 0; i < nrOfWorkers; i++) {
    workers[i] = actorOf(Worker.class).start();
}

// wrap them with a load-balancing router
ActorRef router = actorOf(new UntypedActorFactory() {
    public UntypedActor create() {
        return new PiRouter(workers);
    }
}).start();
}
}

```

As you can see we are using the `actorOf` factory method to create actors, this method returns as an `ActorRef` which is a reference to our newly created actor. This method is available in the `Actors` object but is usually imported:

```
import static akka.actor.Actors.actorOf;
```

One thing to note is that we used two different versions of the `actorOf` method. For creating the `Worker` actor we just pass in the class but to create the `PiRouter` actor we can't do that since the constructor in the `PiRouter` class takes arguments, instead we need to use the `UntypedActorFactory` which unfortunately is a bit more verbose.

`actorOf` is the only way to create an instance of an `Actor`, this is enforced by Akka runtime. The `actorOf` method instantiates the actor and returns, not an instance to the actor, but an instance to an `ActorRef`. This reference is the handle through which you communicate with the actor. It is immutable, serializable and location-aware meaning that it "remembers" its original actor even if it is sent to other nodes across the network and can be seen as the equivalent to the Erlang actor's PID.

The actor's life-cycle is:

- Created – `Actor.actorOf[MyActor]` – can **not** receive messages
- Started – `actorRef.start()` – can receive messages
- Stopped – `actorRef.stop()` – can **not** receive messages

Once the actor has been stopped it is dead and can not be started again.

Now we have a router that is representing all our workers in a single abstraction. If you paid attention to the code above, you saw that we were using the `nrOfWorkers` variable. This variable and others we have to pass to the `Master` actor in its constructor. So now let's create the master actor. We have to pass in three integer variables:

- `nrOfWorkers` – defining how many workers we should start up
- `nrOfMessages` – defining how many number chunks to send out to the workers
- `nrOfElements` – defining how big the number chunks sent to each worker should be

Here is the master actor:

```

static class Master extends UntypedActor {
    private final int nrOfMessages;
    private final int nrOfElements;
    private final CountdownLatch latch;

    private double pi;
    private int nrOfResults;
    private long start;

    private ActorRef router;
}

```


But we are not done yet. We are missing the message handler for the `Master` actor. This message handler needs to be able to react to two different messages:

- `Calculate` – which should start the calculation
- `Result` – which should aggregate the different results

The `Calculate` handler is sending out work to all the `Worker` actors and after doing that it also sends a new `Broadcast(poisonPill())` message to the router, which will send out the `PoisonPill` message to all the actors it is representing (in our case all the `Worker` actors). `PoisonPill` is a special kind of message that tells the receiver to shut itself down using the normal shutdown method; `getContext().stop()`, and is created through the `poisonPill()` method. We also send a `PoisonPill` to the router itself (since it's also an actor that we want to shut down).

The `Result` handler is simpler, here we get the value from the `Result` message and aggregate it to our `pi` member variable. We also keep track of how many results we have received back, and if that matches the number of tasks sent out, the `Master` actor considers itself done and shuts down.

Let's capture this in code:

```
// message handler
public void onReceive(Object message) {

    if (message instanceof Calculate) {
        // schedule work
        for (int start = 0; start < nrOfMessages; start++) {
            router.tell(new Work(start, nrOfElements), getContext());
        }

        // send a PoisonPill to all workers telling them to shut down themselves
        router.tell(new Broadcast(poisonPill()));

        // send a PoisonPill to the router, telling him to shut himself down
        router.tell(poisonPill());

    } else if (message instanceof Result) {

        // handle result from the worker
        Result result = (Result) message;
        pi += result.getValue();
        nrOfResults += 1;
        if (nrOfResults == nrOfMessages) getContext().stop();

    } else throw new IllegalArgumentException("Unknown message [" + message + "]");
}
```

1.6.11 Bootstrap the calculation

Now the only thing that is left to implement is the runner that should bootstrap and run the calculation for us. We do that by adding a `main` method to the enclosing `Pi` class in which we create a new instance of `Pi` and invoke method `calculate` in which we start up the `Master` actor and wait for it to finish:

```
public class Pi {

    public static void main(String[] args) throws Exception {
        Pi pi = new Pi();
        pi.calculate(4, 10000, 10000);
    }

    public void calculate(final int nrOfWorkers, final int nrOfElements, final int nrOfMessages)
        throws Exception {

        // this latch is only plumbing to know when the calculation is completed
```



```

    final CountDownLatch latch = new CountDownLatch(1);

    // create the master
    ActorRef master = actorOf(new UntypedActorFactory() {
        public UntypedActor create() {
            return new Master(nrOfWorkers, nrOfMessages, nrOfElements, latch);
        }
    }).start();

    // start the calculation
    master.tell(new Calculate());

    // wait for master to shut down
    latch.await();
}
}

```

That's it. Now we are done.

Before we package it up and run it, let's take a look at the full code now, with package declaration, imports and all:

```

package akka.tutorial.first.java;

import static akka.actor.ActorRefs.actorOf;
import static akka.actor.ActorRefs.poisonPill;
import static java.util.Arrays.asList;

import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.actor.UntypedActorFactory;
import akka.routing.CyclicIterator;
import akka.routing.InfiniteIterator;
import akka.routing.Routing.Broadcast;
import akka.routing.UntypedLoadBalancer;

import java.util.concurrent.CountDownLatch;

public class Pi {

    public static void main(String[] args) throws Exception {
        Pi pi = new Pi();
        pi.calculate(4, 10000, 10000);
    }

    // =====
    // ===== Messages =====
    // =====
    static class Calculate {}

    static class Work {
        private final int start;
        private final int nrOfElements;

        public Work(int start, int nrOfElements) {
            this.start = start;
            this.nrOfElements = nrOfElements;
        }

        public int getStart() { return start; }
        public int getNrOfElements() { return nrOfElements; }
    }
}

```

```

static class Result {
    private final double value;

    public Result(double value) {
        this.value = value;
    }

    public double getValue() { return value; }
}

// =====
// ===== Worker =====
// =====
static class Worker extends UntypedActor {

    // define the work
    private double calculatePiFor(int start, int nrOfElements) {
        double acc = 0.0;
        for (int i = start * nrOfElements; i <= ((start + 1) * nrOfElements - 1); i++) {
            acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1);
        }
        return acc;
    }

    // message handler
    public void onReceive(Object message) {
        if (message instanceof Work) {
            Work work = (Work) message;

            // perform the work
            double result = calculatePiFor(work.getStart(), work.getNrOfElements());

            // reply with the result
            getContext().replyUnsafe(new Result(result));

        } else throw new IllegalArgumentException("Unknown message [" + message + "]");
    }
}

// =====
// ===== Master =====
// =====
static class Master extends UntypedActor {
    private final int nrOfMessages;
    private final int nrOfElements;
    private final CountDownLatch latch;

    private double pi;
    private int nrOfResults;
    private long start;

    private ActorRef router;

    static class PiRouter extends UntypedLoadBalancer {
        private final InfiniteIterator<ActorRef> workers;

        public PiRouter(ActorRef[] workers) {
            this.workers = new CyclicIterator<ActorRef>(asList(workers));
        }

        public InfiniteIterator<ActorRef> seq() {
            return workers;
        }
    }
}

```



```
// =====
// ===== Run it =====
// =====
public void calculate(final int nrOfWorkers, final int nrOfElements, final int nrOfMessages)
    throws Exception {

    // this latch is only plumbing to know when the calculation is completed
    final CountDownLatch latch = new CountDownLatch(1);

    // create the master
    ActorRef master = actorOf(new UntypedActorFactory() {
        public UntypedActor create() {
            return new Master(nrOfWorkers, nrOfMessages, nrOfElements, latch);
        }
    }).start();

    // start the calculation
    master.tell(new Calculate());

    // wait for master to shut down
    latch.await();
}
}
```

1.6.12 Run it as a command line application

If you have not typed in (or copied) the code for the tutorial as `$AKKA_HOME/tutorial/akka/tutorial/first/java/Pi.java` then now is the time. When that's done open up a shell and step in to the Akka distribution (`cd $AKKA_HOME`).

First we need to compile the source file. That is done with Java's compiler `javac`. Our application depends on the `akka-actor-1.2.jar` and the `scala-library.jar` JAR files, so let's add them to the compiler classpath when we compile the source:

```
$ javac -cp lib/scala-library.jar:lib/akka/akka-actor-1.2.jar tutorial/akka/tutorial/first/java/Pi.java
```

When we have compiled the source file we are ready to run the application. This is done with `java` but yet again we need to add the `akka-actor-1.2.jar` and the `scala-library.jar` JAR files to the classpath as well as the classes we compiled ourselves:

```
$ java \
  -cp lib/scala-library.jar:lib/akka/akka-actor-1.2.jar:tutorial \
  akka.tutorial.java.first.Pi
AKKA_HOME is defined as [/Users/jboner/tools/akka-actors-1.2]
loading config from [/Users/jboner/tools/akka-actors-1.2/config/akka.conf].

Pi estimate:      3.1435501812459323
Calculation time: 822 millis
```

Yippee! It is working.

If you have not defined the `AKKA_HOME` environment variable then Akka can't find the `akka.conf` configuration file and will print out a `Can't load akka.conf` warning. This is ok since it will then just use the defaults.

1.6.13 Run it inside Maven

If you used Maven, then you can run the application directly inside Maven. First you need to compile the project:

```
$ mvn compile
```

When this is done we can run our application directly inside Maven:

```
$ mvn exec:java -Dexec.mainClass="akka.tutorial.first.java.Pi"
...
Pi estimate:      3.1435501812459323
Calculation time: 939 millis
```

Yippee! It is working.

If you have not defined an the `AKKA_HOME` environment variable then Akka can't find the `akka.conf` configuration file and will print out a `Can't load akka.conf` warning. This is ok since it will then just use the defaults.

1.6.14 Conclusion

We have learned how to create our first Akka project using Akka's actors to speed up a computation-intensive problem by scaling out on multi-core processors (also known as scaling up). We have also learned to compile and run an Akka project using either the tools on the command line or the SBT build system.

If you have a multi-core machine then I encourage you to try out different number of workers (number of working actors) by tweaking the `nrOfWorkers` variable to for example; 2, 4, 6, 8 etc. to see performance improvement by scaling up.

Now we are ready to take on more advanced problems. In the next tutorial we will build on this one, refactor it into more idiomatic Akka and Scala code, and introduce a few new concepts and abstractions. Whenever you feel ready, join me in the Getting Started Tutorial: Second Chapter.

Happy hacking.

1.7 Use-case and Deployment Scenarios

1.7.1 How can I use and deploy Akka?

Akka can be used in two different ways:

- As a library: used as a regular JAR on the classpath and/or in a web app, to be put into `WEB-INF/lib`
- As a microkernel: stand-alone microkernel, embedding a servlet container along with many other services

Using Akka as library

This is most likely what you want if you are building Web applications. There are several ways you can use Akka in Library mode by adding more and more modules to the stack.

Actors as services

The simplest way you can use Akka is to use the actors as services in your Web application. All that's needed to do that is to put the Akka charts as well as its dependency jars into `WEB-INF/lib`. You also need to put the `akka.conf` config file in the `$AKKA_HOME/config` directory. Now you can create your Actors as regular services referenced from your Web application. You should also be able to use the Remoting service, e.g. be able to make certain Actors remote on other hosts. Please note that remoting service does not speak HTTP over port 80, but a custom protocol over the port is specified in `akka.conf`.

Using Akka as a stand alone microkernel

Akka can also be run as a stand-alone microkernel. It implements a full enterprise stack. See the [Add-on Modules](#) for more information.

Using the Akka sbt plugin to package your application

The Akka sbt plugin can create a full Akka microkernel deployment for your sbt project.

To use the plugin, first add a plugin definition to your SBT project by creating `project/plugins/Plugins.scala` with:

```
import sbt._

class Plugins(info: ProjectInfo) extends PluginDefinition(info) {
  val akkaRepo = "Akka Repo" at "http://akka.io/repository"
  val akkaPlugin = "se.scalablesolutions.akka" % "akka-sbt-plugin" % "1.2"
}
```

Then mix the `AkkaKernelProject` trait into your project definition. For example:

```
class MyProject(info: ProjectInfo) extends DefaultProject(info) with AkkaKernelProject
```

This will automatically add all the Akka dependencies needed for a microkernel deployment (download them with `sbt update`).

Place your config files in `src/main/config`.

To build a microkernel deployment use the `dist` task:

```
sbt dist
```

1.8 Examples of use-cases for Akka

There is a great discussion on use-cases for Akka with some good write-ups by production users [here](#)

1.8.1 Here are some of the areas where Akka is being deployed into production

Transaction processing (Online Gaming, Finance/Banking, Trading, Statistics, Betting, Social Media, Telecom)

Scale up, scale out, fault-tolerance / HA

Service backend (any industry, any app)

Service REST, SOAP, Cometd, WebSockets etc Act as message hub / integration layer Scale up, scale out, fault-tolerance / HA

Concurrency/parallelism (any app)

Correct Simple to work with and understand Just add the jars to your existing JVM project (use Scala, Java, Groovy or JRuby)

Simulation

Master/Worker, Compute Grid, MapReduce etc.

Batch processing (any industry)

Camel integration to hook up with batch data sources Actors divide and conquer the batch workloads

Communications Hub (Telecom, Web media, Mobile media)

Scale up, scale out, fault-tolerance / HA

Gaming and Betting (MOM, online gaming, betting)

Scale up, scale out, fault-tolerance / HA

Business Intelligence/Data Mining/general purpose crunching

Scale up, scale out, fault-tolerance / HA

Complex Event Stream Processing

Scale up, scale out, fault-tolerance / HA

GENERAL

2.1 Akka and the Java Memory Model

Prior to Java 5, the Java Memory Model (JMM) was broken. It was possible to get all kinds of strange results like unpredictable merged writes made by concurrent executing threads, unexpected reordering of instructions, and even final fields were not guaranteed to be final. With Java 5 and JSR-133, the Java Memory Model is clearly specified. This specification makes it possible to write code that performs, but doesn't cause concurrency problems. The Java Memory Model is specified in 'happens before'-rules, e.g.:

- **monitor lock rule:** a release of a lock happens before every subsequent acquire of the same lock.
- **volatile variable rule:** a write of a volatile variable happens before every subsequent read of the same volatile variable

The 'happens before'-rules clearly specify which visibility guarantees are provided on memory and which reorderings are allowed. Without these rules it would not be possible to write concurrent and performant code in Java.

2.1.1 Actors and the Java Memory Model

With the Actors implementation in Akka, there are 2 ways multiple threads can execute actions on shared memory over time:

- if a message is send to an actor (e.g. by another actor). In most cases messages are immutable, but if that message is not a properly constructed immutable object, without happens before rules, the system still could be subject to instruction re-orderings and visibility problems (so a possible source of concurrency errors).
- if an actor makes changes to its internal state in one 'receive' method and access that state while processing another message. With the actors model you don't get any guarantee that the same thread will be executing the same actor for different messages. Without a happens before relation between these actions, there could be another source of concurrency errors.

To solve the 2 problems above, Akka adds the following 2 'happens before'-rules to the JMM:

- **the actor send rule:** where the send of the message to an actor happens before the receive of the **same** actor.
- **the actor subsequent processing rule:** where processing of one message happens before processing of the next message by the **same** actor.

Both rules only apply for the same actor instance and are not valid if different actors are used.

2.1.2 STM and the Java Memory Model

The Akka STM also provides a happens before rule called:

- **the transaction rule:** a commit on a transaction happens before every subsequent start of a transaction where there is at least 1 shared reference.

How these rules are realized in Akka, is an implementation detail and can change over time (the exact details could even depend on the used configuration) but they will lift on the other JMM rules like the monitor lock rule or the volatile variable rule. Essentially this means that you, the Akka user, do not need to worry about adding synchronization to provide such a happens before relation, because it is the responsibility of Akka. So you have your hands free to deal with your problems and not that of the framework.

2.2 Configuration

Contents

- [Specifying the configuration file](#)
- [Defining the configuration file](#)
- [Specifying files for different modes](#)
- [Including files](#)
- [Showing Configuration Source](#)
- [Summary of System Properties](#)

2.2.1 Specifying the configuration file

If you don't specify a configuration file then Akka uses default values, corresponding to the `akka-reference.conf` that you see below. You can specify your own configuration file to override any property in the reference config. You only have to define the properties that differ from the default configuration.

The location of the config file to use can be specified in various ways:

- Define the `-Dakka.config=...` system property parameter with a file path to configuration file.
- Put an `akka.conf` file in the root of the classpath.
- Define the `AKKA_HOME` environment variable pointing to the root of the Akka distribution. The config is taken from the `AKKA_HOME/config/akka.conf`. You can also point to the `AKKA_HOME` by specifying the `-Dakka.home=...` system property parameter.

If several of these ways to specify the config file are used at the same time the precedence is the order as given above, i.e. you can always redefine the location with the `-Dakka.config=...` system property.

2.2.2 Defining the configuration file

Here is the reference configuration file:

```
#####
# Akka Config File #
#####

# This file has all the default settings, so all these could be removed with no visible effect.
# Modify as needed.

akka {
  version = "1.2"    # Akka version, checked against the runtime version of Akka.

  enabled-modules = []      # Comma separated list of the enabled modules. Options: ["remote", "

  time-unit = "seconds"    # Time unit for all timeout properties throughout the config

  event-handlers = ["akka.event.EventHandler$DefaultListener"] # event handlers to register at boot
```

```

event-handler-level = "INFO" # Options: ERROR, WARNING, INFO, DEBUG

# These boot classes are loaded (and created) automatically when the Akka Microkernel boots up
#   Can be used to bootstrap your application(s)
#   Should be the FQN (Fully Qualified Name) of the boot class which needs to have a default constructor
# boot = ["sample.camel.Boot",
#         "sample.rest.java.Boot",
#         "sample.rest.scala.Boot",
#         "sample.security.Boot"]
boot = []

actor {
  timeout = 5 # Default timeout for Future based invocations
              # - Actor: ? and ask
              # - UntypedActor: ask
              # - TypedActor: methods with non-void return type

  serialize-messages = off # Does a deep clone of (non-primitive) messages to ensure immutability
  throughput = 5 # Default throughput for all ExecutorBasedEventDrivenDispatchers
  throughput-deadline-time = -1 # Default throughput deadline for all ExecutorBasedEventDrivenDispatchers
  dispatcher-shutdown-timeout = 1 # Using the akka.time-unit, how long dispatchers by default wait for tasks

  default-dispatcher {
    type = "GlobalExecutorBasedEventDriven" # Must be one of the following, all "Global*" are non-fair
                                           # - ExecutorBasedEventDriven
                                           # - ExecutorBasedEventDrivenWorkStealing
                                           # - GlobalExecutorBasedEventDriven

    keep-alive-time = 60 # Keep alive time for threads
    core-pool-size-factor = 1.0 # No of core threads ... ceil(available processors * factor)
    max-pool-size-factor = 4.0 # Max no of threads ... ceil(available processors * factor)
    executor-bounds = -1 # Makes the Executor bounded, -1 is unbounded
    task-queue-size = -1 # Specifies the bounded capacity of the task queue (< 1 == unbounded)
    task-queue-type = "linked" # Specifies which type of task queue will be used, can be "linked", "fifo", "lifo"
    allow-core-timeout = on # Allow core threads to time out
    rejection-policy = "caller-runs" # abort, caller-runs, discard-oldest, discard
    throughput = 5 # Throughput for ExecutorBasedEventDrivenDispatcher, set to 0 to disable
    throughput-deadline-time = -1 # Throughput deadline for ExecutorBasedEventDrivenDispatcher
    mailbox-capacity = -1 # If negative (or zero) then an unbounded mailbox is used
                        # If positive then a bounded mailbox is used and the capacity is the value
                        # NOTE: setting a mailbox to 'blocking' can be a bit dangerous
                        #       could lead to deadlock, use with care
                        #
                        # The following are only used for ExecutorBasedEventDriven
                        # and only if mailbox-capacity > 0

    mailbox-push-timeout-time = 10 # Specifies the timeout to add a new message to a mailbox
                                  # (in unit defined by the time-unit property)
  }

  debug {
    receive = "false" # enable function of Actor.loggable(), which is
                     # to log any received message at DEBUG level
    autoreceive = "false" # enable DEBUG logging of all AutoReceiveMessages
                          # (Kill, PoisonPill and the like)
    lifecycle = "false" # enable DEBUG logging of actor lifecycle changes
  }
}

stm {
  fair = on # Should global transactions be fair or non-fair (non fair yield better performance)
  max-retries = 1000
  timeout = 5 # Default timeout for blocking transactions and transaction set (in unit defined by the time-unit property)
  write-skew = true
  blocking-allowed = false
}

```

```

interruptible    = false
speculative     = true
quick-release   = true
propagation     = "requires"
trace-level     = "none"
}

http {
  hostname = "localhost"
  port = 9998

  #If you are using akka.http.AkkaRestServlet
  filters = ["akka.security.AkkaSecurityFilterFactory"] # List with all jersey filters to use
  # resource-packages = ["sample.rest.scala",
  #                      "sample.rest.java",
  #                      "sample.security"] # List with all resource packages for your Jersey s
  resource-packages = []

  # The authentication service to use. Need to be overridden (sample now)
  # authenticator = "sample.security.BasicAuthenticationService"
  authenticator = "N/A"

  # Uncomment if you are using the KerberosAuthenticationActor
  # kerberos {
  #   servicePrincipal = "HTTP/localhost@EXAMPLE.COM"
  #   keyTabLocation   = "URL to keytab"
  #   kerberosDebug    = "true"
  #   realm            = "EXAMPLE.COM"
  # }
  kerberos {
    servicePrincipal = "N/A"
    keyTabLocation   = "N/A"
    kerberosDebug    = "N/A"
    realm            = ""
  }

  #If you are using akka.http.AkkaMistServlet
  mist-dispatcher {
    #type = "GlobalExecutorBasedEventDriven" # Uncomment if you want to use a different dispatcher
  }
  connection-close = true           # toggles the addition of the "Connection" response header
  root-actor-id = "_httproot"       # the id of the actor to use as the root endpoint
  root-actor-builtin = true         # toggles the use of the built-in root endpoint base class
  timeout = 1000                    # the default timeout for all async requests (in ms)
  expired-header-name = "Async-Timeout" # the name of the response header to use when an async request times out
  expired-header-value = "expired"    # the value of the response header to use when an async request times out
}

remote {

  # secure-cookie = "050E0A0D0D06010A00000900040D060F0C09060B" # generate your own with '$AKKA_COOKIE'
  secure-cookie = ""

  compression-scheme = "" # Options: "zlib" (lzf to come), leave out for no compression
  zlib-compression-level = 6 # Options: 0-9 (1 being fastest and 9 being the most compressed),
  layer = "akka.remote.netty.NettyRemoteSupport"

  server {
    hostname = "localhost" # The hostname or IP that clients should connect to
    port = 2552            # The port clients should connect to. Default is 2552 (AKKA)
    message-frame-size = 1048576 # Increase this if you want to be able to send messages with 1MB
    connection-timeout = 100    # Number in time-unit
  }
}

```

```

require-cookie = off          # Should the remote server require that it peers share the same
untrusted-mode = off          # Enable untrusted mode for full security of server managed act
backlog = 4096                 # Sets the size of the connection backlog
execution-pool-keepalive = 60 # Length in akka.time-unit how long core threads will be kept a
execution-pool-size           = 16 # Size of the core pool of the remote execution unit
max-channel-memory-size      = 0 # Maximum channel size, 0 for off
max-total-memory-size        = 0 # Maximum total size of all channels, 0 for off
}

client {
  buffering {
    retry-message-send-on-failure = off # Buffer outbound messages when send failed, if off y
    capacity = -1                     # If negative (or zero) then an unbounded mailbox is us
                                     # If positive then a bounded mailbox is used and the ca
  }
  reconnect-delay = 5             # Number in time-unit
  read-timeout = 10               # Number in time-unit
  message-frame-size = 1048576    # Size in bytes
  reap-futures-delay = 5          # Number in time-unit
  reconnection-time-window = 600 # Maximum time window that a client should try to reconnect
}
}

test {
  timefactor = "1.0"             # factor by which to scale timeouts during tests, e.g. to account for sl
}
}

```

A custom `akka.conf` might look like this:

```

# In this file you can override any option defined in the 'akka-reference.conf' file.
# Copy in all or parts of the 'akka-reference.conf' file and modify as you please.

akka {
  event-handlers = ["akka.event.slf4j.Slf4jEventHandler"]

  # Comma separated list of the enabled modules.
  enabled-modules = ["camel", "remote"]

  # These boot classes are loaded (and created) automatically when the Akka Microkernel boots up
  # Can be used to bootstrap your application(s)
  # Should be the FQN (Fully Qualified Name) of the boot class which needs to have a default c
  boot = ["sample.camel.Boot",
         "sample.myservice.Boot"]

  actor {
    throughput = 10 # Throughput for ExecutorBasedEventDrivenDispatcher, set to 1 for complete f
  }

  remote {
    server {
      port = 2562 # The port clients should connect to. Default is 2552 (AKKA)
    }
  }
}

```

2.2.3 Specifying files for different modes

You can use different configuration files for different purposes by specifying a mode option, either as `-Dakka.mode=...` system property or as `AKKA_MODE=...` environment variable. For example using `DEBUG` log level when in development mode. Run with `-Dakka.mode=dev` and place the following

`akka.dev.conf` in the root of the classpath.

`akka.dev.conf`:

```
akka {
  event-handler-level = "DEBUG"
}
```

The mode option works in the same way when using configuration files in `AKKA_HOME/config/` directory.

The mode option is not used when specifying the configuration file with `-Dakka.config=...` system property.

2.2.4 Including files

Sometimes it can be useful to include another configuration file, for example if you have one `akka.conf` with all environment independent settings and then override some settings for specific modes.

`akka.dev.conf`:

```
include "akka.conf"

akka {
  event-handler-level = "DEBUG"
}
```

2.2.5 Showing Configuration Source

If the system property `akka.output.config.source` is set to anything but null, then the source from which Akka reads its configuration is printed to the console during application startup.

2.2.6 Summary of System Properties

- *akka.home* (`AKKA_HOME`): where Akka searches for configuration
- *akka.config*: explicit configuration file location
- *akka.mode* (`AKKA_MODE`): modify configuration file name for multiple profiles
- *akka.output.config.source*: whether to print configuration source to console

2.3 Event Handler

There is an Event Handler which takes the place of a logging system in Akka:

```
akka.event.EventHandler
```

You can configure which event handlers should be registered at boot time. That is done using the ‘event-handlers’ element in `akka.conf`. Here you can also define the log level.

```
akka {
  # event handlers to register at boot time (EventHandler$DefaultListener logs to STDOUT)
  event-handlers = ["akka.event.EventHandler$DefaultListener"]
  event-handler-level = "DEBUG" # Options: ERROR, WARNING, INFO, DEBUG
}
```

The default one logs to STDOUT and is registered by default. It is not intended to be used for production. There is also an *SLF4J* event handler available in the ‘akka-slf4j’ module.

Example of creating a listener from Scala (from Java you just have to create an ‘UntypedActor’ and create a handler for these messages):

```
val errorHandlerEventListener = Actor.actorOf(new Actor {
  self.dispatcher = EventHandler.EventHandlerDispatcher

  def receive = {
    case EventHandler.Error(cause, instance, message) => ...
    case EventHandler.Warning(instance, message) => ...
    case EventHandler.Info(instance, message) => ...
    case EventHandler.Debug(instance, message) => ...
    case genericEvent => ...
  }
})
```

To add the listener:

```
EventHandler.addListener(errorHandlerEventListener)
```

To remove the listener:

```
EventHandler.removeListener(errorHandlerEventListener)
```

To log an event:

```
EventHandler.notify(EventHandler.Error(exception, this, message))

EventHandler.notify(EventHandler.Warning(this, message))

EventHandler.notify(EventHandler.Info(this, message))

EventHandler.notify(EventHandler.Debug(this, message))

EventHandler.notify(object)
```

You can also use one of the direct methods (for a bit better performance):

```
EventHandler.error(exception, this, message)

EventHandler.error(this, message)

EventHandler.warning(this, message)

EventHandler.info(this, message)

EventHandler.debug(this, message)
```

The event handler allows you to send an arbitrary object to the handler which you can handle in your event handler listener. The default listener prints it's toString String out to STDOUT.

```
EventHandler.notify(anyRef)
```

The methods take a call-by-name parameter for the message to avoid object allocation and execution if level is disabled. The following formatting function will not be evaluated if level is INFO, WARNING, or ERROR.

```
EventHandler.debug(this, "Processing took %s ms".format(duration))
```

From Java you need to nest the call in an if statement to achieve the same thing.

```
if (EventHandler.isDebugEnabled()) {
  EventHandler.debug(this, String.format("Processing took %s ms", duration));
}
```

2.4 SLF4J

This module is available in the ‘akka-slf4j.jar’. It has one single dependency; the slf4j-api jar. In runtime you also need a SLF4J backend, we recommend:

```
lazy val logback = "ch.qos.logback" % "logback-classic" % "0.9.28" % "runtime"
```

2.4.1 Event Handler

This module includes a SLF4J Event Handler that works with Akka’s standard Event Handler. You enabled it in the ‘event-handlers’ element in akka.conf. Here you can also define the log level.

```
akka {  
  event-handlers = ["akka.event.slf4j.Slf4jEventHandler"]  
  event-handler-level = "DEBUG"  
}
```

Read more about how to use the [Event Handler](#).

COMMON UTILITIES

3.1 Scheduler

Module stability: **SOLID**

Akka has a little scheduler written using actors. This can be convenient if you want to schedule some periodic task for maintenance or similar.

It allows you to register a message that you want to be sent to a specific actor at a periodic interval.

3.1.1 Here is an example:

```
import akka.actor.Scheduler

//Sends messageToBeSent to receiverActor after initialDelayBeforeSending and then after each delay
Scheduler.schedule(receiverActor, messageToBeSent, initialDelayBeforeSending, delayBetweenMessages)

//Sends messageToBeSent to receiverActor after delayUntilSend
Scheduler.scheduleOnce(receiverActor, messageToBeSent, delayUntilSend, timeUnit)
```

3.2 Duration

Module stability: **SOLID**

Durations are used throughout the Akka library, wherefore this concept is represented by a special data type, `Duration`. Values of this type may represent infinite (`Duration.Inf`, `Duration.MinusInf`) or finite durations.

3.2.1 Scala

In Scala durations are constructable using a mini-DSL and support all expected operations:

```
import akka.util.duration._ // notice the small d

val fivesec = 5.seconds
val threemillis = 3.millis
val diff = fivesec - threemillis
assert (diff < fivesec)
val fourmillis = threemillis * 4 / 3 // though you cannot write it the other way around
val n = threemillis / (1 millisecond)
```


Note: You may leave out the dot if the expression is clearly delimited (e.g. within parentheses or in an argument list), but it is recommended to use it if the time unit is the last token on a line, otherwise semi-colon inference might go wrong, depending on what starts the next line.

3.2.2 Java

Java provides less syntactic sugar, so you have to spell out the operations as method calls instead:

```
final Duration fivesec = Duration.create(5, "seconds");
final Duration threemillis = Duration.parse("3 millis");
final Duration diff = fivesec.minus(threemillis);
assert (diff.lt(fivesec));
assert (Duration.Zero().lt(Duration.Inf()));
```

SCALA API

4.1 Actors (Scala)

Contents

- Creating Actors
 - Defining an Actor class
 - Creating Actors
 - Creating Actors with non-default constructor
 - Running a block of code asynchronously
- Actor Internal API
 - Start Hook
 - Restart Hooks
 - Stop Hook
- Identifying Actors
- Messages and immutability
- Send messages
 - Fire-forget
 - Send-And-Receive-Future
 - Send-And-Receive-Eventually
 - Forward message
- Receive messages
- Reply to messages
 - Reply using the channel
 - Reply using the reply and reply_? methods
 - Summary of reply semantics and options
- Initial receive timeout
- Starting actors
- Stopping actors
- PoisonPill
- HotSwap
 - Upgrade
- Encoding Scala Actors nested receives without accidentally leaking memory: UnnestedReceive
 - Downgrade
- Killing an Actor
- Actor life-cycle
- Extending Actors using PartialFunction chaining

Module stability: **SOLID**

The **Actor Model** provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct

concurrent and parallel systems. Actors were defined in the 1973 paper by Carl Hewitt but have been popularized by the Erlang language, and used for example at Ericsson with great success to build highly concurrent and reliable telecom systems.

The API of Akka's Actors is similar to Scala Actors which has borrowed some of its syntax from Erlang.

The Akka 0.9 release introduced a new concept; ActorRef, which requires some refactoring. If you are new to Akka just read along, but if you have used Akka 0.6.x, 0.7.x and 0.8.x then you might be helped by the [0.8.x => 0.9.x migration guide](#)

4.1.1 Creating Actors

Actors can be created either by:

- Extending the Actor class and implementing the receive method.
- Create an anonymous actor using one of the actor methods.

Defining an Actor class

Actor classes are implemented by extending the Actor class and implementing the receive method. The receive method should define a series of case statements (which has the type PartialFunction[Any, Unit]) that defines which messages your Actor can handle, using standard Scala pattern matching, along with the implementation of how the messages should be processed.

Here is an example:

```
import akka.actor.Actor
import akka.event.EventHandler

class MyActor extends Actor {
  def receive = {
    case "test" => EventHandler.info(this, "received test")
    case _ => EventHandler.info(this, "received unknown message")
  }
}
```

Please note that the Akka Actor receive message loop is exhaustive, which is different compared to Erlang and Scala Actors. This means that you need to provide a pattern match for all messages that it can accept and if you want to be able to handle unknown messages then you need to have a default case as in the example above.

Creating Actors

```
val myActor = Actor.actorOf[MyActor]
myActor.start()
```

Normally you would want to import the actorOf method like this:

```
import akka.actor.Actor._

val myActor = actorOf[MyActor]
```

To avoid prefixing it with Actor every time you use it.

You can also start it in the same statement:

```
val myActor = actorOf[MyActor].start()
```

The call to actorOf returns an instance of ActorRef. This is a handle to the Actor instance which you can use to interact with the Actor. The ActorRef is immutable and has a one to one relationship with the Actor it represents. The ActorRef is also serializable and network-aware. This means that you can serialize it, send

it over the wire and use it on a remote host and it will still be representing the same Actor on the original node, across the network.

Creating Actors with non-default constructor

If your Actor has a constructor that takes parameters then you can't create it using `actorOf[TYPE]`. Instead you can use a variant of `actorOf` that takes a call-by-name block in which you can create the Actor in any way you like.

Here is an example:

```
val a = actorOf(new MyActor(..)).start() // allows passing in arguments into the MyActor constructor
```

Running a block of code asynchronously

Here we create a light-weight actor-based thread, that can be used to spawn off a task. Code blocks spawned up like this are always implicitly started, shut down and made eligible for garbage collection. The actor that is created “under the hood” is not reachable from the outside and there is no way of sending messages to it. It being an actor is only an implementation detail. It will only run the block in an event-based thread and exit once the block has run to completion.

```
spawn {
  ... // do stuff
}
```

4.1.2 Actor Internal API

The Actor trait defines only one abstract method, the abovementioned `receive`. In addition, it offers two convenience methods `become/unbecome` for modifying the hotswap behavior stack as described in [HotSwap](#) and the self reference to this actor's `ActorRef` object. If the current actor behavior does not match a received message, `unhandled` is called, which by default throws an `UnhandledMessageException`.

The remaining visible methods are user-overridable life-cycle hooks which are described in the following:

```
def preStart() {}
def preRestart(cause: Throwable, message: Option[Any]) {}
def freshInstance(): Option[Actor] = None
def postRestart(cause: Throwable) {}
def postStop() {}
```

The implementations shown above are the defaults provided by the Actor trait.

Note: There is still the single-argument method `preRestart(cause: Throwable)`, which in fact is called by the default implementation of the two-argument variant. This method will be removed in version 2.0; you should add the second (dummy) argument to your actors before upgrading.

Start Hook

Right after starting the actor, its `preStart` method is invoked. This is guaranteed to happen before the first message from external sources is queued to the actor's mailbox.

```
override def preStart {
  // e.g. send initial message to self
  self ! GetMeStarted
  // or do any other stuff, e.g. registering with other actors
  someService ! Register(self)
}
```

Restart Hooks

A supervised actor, i.e. one which is linked to another actor with a fault handling strategy, will be restarted in case an exception is thrown while processing a message. This restart involves four of the hooks mentioned above:

1. The old actor is informed by calling `preRestart` with the exception which caused the restart and the message which triggered that exception; the latter may be `None` if the restart was not caused by processing a message, e.g. when a supervisor does not trap the exception and is restarted in turn by its supervisor. This method is the best place for cleaning up, preparing hand-over to the fresh actor instance, etc.
2. The old actor's `freshInstance` factory method is invoked, which may optionally produce the new actor instance which will replace this actor. If this method returns `None` or throws an exception, the initial factory from the `Actor.actorOf` call is used to produce the fresh instance.
3. The new actor's `preStart` method is invoked, just as in the normal start-up case.
4. The new actor's `postRestart` method is called with the exception which caused the restart.

Warning: The `freshInstance` hook may be used to propagate (part of) the failed actor's state to the fresh instance. This carries the risk of proliferating the cause for the crash which triggered the restart. If you are tempted to take this route, it is strongly advised to step back and consider other possible approaches, e.g. distributing the state in question using other means or spawning short-lived worker actors for carrying out “risky” tasks.

An actor restart replaces only the actual actor object; the contents of the mailbox and the hotswap stack are unaffected by the restart, so processing of messages will resume after the `postRestart` hook returns. Any message sent to an actor while it is being restarted will be queued to its mailbox as usual.

Stop Hook

After stopping an actor, its `postStop` hook is called, which may be used e.g. for deregistering this actor from other services. This hook is guaranteed to run after message queuing has been disabled for this actor, i.e. sending messages would fail with an `IllegalActorStateException`.

4.1.3 Identifying Actors

Each Actor has two fields:

- `self.uuid`
- `self.id`

The difference is that the `uuid` is generated by the runtime, guaranteed to be unique and can't be modified. While the `id` is modifiable by the user, and defaults to the Actor class name. You can retrieve Actors by both UUID and ID using the `ActorRegistry`, see the section further down for details.

4.1.4 Messages and immutability

IMPORTANT: Messages can be any kind of object but have to be immutable. Scala can't enforce immutability (yet) so this has to be by convention. Primitives like `String`, `Int`, `Boolean` are always immutable. Apart from these the recommended approach is to use Scala case classes which are immutable (if you don't explicitly expose the state) and works great with pattern matching at the receiver side.

Here is an example:

```
// define the case class
case class Register(user: User)

// create a new case class message
val message = Register(user)
```

Other good messages types are `scala.Tuple2`, `scala.List`, `scala.Map` which are all immutable and great for pattern matching.

4.1.5 Send messages

Messages are sent to an Actor through one of the following methods.

- `!` means “fire-and-forget”, e.g. send a message asynchronously and return immediately.
- `?` sends a message asynchronously and returns a `Future` representing a possible reply.

Note: There used to be two more “bang” methods, which are deprecated and will be removed in Akka 2.0:

- `!!` was similar to the current `(actor ? msg).as[T]`; deprecation followed from the change of timeout handling described below.
 - `!!![T]` was similar to the current `(actor ? msg).mapTo[T]`, with the same change in the handling of `Future`’s timeout as for `!!`, but additionally the old method could defer possible type cast problems into seemingly unrelated parts of the code base.
-

Fire-forget

This is the preferred way of sending messages. No blocking waiting for a message. This gives the best concurrency and scalability characteristics.

```
actor ! "Hello"
```

If invoked from within an Actor, then the sending actor reference will be implicitly passed along with the message and available to the receiving Actor in its `channel: UntypedChannel` member field. The target actor can use this to reply to the original sender, e.g. by using the `self.reply(message: Any)` method.

If invoked from an instance that is **not** an Actor there will be no implicit sender passed along with the message and you will get an `IllegalActorStateException` when calling `self.reply(...)`.

Send-And-Receive-Future

Using `?` will send a message to the receiving Actor asynchronously and will return a `Future`:

```
val future = actor ? "Hello"
```

The receiving actor should reply to this message, which will complete the future with the reply message as value; if the actor throws an exception while processing the invocation, this exception will also complete the future. If the actor does not complete the future, it will expire after the timeout period, which is taken from one of the following three locations in order of precedence:

1. explicitly given timeout as in `actor ? ("hello") (timeout = 12 millis)`
2. implicit argument of type `Actor.Timeout`, e.g.

```
implicit val timeout = Actor.Timeout(12 millis)
val future = actor ? "hello"
```

3. default timeout from `akka.conf`

See [Futures \(Scala\)](#) for more information on how to await or query a future.

Send-And-Receive-Eventually

The future returned from the `? method` can conveniently be passed around or chained with further processing steps, but sometimes you just need the value, even if that entails waiting for it (but keep in mind that waiting inside an actor is prone to dead-locks, e.g. if obtaining the result depends on processing another message on this actor).

For this purpose, there is the method `Future.as[T]` which waits until either the future is completed or its timeout expires, whichever comes first. The result is then inspected and returned as `Some[T]` if it was normally completed and the answer's runtime type matches the desired type; if the future contains an exception or the value cannot be cast to the desired type, it will throw the exception or a `ClassCastException` (if you want to get `None` in the latter case, use `Future.asSilently[T]`). In case of a timeout, `None` is returned.

```
(actor ? msg).as[String] match {
  case Some(answer) => ...
  case None         => ...
}

val resultOption = (actor ? msg).as[String]
if (resultOption.isDefined) ... else ...

for (x <- (actor ? msg).as[Int]) yield { 2 * x }
```

Forward message

You can forward a message from one actor to another. This means that the original sender address/reference is maintained even though the message is going through a ‘mediator’. This can be useful when writing actors that work as routers, load-balancers, replicators etc.

```
actor.forward(message)
```

4.1.6 Receive messages

An Actor has to implement the `receive` method to receive messages:

```
protected def receive: PartialFunction[Any, Unit]
```

Note: Akka has an alias to the `PartialFunction[Any, Unit]` type called `Receive` (`akka.actor.Actor.Receive`), so you can use this type instead for clarity. But most often you don't need to spell it out.

This method should return a `PartialFunction`, e.g. a ‘match/case’ clause in which the message can be matched against the different case clauses using Scala pattern matching. Here is an example:

```
class MyActor extends Actor {
  def receive = {
    case "Hello" =>
      log.info("Received 'Hello'")

    case _ =>
      throw new RuntimeException("unknown message")
  }
}
```

4.1.7 Reply to messages

Reply using the channel

If you want to have a handle to an object to whom you can reply to the message, you can use the `Channel` abstraction. Simply call `self.channel` and then you can forward that to others, store it away or otherwise until you want to reply, which you do by `channel ! response`:

```
case request =>
  val result = process(request)
  self.channel ! result // will throw an exception if there is no sender information
  self.channel tryTell result // will return Boolean whether reply succeeded
```

The `Channel` trait is contravariant in the expected message type. Since `self.channel` is subtype of `Channel[Any]`, you may specialise your return channel to allow the compiler to check your replies:

```
class MyActor extends Actor {
  def doIt(channel: Channel[String], x: Any) = { channel ! x.toString }
  def receive = {
    case x => doIt(self.channel, x)
  }
}
```

```
case request =>
  friend forward self.channel
```

We recommend that you as first choice use the channel abstraction instead of the other ways described in the following sections.

Reply using the reply and reply_? methods

If you want to send a message back to the original sender of the message you just received then you can use the `reply(..)` method.

```
case request =>
  val result = process(request)
  self.reply(result)
```

In this case the `result` will be send back to the Actor that sent the `request`.

The `reply` method throws an `IllegalStateException` if unable to determine what to reply to, e.g. the sender is not an actor. You can also use the more forgiving `tryReply` method which returns `true` if reply was sent, and `false` if unable to determine what to reply to.

```
case request =>
  val result = process(request)
  if (self.tryReply(result)) ...// success
  else ... // handle failure
```

Summary of reply semantics and options

- `self.reply(...)` can be used to reply to an Actor or a Future from within an actor; the current actor will be passed as reply channel if the current channel supports this.
- `self.channel` is a reference providing an abstraction for the reply channel; this reference may be passed to other actors or used by non-actor code.

Note: There used to be two methods for determining the sending Actor or Future for the current invocation:

- `self.sender` yielded a `Option[ActorRef]`
- `self.senderFuture` yielded a `Option[CompletableFuture[Any]]`

These two concepts have been unified into the `channel`. If you need to know the nature of the channel, you may do so using pattern matching:

```
channel match {
  case ref : ActorRef => ...
  case f : ActorCompletableFuture => ...
}
```

4.1.8 Initial receive timeout

A timeout mechanism can be used to receive a message when no initial message is received within a certain time. To receive this timeout you have to set the `receiveTimeout` property and declare a case handling the `ReceiveTimeout` object.

```
self.receiveTimeout = Some(30000L) // 30 seconds

def receive = {
  case "Hello" =>
    log.info("Received 'Hello'")
  case ReceiveTimeout =>
    throw new RuntimeException("received timeout")
}
```

This mechanism also work for hotswapped receive functions. Every time a `HotSwap` is sent, the receive timeout is reset and rescheduled.

4.1.9 Starting actors

Actors are started by invoking the `start` method.

```
val actor = actorOf[MyActor]
actor.start()
```

You can create and start the `Actor` in a one liner like this:

```
val actor = actorOf[MyActor].start()
```

When you start the `Actor` then it will automatically call the `def preStart` callback method on the `Actor` trait. This is an excellent place to add initialization code for the actor.

```
override def preStart() = {
  ... // initialization code
}
```

4.1.10 Stopping actors

Actors are stopped by invoking the `stop` method.

```
actor.stop()
```

When `stop` is called then a call to the `def postStop` callback method will take place. The `Actor` can use this callback to implement shutdown behavior.

```
override def postStop() = {
  ... // clean up resources
}
```

You can shut down all `Actors` in the system by invoking:

```
Actor.registry.shutdownAll()
```

4.1.11 PoisonPill

You can also send an actor the `akka.actor.PoisonPill` message, which will stop the actor when the message is processed.

If the sender is a `Future` (e.g. the message is sent with `?`), the `Future` will be completed with an `akka.actor.ActorKilledException("PoisonPill")`.

4.1.12 HotSwap

Upgrade

Akka supports hotswapping the Actor's message loop (e.g. its implementation) at runtime. There are two ways you can do that:

- Send a `HotSwap` message to the Actor.
- Invoke the `become` method from within the Actor.

Both of these takes a `ActorRef => PartialFunction[Any, Unit]` that implements the new message handler. The hotswapped code is kept in a `Stack` which can be pushed and popped.

To hotswap the Actor body using the `HotSwap` message:

```
actor ! HotSwap( self => {
  case message => self.reply("hotswapped body")
})
```

Using the `HotSwap` message for hotswapping has its limitations. You can not replace it with any code that uses the Actor's `self` reference. If you need to do that the `become` method is better.

To hotswap the Actor using `become`:

```
def angry: Receive = {
  case "foo" => self reply "I am already angry!!!"
  case "bar" => become(happy)
}

def happy: Receive = {
  case "bar" => self reply "I am already happy :-)"
  case "foo" => become(angry)
}

def receive = {
  case "foo" => become(angry)
  case "bar" => become(happy)
}
```

The `become` method is useful for many different things, but a particular nice example of it is in example where it is used to implement a Finite State Machine (FSM): [Dining Hackers](#)

Here is another little cute example of `become` and `unbecome` in action:

```
case object Swap
class Swapper extends Actor {
  def receive = {
    case Swap =>
      println("Hi")
      become {
        case Swap =>
```

```

        println("Ho")
        unbecome() // resets the latest 'become' (just for fun)
    }
}
}

val swap = actorOf[Swapper].start()

swap ! Swap // prints Hi
swap ! Swap // prints Ho
swap ! Swap // prints Hi
swap ! Swap // prints Ho
swap ! Swap // prints Hi
swap ! Swap // prints Ho

```

4.1.13 Encoding Scala Actors nested receives without accidentally leaking memory: UnnestedReceive

Downgrade

Since the hotswapped code is pushed to a Stack you can downgrade the code as well. There are two ways you can do that:

- Send the Actor a `RevertHotSwap` message
- Invoke the `unbecome` method from within the Actor.

Both of these will pop the Stack and replace the Actor's implementation with the `PartialFunction[Any, Unit]` that is at the top of the Stack.

Revert the Actor body using the `RevertHotSwap` message:

```
actor ! RevertHotSwap
```

Revert the Actor body using the `unbecome` method:

```
def receive: Receive = {
  case "revert" => unbecome()
}
```

4.1.14 Killing an Actor

You can kill an actor by sending a `Kill` message. This will restart the actor through regular supervisor semantics.

Use it like this:

```
// kill the actor called 'victim'
victim ! Kill
```

4.1.15 Actor life-cycle

The actor has a well-defined non-circular life-cycle.

```

NEW (newly created actor) - can't receive messages (yet)
  => STARTED (when 'start' is invoked) - can receive messages
    => SHUT DOWN (when 'exit' or 'stop' is invoked) - can't do anything

```

4.1.16 Extending Actors using PartialFunction chaining

A bit advanced but very useful way of defining a base message handler and then extend that, either through inheritance or delegation, is to use `PartialFunction.orElse` chaining.

In generic base Actor:

```
import akka.actor.Actor.Receive

abstract class GenericActor extends Actor {
  // to be defined in subclassing actor
  def specificMessageHandler: Receive

  // generic message handler
  def genericMessageHandler: Receive = {
    case event => printf("generic: %s\n", event)
  }

  def receive = specificMessageHandler orElse genericMessageHandler
}
```

In subclassing Actor:

```
class SpecificActor extends GenericActor {
  def specificMessageHandler = {
    case event: MyMsg => printf("specific: %s\n", event.subject)
  }
}

case class MyMsg(subject: String)
```

4.2 Typed Actors (Scala)

Contents

- Creating Typed Actors
 - Creating Typed Actors with non-default constructor
 - Configuration factory class
- Sending messages
 - One-way message send
 - Request-reply message send
 - Request-reply-with-future message send
- Stopping Typed Actors
- How to use the TypedActorContext for runtime information access
- Messages and immutability

Module stability: **SOLID**

The Typed Actors are implemented through [Typed Actors](#). It uses AOP through [AspectWerkz](#) to turn regular POJOs into asynchronous non-blocking Actors with semantics of the Actor Model. Each method dispatch is turned into a message that is put on a queue to be processed by the Typed Actor sequentially one by one.

If you are using the [Spring Framework](#) then take a look at Akka's Spring integration.

4.2.1 Creating Typed Actors

IMPORTANT: The Typed Actors class must have access modifier ‘public’ (which is default) and can’t be an inner class (unless it is an inner class in an ‘object’).

Akka turns POJOs with interface and implementation into asynchronous (Typed) Actors. Akka is using [AspectWerkz’s Proxy](#) implementation, which is the [most performant](#) proxy implementation there exists.

In order to create a Typed Actor you have to subclass the TypedActor base class.

Here is an example.

If you have a POJO with an interface implementation separation like this:

```
import akka.actor.TypedActor

trait RegistrationService {
  def register(user: User, cred: Credentials): Unit
  def getUserFor(username: String): User
}

public class RegistrationServiceImpl extends TypedActor with RegistrationService {
  def register(user: User, cred: Credentials): Unit = {
    ... // register user
  }

  def getUserFor(username: String): User = {
    ... // fetch user by username
    user
  }
}
```

Then you can create an Typed Actor out of it by creating it through the ‘TypedActor’ factory like this:

```
val service = TypedActor.newInstance(classOf[RegistrationService], classOf[RegistrationServiceImpl])
// The last parameter defines the timeout for Future calls
```

Creating Typed Actors with non-default constructor

To create a typed actor that takes constructor arguments use a variant of ‘newInstance’ or ‘newRemoteInstance’ that takes a call-by-name block in which you can create the Typed Actor in any way you like.

Here is an example:

```
val service = TypedActor.newInstance(classOf[Service], new ServiceWithConstructorArgs("someString"))
```

Configuration factory class

Using a configuration object:

```
import akka.actor.TypedActorConfiguration
import akka.util.Duration
import akka.util.duration._

val config = TypedActorConfiguration()
  .timeout(3000 millis)

val service = TypedActor.newInstance(classOf[RegistrationService], classOf[RegistrationServiceImpl])
```

However, often you will not use these factory methods but declaratively define the Typed Actors as part of a supervisor hierarchy. More on that in the [Fault Tolerance Through Supervisor Hierarchies \(Scala\)](#) section.

4.2.2 Sending messages

Messages are sent simply by invoking methods on the POJO, which is proxy to the “real” POJO now. The arguments to the method are bundled up atomically into a message and sent to the receiver (the actual POJO instance).

One-way message send

Methods that return void are turned into ‘fire-and-forget’ semantics by asynchronously firing off the message and return immediately. In the example above it would be the ‘register’ method, so if this method is invoked then it returns immediately:

```
// method invocation returns immediately and method is invoke asynchronously using the Actor Model
service.register(user, creds)
```

Request-reply message send

Methods that return something (e.g. non-void methods) are turned into ‘send-and-receive-eventually’ semantics by asynchronously firing off the message and wait on the reply using a Future.

```
// method invocation is asynchronously dispatched using the Actor Model semantics,
// but it blocks waiting on a Future to be resolved in the background
val user = service.getUser(username)
```

Generally it is preferred to use fire-forget messages as much as possible since they will never block, e.g. consume a resource by waiting. But sometimes they are neat to use since they: # Simulates standard Java method dispatch, which is more intuitive for most Java developers # Are a neat to model request-reply # Are useful when you need to do things in a defined order

Request-reply-with-future message send

Methods that return a ‘akka.dispatch.Future<TYPE>’ are turned into ‘send-and-receive-with-future’ semantics by asynchronously firing off the message and returns immediately with a Future. You need to use the ‘future(...)’ method in the TypedActor base class to resolve the Future that the client code is waiting on.

Here is an example:

```
class MathTypedActorImpl extends TypedActor with MathTypedActor {
  def square(x: Int): Future[Integer] = future(x * x)
}

// create the ping actor
val math = TypedActor.newInstance(classOf[MathTyped], classOf[MathTypedImpl])

// This method will return immediately when called, caller should wait on the Future for the result
val future = math.square(10)
future.await
val result: Int = future.get
```

4.2.3 Stopping Typed Actors

Once Typed Actors have been created with one of the TypedActor.newInstance methods they need to be stopped with TypedActor.stop to free resources allocated by the created Typed Actor (this is not needed when the Typed Actor is supervised).

```
// Create Typed Actor
val service = TypedActor.newInstance(classOf[RegistrationService], classOf[RegistrationServiceImp

// ...

// Free Typed Actor resources
TypedActor.stop(service)
```

When the Typed Actor defines a shutdown callback method (*Fault Tolerance Through Supervisor Hierarchies (Scala)*) it will be invoked on `TypedActor.stop`.

4.2.4 How to use the TypedActorContext for runtime information access

The `'akka.actor.TypedActorContext'` class Holds 'runtime type information' (RTTI) for the Typed Actor. This context is a member field in the TypedActor base class and holds for example the current sender reference, the current sender future etc.

Here is an example how you can use it to in a 'void' (e.g. fire-forget) method to implement request-reply by using the sender reference:

```
class PingImpl extends TypedActor with Ping {

  def hit(count: Int) {
    val pong = context.getSender.asInstanceOf[Pong]
    pong.hit(count++)
  }
}
```

If the sender, sender future etc. is not available, then these methods will return 'null' so you should have a way of dealing with that scenario.

4.2.5 Messages and immutability

IMPORTANT: Messages can be any kind of object but have to be immutable (there is a workaround, see next section). Java or Scala can't enforce immutability (yet) so this has to be by convention. Primitives like String, int, Long are always immutable. Apart from these you have to create your own immutable objects to send as messages. If you pass on a reference to an instance that is mutable then this instance can be modified concurrently by two different Typed Actors and the Actor model is broken leaving you with NO guarantees and most likely corrupt data.

Akka can help you in this regard. It allows you to turn on an option for serializing all messages, e.g. all parameters to the Typed Actor effectively making a deep clone/copy of the parameters. This will make sending mutable messages completely safe. This option is turned on in the `'$AKKA_HOME/config/akka.conf'` config file like this:

```
akka {
  actor {
    serialize-messages = on # does a deep clone of messages to ensure immutability
  }
}
```

This will make a deep clone (using Java serialization) of all parameters.

4.3 ActorRegistry (Scala)

Module stability: **SOLID**

4.3.1 ActorRegistry: Finding Actors

Actors can be looked up by using the `akka.actor.Actor.registry: akka.actor.ActorRegistry`. Lookups for actors through this registry can be done by:

- `uuid akka.actor.Uuid` – this uses the `uuid` field in the `Actor` class, returns the actor reference for the actor with specified `uuid`, if one exists, otherwise `None`
- `id string` – this uses the `id` field in the `Actor` class, which can be set by the user (default is the class name), returns all actor references to actors with specified `id`
- `specific actor class` - returns an `Array[Actor]` with all actors of this exact class
- `parameterized type` - returns an `Array[Actor]` with all actors that are a subtype of this specific type

Actors are automatically registered in the `ActorRegistry` when they are started, removed or stopped. You can explicitly register and unregister `ActorRef`'s by using the `register` and `unregister` methods. The `ActorRegistry` contains many convenience methods for looking up typed actors.

Here is a summary of the API for finding actors:

```
def actors: Array[ActorRef]
def actorFor(uuid: akka.actor.Uuid): Option[ActorRef]
def actorsFor(id: String): Array[ActorRef]
def actorsFor[T <: Actor](implicit manifest: Manifest[T]): Array[ActorRef]
def actorsFor[T <: Actor](clazz: Class[T]): Array[ActorRef]

// finding typed actors
def typedActors: Array[AnyRef]
def typedActorFor(uuid: akka.actor.Uuid): Option[AnyRef]
def typedActorsFor(id: String): Array[AnyRef]
def typedActorsFor[T <: AnyRef](implicit manifest: Manifest[T]): Array[AnyRef]
def typedActorsFor[T <: AnyRef](clazz: Class[T]): Array[AnyRef]
```

Examples of how to use them:

```
val actor = Actor.registry.actorFor(uuid)
val pojo = Actor.registry.typedActorFor(uuid)
```

```
val actors = Actor.registry.actorsFor(classOf[...])
val pojoes = Actor.registry.typedActorsFor(classOf[...])
```

```
val actors = Actor.registry.actorsFor(id)
val pojoes = Actor.registry.typedActorsFor(id)
```

```
val actors = Actor.registry.actorsFor[MyActorType]
val pojoes = Actor.registry.typedActorsFor[MyTypedActorImpl]
```

The `ActorRegistry` also has a 'shutdownAll' and 'foreach' methods:

```
def foreach(f: (ActorRef) => Unit)
def foreachTypedActor(f: (AnyRef) => Unit)
def shutdownAll()
```

If you need to know when a new `Actor` is added or removed from the registry, you can use the subscription API. You can register an `Actor` that should be notified when an event happens in the `ActorRegistry`:

```
def addListener(listener: ActorRef)
def removeListener(listener: ActorRef)
```

The messages sent to this `Actor` are:

```
case class ActorRegistered(actor: ActorRef)
case class ActorUnregistered(actor: ActorRef)
```

So your listener `Actor` needs to be able to handle these two messages. Example:


```
import akka.actor.Actor
import akka.actor.ActorRegistered;
import akka.actor.ActorUnregistered;
import akka.actor.UntypedActor;
import akka.event.EventHandler;

class RegistryListener extends Actor {
  def receive = {
    case event: ActorRegistered =>
      EventHandler.info(this, "Actor registered: %s - %s".format(
        event.actor.actorClassName, event.actor.uuid))
    case event: ActorUnregistered =>
      // ...
  }
}
```

The above actor can be added as listener of registry events:

```
import akka.actor._
import akka.actor.Actor._

val listener = actorOf[RegistryListener].start()
registry.addListener(listener)
```

4.4 Futures (Scala)

Contents

- Introduction
- Use with Actors
- Use Directly
- onComplete
- Functional Futures
 - Future is a Monad
 - For Comprehensions
 - Composing Futures
 - Scalaz
- Exceptions
- Timeouts

4.4.1 Introduction

In Akka, a [Future](#) is a data structure used to retrieve the result of some concurrent operation. This operation is usually performed by an [Actor](#) or by the [Dispatcher](#) directly. This result can be accessed synchronously (blocking) or asynchronously (non-blocking).

4.4.2 Use with Actors

There are generally two ways of getting a reply from an [Actor](#): the first is by a sent message (`actor ! msg`), which only works if the original sender was an [Actor](#)) and the second is through a [Future](#).

Using an [Actor](#)'s `? method to send a message will return a Future. To wait for and retrieve the actual result the simplest method is:`

```
val future = actor ? msg
val result: Any = future.get()
```

This will cause the current thread to block and wait for the `Actor` to ‘complete’ the `Future` with its reply. Due to the dynamic nature of Akka’s `Actors` this result will be untyped and will default to `Nothing`. The safest way to deal with this is to cast the result to an `Any` as is shown in the above example. You can also use the expected result type instead of `Any`, but if an unexpected type were to be returned you will get a `ClassCastException`. For more elegant ways to deal with this and to use the result without blocking, refer to [Functional Futures](#).

4.4.3 Use Directly

A common use case within Akka is to have some computation performed concurrently without needing the extra utility of an `Actor`. If you find yourself creating a pool of `Actors` for the sole reason of performing a calculation in parallel, there is an easier (and faster) way:

```
import akka.dispatch.Future

val future = Future {
  "Hello" + "World"
}
val result = future.get()
```

In the above code the block passed to `Future` will be executed by the default `Dispatcher`, with the return value of the block used to complete the `Future` (in this case, the result would be the string: “HelloWorld”). Unlike a `Future` that is returned from an `Actor`, this `Future` is properly typed, and we also avoid the overhead of managing an `Actor`.

4.4.4 onComplete

The mother of “`Future`“-composition is the `onComplete` callback, which allows you to get notified asynchronously when the “`Future`“ gets completed:

```
val future: Future[Any] = ...
future onComplete {
  _.value.get match {
    case Left(problem) => handleCompletedWithException(problem)
    case Right(result) => handleCompletedWithResult(result)
  }
}

//You can also short that down to:
future onComplete { _.value.get.fold(handleCompletedWithException(_), handleCompletedWithResult(_)) }

//There’s also a callback named ‘onResult’ that only deals with results (and the other, ‘onException’)
future onResult {
  case "foo" => logResult("GOT MYSELF A FOO OH YEAH BABY!")
}
```

4.4.5 Functional Futures

A recent addition to Akka’s `Future` is several monadic methods that are very similar to the ones used by Scala’s collections. These allow you to create ‘pipelines’ or ‘streams’ that the result will travel through.

Future is a Monad

The first method for working with `Future` functionally is `map`. This method takes a `Function` which performs some operation on the result of the `Future`, and returning a new result. The return value of the `map` method is another `Future` that will contain the new result:

```
val f1 = Future {
  "Hello" + "World"
}

val f2 = f1 map { x =>
  x.length
}

val result = f2.get
```

In this example we are joining two strings together within a `Future`. Instead of waiting for this to complete, we apply our function that calculates the length of the string using the `map` method. Now we have a second `Future` that will eventually contain an `Int`. When our original `Future` completes, it will also apply our function and complete the second `Future` with it's result. When we finally `get` the result, it will contain the number 10. Our original `Future` still contains the string "HelloWorld" and is unaffected by the `map`.

The `map` method is fine if we are modifying a single `Future`, but if 2 or more `Futures` are involved `map` will not allow you to combine them together:

```
val f1 = Future {
  "Hello" + "World"
}

val f2 = Future {
  3
}

val f3 = f1 map { x =>
  f2 map { y =>
    x.length * y
  }
}

val result = f2.get.get
```

The `get` method had to be used twice because `f3` is a `Future[Future[Int]]` instead of the desired `Future[Int]`. Instead, the `flatMap` method should be used:

```
val f1 = Future {
  "Hello" + "World"
}

val f2 = Future {
  3
}

val f3 = f1 flatMap { x =>
  f2 map { y =>
    x.length * y
  }
}

val result = f2.get
```

For Comprehensions

Since `Future` has a `map` and `flatMap` method it can be easily used in a ‘for comprehension’:

```
val f = for {
  a <- Future(10 / 2) // 10 / 2 = 5
  b <- Future(a + 1)  // 5 + 1 = 6
  c <- Future(a - 1)  // 5 - 1 = 4
} yield b * c          // 6 * 4 = 24

val result = f.get
```

Something to keep in mind when doing this is even though it looks like parts of the above example can run in parallel, each step of the for comprehension is run sequentially. This will happen on separate threads for each step but there isn’t much benefit over running the calculations all within a single `Future`. The real benefit comes when the `Futures` are created first, and then combining them together.

Composing Futures

The example for comprehension above is an example of composing `Futures`. A common use case for this is combining the replies of several `Actors` into a single calculation without resorting to calling `get` or `await` to block for each result. First an example of using `get`:

```
val f1 = actor1 ? msg1
val f2 = actor2 ? msg2

val a: Int = f1.get
val b: Int = f2.get

val f3 = actor3 ? (a + b)

val result: String = f3.get
```

Here we wait for the results from the first 2 `Actors` before sending that result to the third `Actor`. We called `get` 3 times, which caused our little program to block 3 times before getting our final result. Now compare that to this example:

```
val f1 = actor1 ? msg1
val f2 = actor2 ? msg2

val f3 = for {
  a: Int    <- f1
  b: Int    <- f2
  c: String <- actor3 ? (a + b)
} yield c

val result = f3.get
```

Here we have 2 actors processing a single message each. Once the 2 results are available (note that we don’t block to get these results!), they are being added together and sent to a third `Actor`, which replies with a string, which we assign to ‘result’.

This is fine when dealing with a known amount of `Actors`, but can grow unwieldy if we have more than a handful. The `sequence` and `traverse` helper methods can make it easier to handle more complex use cases. Both of these methods are ways of turning, for a subclass `T` of `Traversable`, `T[Future[A]]` into a `Future[T[A]]`. For example:

```
// oddActor returns odd numbers sequentially from 1
val listOfFutures: List[Future[Int]] = List.fill(100)(oddActor ? GetNext)

// now we have a Future[List[Int]]
val futureList = Future.sequence(listOfFutures)
```

```
// Find the sum of the odd numbers
val oddSum = futureList.map(_._sum).get
```

To better explain what happened in the example, `Future.sequence` is taking the `List[Future[Int]]` and turning it into a `Future[List[Int]]`. We can then use `map` to work with the `List[Int]` directly, and we find the sum of the `List`.

The `traverse` method is similar to `sequence`, but it takes a `T[A]` and a function `T => Future[B]` to return a `Future[T[B]]`, where `T` is again a subclass of `Traversable`. For example, to use `traverse` to sum the first 100 odd numbers:

```
val oddSum = Future.traverse((1 to 100).toList)(x => Future(x * 2 - 1)).map(_._sum).get
```

This is the same result as this example:

```
val oddSum = Future.sequence((1 to 100).toList.map(x => Future(x * 2 - 1))).map(_._sum).get
```

But it may be faster to use `traverse` as it doesn't have to create an intermediate `List[Future[Int]]`.

Then there's a method that's called `fold` that takes a start-value, a sequence of `Futures` and a function from the type of the start-value and the type of the futures and returns something with the same type as the start-value, and then applies the function to all elements in the sequence of futures, non-blockingly, the execution will run on the Thread of the last completing `Future` in the sequence.

```
val futures = for(i <- 1 to 1000) yield Future(i * 2) // Create a sequence of Futures
val futureSum = Futures.fold(0)(futures)(_ + _)
```

That's all it takes!

If the sequence passed to `fold` is empty, it will return the start-value, in the case above, that will be 0. In some cases you don't have a start-value and you're able to use the value of the first completing `Future` in the sequence as the start-value, you can use `reduce`, it works like this:

```
val futures = for(i <- 1 to 1000) yield Future(i * 2) // Create a sequence of Futures
val futureSum = Futures.reduce(futures)(_ + _)
```

Same as with `fold`, the execution will be done by the Thread that completes the last of the `Futures`, you can also parallelize it by chunking your futures into sub-sequences and reduce them, and then reduce the reduced results again.

This is just a sample of what can be done, but to use more advanced techniques it is easier to take advantage of `Scalaz`, which Akka has support for in its `akka-scalaz` module.

Scalaz

Akka also has a `Scalaz` module (*Add-on Modules*) for a more complete support of programming in a functional style.

4.4.6 Exceptions

Since the result of a `Future` is created concurrently to the rest of the program, exceptions must be handled differently. It doesn't matter if an `Actor` or the dispatcher is completing the `Future`, if an `Exception` is caught the `Future` will contain it instead of a valid result. If a `Future` does contain an `Exception`, calling `get` will cause it to be thrown again so it can be handled properly.

It is also possible to handle an `Exception` by returning a different result. This is done with the `recover` method. For example:

```
val future = actor ? msg1 recover {
  case e: ArithmeticException => 0
}
```

In this example, if an `ArithmeticException` was thrown while the `Actor` processed the message, our `Future` would have a result of 0. The `failure` method works very similarly to the standard `try/catch` blocks, so multiple `Exceptions` can be handled in this manner, and if an `Exception` is not handled this way it will behave as if we hadn't used the `failure` method.

You also have the option to register a callback that will be executed if the `Future` is completed with an exception:

```
val f: Future[Any] = ...
f onFailure {
  case npe: NullPointerException => doSomething
  case e: Exception => doSomethingElse
  case SomeRegex(param) => doSomethingOther
  case _ => doAnything
} // Applies the specified partial function to the result of the future when it is completed with an exception
```

4.4.7 Timeouts

Waiting forever for a `Future` to be completed can be dangerous. It could cause your program to block indefinitely or produce a memory leak. `Future` has support for a timeout already builtin with a default of 5000 milliseconds (taken from 'akka.conf'). A timeout can also be specified when creating a `Future`:

```
val future = Future( { doSomething }, 10000 )
```

This example creates a `Future` with a 10 second timeout.

If the timeout is reached the `Future` becomes unusable, even if an attempt is made to complete it. It is possible to have a `Future` perform an action on timeout if needed with the `onTimeout` method:

```
val future1 = actor ? msg onTimeout { _ =>
  println("Timed out!")
}
```

4.5 Dataflow Concurrency (Scala)

Contents

- Description
- Getting Started
- Dataflow Variables
- Dataflow Delimiter
- Examples
 - Simple `DataFlowVariable` example
 - Example of using `DataFlowVariable` with recursion
 - Example using concurrent `Futures`

4.5.1 Description

Akka implements [Oz-style dataflow concurrency](#) by using a special API for *Futures (Scala)* that allows single assignment variables and multiple lightweight (event-based) processes/threads.

Dataflow concurrency is deterministic. This means that it will always behave the same. If you run it once and it yields output 5 then it will do that **every time**, run it 10 million times, same result. If it on the other hand deadlocks the first time you run it, then it will deadlock **every single time** you run it. Also, there is **no difference** between sequential code and concurrent code. These properties makes it very easy to reason about concurrency. The limitation is that the code needs to be side-effect free, e.g. deterministic. You can't use exceptions, time, random etc., but need to treat the part of your program that uses dataflow concurrency as a pure function with input and output.

The best way to learn how to program with dataflow variables is to read the fantastic book [Concepts, Techniques, and Models of Computer Programming](#). By Peter Van Roy and Seif Haridi.

The documentation is not as complete as it should be, something we will improve shortly. For now, besides above listed resources on dataflow concurrency, I recommend you to read the documentation for the GPar implementation, which is heavily influenced by the Akka implementation:

- <http://gpars.codehaus.org/Dataflow>
- <http://www.gpars.org/guide/guide/7.%20Dataflow%20Concurrency.html>

4.5.2 Getting Started

Scala's Delimited Continuations plugin is required to use the Dataflow API. To enable the plugin when using sbt, your project must inherit the `AutoCompilerPlugins` trait and contain a bit of configuration as is seen in this example:

```
import sbt._

class MyAkkaProject(info: ProjectInfo) extends DefaultProject(info) with AkkaProject with AutoCompilerPlugins {
  val continuationsPlugin = compilerPlugin("org.scala-lang.plugins" % "continuations" % "2.9.1")
  override def compileOptions = super.compileOptions ++ compileOptions("-P:continuations:enable")
}
```

4.5.3 Dataflow Variables

Dataflow Variable defines four different operations:

1. Define a Dataflow Variable

```
val x = Promise[Int]()
```

2. Wait for Dataflow Variable to be bound (must be contained within a `Future.flow` block as described in the next section)

```
x()
```

3. Bind Dataflow Variable (must be contained within a `Future.flow` block as described in the next section)

```
x << 3
```

4. Bind Dataflow Variable with a Future (must be contained within a `Future.flow` block as described in the next section)

```
x << y
```

A Dataflow Variable can only be bound once. Subsequent attempts to bind the variable will be ignored.

4.5.4 Dataflow Delimiter

Dataflow is implemented in Akka using Scala's Delimited Continuations. To use the Dataflow API the code must be contained within a `Future.flow` block. For example:

```
import Future.flow

val a = Future( ... )
val b = Future( ... )
val c = Promise[Int]()

flow {
  c << (a() + b())
}

val result = c.get()
```

The `flow` method also returns a `Future` for the result of the contained expression, so the previous example could also be written like this:

```
import Future.flow

val a = Future( ... )
val b = Future( ... )

val c = flow {
  a() + b()
}

val result = c.get()
```

4.5.5 Examples

Most of these examples are taken from the [Oz wikipedia page](#)

To run these examples:

1. Start REPL

```
$ sbt
> project akka-actor
> console
```

```
Welcome to Scala version 2.9.1 (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_25).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

2. Paste the examples (below) into the Scala REPL. Note: Do not try to run the Oz version, it is only there for reference.

3. Have fun.

Simple DataFlowVariable example

This example is from Oz wikipedia page: [http://en.wikipedia.org/wiki/Oz_\(programming_language\)](http://en.wikipedia.org/wiki/Oz_(programming_language)). Sort of the “Hello World” of dataflow concurrency.

Example in Oz:

```
thread
  Z = X+Y      % will wait until both X and Y are bound to a value.
  {Browse Z}   % shows the value of Z.
end
thread X = 40 end
thread Y = 2  end
```


Example in Akka:

```
import akka.dispatch._
import Future.flow

val x, y, z = Promise[Int]()

flow {
  z << x() + y()
  println("z = " + z())
}
flow { x << 40 }
flow { y << 2 }
```

Example of using DataFlowVariable with recursion

Using DataFlowVariable and recursion to calculate sum.

Example in Oz:

```
fun {Ints N Max}
  if N == Max then nil
  else
    {Delay 1000}
    N|{Ints N+1 Max}
  end
end

fun {Sum S Stream}
  case Stream of nil then S
  [] H|T then S|{Sum H+S T} end
end

local X Y in
  thread X = {Ints 0 1000} end
  thread Y = {Sum 0 X} end
  {Browse Y}
end
```

Example in Akka:

```
import akka.dispatch._
import Future.flow

def ints(n: Int, max: Int): List[Int] = {
  if (n == max) Nil
  else n :: ints(n + 1, max)
}

def sum(s: Int, stream: List[Int]): List[Int] = stream match {
  case Nil => s :: Nil
  case h :: t => s :: sum(h + s, t)
}

val x, y = Promise[List[Int]]()

flow { x << ints(0, 1000) }
flow { y << sum(0, x()) }
flow { println("List of sums: " + y()) }
```

Example using concurrent Futures

Shows how to have a calculation run in another thread.

Example in Akka:

```
import akka.dispatch._
import Future.flow

// create four 'Int' data flow variables
val x, y, z, v = Promise[Int]()

flow {
  println("Thread 'main'")

  x << 1
  println("'x' set to: " + x())

  println("Waiting for 'y' to be set...")

  if (x() > y()) {
    z << x
    println("'z' set to 'x': " + z())
  } else {
    z << y
    println("'z' set to 'y': " + z())
  }
}

flow {
  y << Future {
    println("Thread 'setY', sleeping")
    Thread.sleep(2000)
    2
  }
  println("'y' set to: " + y())
}

flow {
  println("Thread 'setV'")
  v << y
  println("'v' set to 'y': " + v())
}
```

4.6 Agents (Scala)

Contents

- Creating and stopping Agents
- Updating Agents
- Reading an Agent's value
- Awaiting an Agent's value
- Transactional Agents
- Monadic usage

Module stability: **SOLID**

Agents in Akka were inspired by agents in Clojure.

Agents provide asynchronous change of individual locations. Agents are bound to a single storage location for their lifetime, and only allow mutation of that location (to a new state) to occur as a result of an action. Update actions are functions that are asynchronously applied to the Agent's state and whose return value becomes the Agent's new state. The state of an Agent should be immutable.

While updates to Agents are asynchronous, the state of an Agent is always immediately available for reading by any thread (using `get` or `apply`) without any messages.

Agents are reactive. The update actions of all Agents get interleaved amongst threads in a thread pool. At any point in time, at most one `send` action for each Agent is being executed. Actions dispatched to an agent from another thread will occur in the order they were sent, potentially interleaved with actions dispatched to the same agent from other sources.

If an Agent is used within an enclosing transaction, then it will participate in that transaction. Agents are integrated with the STM - any dispatches made in a transaction are held until that transaction commits, and are discarded if it is retried or aborted.

4.6.1 Creating and stopping Agents

Agents are created by invoking `Agent(value)` passing in the Agent's initial value.

```
val agent = Agent(5)
```

An Agent will be running until you invoke `close` on it. Then it will be eligible for garbage collection (unless you hold on to it in some way).

```
agent.close()
```

4.6.2 Updating Agents

You update an Agent by sending a function that transforms the current value or by sending just a new value. The Agent will apply the new value or function atomically and asynchronously. The update is done in a fire-forget manner and you are only guaranteed that it will be applied. There is no guarantee of when the update will be applied but dispatches to an Agent from a single thread will occur in order. You apply a value or a function by invoking the `send` function.

```
// send a value
agent send 7

// send a function
agent send (_ + 1)
agent send (_ * 2)
```

You can also dispatch a function to update the internal state but on its own thread. This does not use the reactive thread pool and can be used for long-running or blocking operations. You do this with the `sendOff` method. Dispatches using either `sendOff` or `send` will still be executed in order.

```
// sendOff a function
agent sendOff (longRunningOrBlockingFunction)
```

4.6.3 Reading an Agent's value

Agents can be dereferenced, e.g. you can get an Agent's value, by invoking the Agent with parenthesis like this:

```
val result = agent()
```

Or by using the `get` method.

```
val result = agent.get
```

Reading an Agent's current value does not involve any message passing and happens immediately. So while updates to an Agent are asynchronous, reading the state of an Agent is synchronous.

4.6.4 Awaiting an Agent's value

It is also possible to read the value after all currently queued sends have completed. You can do this with `await`:

```
val result = agent.await
```

You can also get a `Future` to this value, that will be completed after the currently queued updates have completed:

```
val future = agent.future
// ...
val result = future.await.result.get
```

4.6.5 Transactional Agents

If an Agent is used within an enclosing transaction, then it will participate in that transaction. If you send to an Agent within a transaction then the dispatch to the Agent will be held until that transaction commits, and discarded if the transaction is aborted.

```
import akka.agent.Agent
import akka.stm._

def transfer(from: Agent[Int], to: Agent[Int], amount: Int): Boolean = {
  atomic {
    if (from.get < amount) false
    else {
      from send (_ - amount)
      to send (_ + amount)
      true
    }
  }
}

val from = Agent(100)
val to = Agent(20)
val ok = transfer(from, to, 50)

from() // -> 50
to()   // -> 70
```

4.6.6 Monadic usage

Agents are also monadic, allowing you to compose operations using for-comprehensions. In a monadic usage, new Agents are created leaving the original Agents untouched. So the old values (Agents) are still available as-is. They are so-called 'persistent'.

Example of a monadic usage:

```
val agent1 = Agent(3)
val agent2 = Agent(5)

// uses foreach
var result = 0
for (value <- agent1) {
  result = value + 1
}

// uses map
```

```

val agent3 =
  for (value <- agent1) yield value + 1

// uses flatMap
val agent4 = for {
  value1 <- agent1
  value2 <- agent2
} yield value1 + value2

agent1.close()
agent2.close()
agent3.close()
agent4.close()

```

4.7 Software Transactional Memory (Scala)

Contents

- Overview of STM
- Simple example
- Ref
 - Creating a Ref
 - Accessing the value of a Ref
 - Changing the value of a Ref
 - Refs in for-comprehensions
- Transactions
 - Retries
 - Unexpected retries
 - Coordinated transactions and Transactors
 - Configuring transactions
 - Transaction lifecycle listeners
 - Blocking transactions
 - Alternative blocking transactions
- Transactional datastructures
- Persistent datastructures
- Ants simulation sample

Module stability: **SOLID**

4.7.1 Overview of STM

An **STM** turns the Java heap into a transactional data set with begin/commit/rollback semantics. Very much like a regular database. It implements the first three letters in ACID; ACI: * Atomic * Consistent * Isolated

Generally, the STM is not needed very often when working with Akka. Some use-cases (that we can think of) are:

- When you really need composable message flows across many actors updating their **internal local** state but need them to do that atomically in one big transaction. Might not be often, but when you do need this then you are screwed without it.
- When you want to share a datastructure across actors.
- When you need to use the persistence modules.

Akka's STM implements the concept in [Clojure's STM](#) view on state in general. Please take the time to read [this excellent document](#) and view [this presentation](#) by Rich Hickey (the genius behind Clojure), since it forms the basis of Akka's view on STM and state in general.

The STM is based on Transactional References (referred to as Refs). Refs are memory cells, holding an (arbitrary) immutable value, that implement CAS (Compare-And-Swap) semantics and are managed and enforced by the STM for coordinated changes across many Refs. They are implemented using the excellent [Multiverse STM](#).

Working with immutable collections can sometimes give bad performance due to extensive copying. Scala provides so-called persistent datastructures which makes working with immutable collections fast. They are immutable but with constant time access and modification. They use structural sharing and an insert or update does not ruin the old structure, hence "persistent". Makes working with immutable composite types fast. The persistent datastructures currently consist of a Map and Vector.

4.7.2 Simple example

Here is a simple example of an incremental counter using STM. This shows creating a Ref, a transactional reference, and then modifying it within a transaction, which is delimited by `atomic`.

```
import akka.stm._

val ref = Ref(0)

def counter = atomic {
  ref alter (_ + 1)
}

counter
// -> 1

counter
// -> 2
```

4.7.3 Ref

Refs (transactional references) are mutable references to values and through the STM allow the safe sharing of mutable data. Refs separate identity from value. To ensure safety the value stored in a Ref should be immutable (they can of course contain refs themselves). The value referenced by a Ref can only be accessed or swapped within a transaction. If a transaction is not available, the call will be executed in its own transaction (the call will be atomic). This is a different approach than the Clojure Refs, where a missing transaction results in an error.

Creating a Ref

You can create a Ref with or without an initial value.

```
import akka.stm._

// giving an initial value
val ref = Ref(0)

// specifying a type but no initial value
val ref = Ref[Int]
```

Accessing the value of a Ref

Use `get` to access the value of a Ref. Note that if no initial value has been given then the value is initially `null`.

```
import akka.stm._

val ref = Ref(0)

atomic {
  ref.get
}

// -> 0
```

If there is a chance that the value of a `Ref` is null then you can use `opt`, which will create an `Option`, either `Some(value)` or `None`, or you can provide a default value with `getOrElse`. You can also check for null using `isNull`.

```
import akka.stm._

val ref = Ref[Int]

atomic {
  ref.opt           // -> None
  ref.getOrElse(0) // -> 0
  ref.isNull       // -> true
}
```

Changing the value of a Ref

To set a new value for a `Ref` you can use `set` (or equivalently `swap`), which sets the new value and returns the old value.

```
import akka.stm._

val ref = Ref(0)

atomic {
  ref.set(5)
}

// -> 0

atomic {
  ref.get
}

// -> 5
```

You can also use `alter` which accepts a function that takes the old value and creates a new value of the same type.

```
import akka.stm._

val ref = Ref(0)

atomic {
  ref.alter(_ + 5)
}

// -> 5

val inc = (i: Int) => i + 1

atomic {
  ref.alter(inc)
}

// -> 6
```

Refs in for-comprehensions

Ref is monadic and can be used in for-comprehensions.

```
import akka.stm._

val ref = Ref(1)

atomic {
  for (value <- ref) {
    // do something with value
  }
}

val anotherRef = Ref(3)

atomic {
  for {
    value1 <- ref
    value2 <- anotherRef
  } yield (value1 + value2)
}
// -> Ref(4)

val emptyRef = Ref[Int]

atomic {
  for {
    value1 <- ref
    value2 <- emptyRef
  } yield (value1 + value2)
}
// -> Ref[Int]
```

4.7.4 Transactions

A transaction is delimited using `atomic`.

```
atomic {
  // ...
}
```

All changes made to transactional objects are isolated from other changes, all make it or non make it (so failure atomicity) and are consistent. With the AkkaSTM you automatically have the Oracle version of the **SERIALIZED** isolation level, lower isolation is not possible. To make it fully serialized, set the `writeskew` property that checks if a writeskew problem is allowed to happen.

Retries

A transaction is automatically retried when it runs into some read or write conflict, until the operation completes, an exception (throwable) is thrown or when there are too many retries. When a read or write conflict is encountered, the transaction uses a bounded exponential backoff to prevent cause more contention and give other transactions some room to complete.

If you are using non transactional resources in an atomic block, there could be problems because a transaction can be retried. If you are using print statements or logging, it could be that they are called more than once. So you need to be prepared to deal with this. One of the possible solutions is to work with a deferred or compensating task that is executed after the transaction aborts or commits.

Unexpected retries

It can happen for the first few executions that you get a few failures of execution that lead to unexpected retries, even though there is not any read or write conflict. The cause of this is that speculative transaction configuration/selection is used. There are transactions optimized for a single transactional object, for 1..n and for n to unlimited. So based on the execution of the transaction, the system learns; it begins with a cheap one and upgrades to more expensive ones. Once it has learned, it will reuse this knowledge. It can be activated/deactivated using the speculative property on the TransactionFactory. In most cases it is best use the default value (enabled) so you get more out of performance.

Coordinated transactions and Transactors

If you need coordinated transactions across actors or threads then see *Transactors (Scala)*.

Configuring transactions

It's possible to configure transactions. The `atomic` method can take an implicit or explicit `TransactionFactory`, which can determine properties of the transaction. A default transaction factory is used if none is specified explicitly or there is no implicit `TransactionFactory` in scope.

Configuring transactions with an **implicit** `TransactionFactory`:

```
import akka.stm._

implicit val txFactory = TransactionFactory(readonly = true)

atomic {
  // read only transaction
}
```

Configuring transactions with an **explicit** `TransactionFactory`:

```
import akka.stm._

val txFactory = TransactionFactory(readonly = true)

atomic(txFactory) {
  // read only transaction
}
```

The following settings are possible on a `TransactionFactory`:

- `familyName` - Family name for transactions. Useful for debugging.
- `readonly` - Sets transaction as readonly. Readonly transactions are cheaper.
- `maxRetries` - The maximum number of times a transaction will retry.
- `timeout` - The maximum time a transaction will block for.
- `trackReads` - Whether all reads should be tracked. Needed for blocking operations.
- `writeSkew` - Whether writeskew is allowed. Disable with care.
- `blockingAllowed` - Whether explicit retries are allowed.
- `interruptible` - Whether a blocking transaction can be interrupted.
- `speculative` - Whether speculative configuration should be enabled.
- `quickRelease` - Whether locks should be released as quickly as possible (before whole commit).
- `propagation` - For controlling how nested transactions behave.
- `traceLevel` - Transaction trace level.

You can also specify the default values for some of these options in akka.conf. Here they are with their default values:

```
stm {
  fair                = on          # Should global transactions be fair or non-fair (non fair yield better performance)
  max-retries         = 1000
  timeout             = 5           # Default timeout for blocking transactions and transaction set (in units of the time-unit property)

  write-skew          = true
  blocking-allowed    = false
  interruptible       = false
  speculative         = true
  quick-release       = true
  propagation         = "requires"
  trace-level         = "none"
}
```

You can also determine at which level a transaction factory is shared or not shared, which affects the way in which the STM can optimise transactions.

Here is a shared transaction factory for all instances of an actor.

```
import akka.actor._
import akka.stm._

object MyActor {
  implicit val txFactory = TransactionFactory(readonly = true)
}

class MyActor extends Actor {
  import MyActor.txFactory

  def receive = {
    case message: String =>
      atomic {
        // read only transaction
      }
  }
}
```

Here's a similar example with an individual transaction factory for each instance of an actor.

```
import akka.actor._
import akka.stm._

class MyActor extends Actor {
  implicit val txFactory = TransactionFactory(readonly = true)

  def receive = {
    case message: String =>
      atomic {
        // read only transaction
      }
  }
}
```

Transaction lifecycle listeners

It's possible to have code that will only run on the successful commit of a transaction, or when a transaction aborts. You can do this by adding deferred or compensating blocks to a transaction.

```
import akka.stm._
```

```

atomic {
  deferred {
    // executes when transaction commits
  }
  compensating {
    // executes when transaction aborts
  }
}

```

Blocking transactions

You can block in a transaction until a condition is met by using an explicit `retry`. To use `retry` you also need to configure the transaction to allow explicit retries.

Here is an example of using `retry` to block until an account has enough money for a withdrawal. This is also an example of using actors and STM together.

```

import akka.stm._
import akka.actor._
import akka.util.duration._
import akka.event.EventHandler

type Account = Ref[Double]

case class Transfer(from: Account, to: Account, amount: Double)

class Transferer extends Actor {
  implicit val txFactory = TransactionFactory(blockingAllowed = true, trackReads = true, timeout = 10.seconds)

  def receive = {
    case Transfer(from, to, amount) =>
      atomic {
        if (from.get < amount) {
          EventHandler.info(this, "not enough money - retrying")
          retry
        }
        EventHandler.info(this, "transferring")
        from alter (_ - amount)
        to alter (_ + amount)
      }
  }
}

val account1 = Ref(100.0)
val account2 = Ref(100.0)

val transferer = Actor.actorOf(new Transferer).start()

transferer ! Transfer(account1, account2, 500.0)
// INFO Transferer: not enough money - retrying

atomic { account1 alter (_ + 2000) }
// INFO Transferer: transferring

atomic { account1.get }
// -> 1600.0

atomic { account2.get }
// -> 600.0

transferer.stop()

```

Alternative blocking transactions

You can also have two alternative blocking transactions, one of which can succeed first, with `either-orElse`.

```
import akka.stm._
import akka.actor._
import akka.util.duration._
import akka.event.EventHandlerler

case class Branch(left: Ref[Int], right: Ref[Int], amount: Int)

class Brancher extends Actor {
  implicit val txFactory = TransactionFactory(blockingAllowed = true, trackReads = true, timeout = 10.seconds)

  def receive = {
    case Branch(left, right, amount) =>
      atomic {
        either {
          if (left.get < amount) {
            EventHandlerler.info(this, "not enough on left - retrying")
            retry
          }
          log.info("going left")
        } orElse {
          if (right.get < amount) {
            EventHandlerler.info(this, "not enough on right - retrying")
            retry
          }
          log.info("going right")
        }
      }
  }
}

val ref1 = Ref(0)
val ref2 = Ref(0)

val brancher = Actor.actorOf(new Brancher).start()

brancher ! Branch(ref1, ref2, 1)
// INFO Brancher: not enough on left - retrying
// INFO Brancher: not enough on right - retrying

atomic { ref2 alter (_ + 1) }
// INFO Brancher: not enough on left - retrying
// INFO Brancher: going right

brancher.stop()
```

4.7.5 Transactional datastructures

Akka provides two datastructures that are managed by the STM.

- TransactionalMap
- TransactionalVector

TransactionalMap and TransactionalVector look like regular mutable datastructures, they even implement the standard Scala 'Map' and 'RandomAccessSeq' interfaces, but they are implemented using persistent datastructures and managed references under the hood. Therefore they are safe to use in a concurrent environment. Underlying TransactionalMap is HashMap, an immutable Map but with near constant time access and modification operations.

Similarly `TransactionalVector` uses a persistent `Vector`. See the `Persistent Datastructures` section below for more details.

Like managed references, `TransactionalMap` and `TransactionalVector` can only be modified inside the scope of an STM transaction.

IMPORTANT: There have been some problems reported when using transactional datastructures with ‘lazy’ initialization. Avoid that.

Here is how you create these transactional datastructures:

```
import akka.stm._

// assuming something like
case class User(name: String)
case class Address(location: String)

// using initial values
val map = TransactionalMap("bill" -> User("bill"))
val vector = TransactionalVector(Address("somewhere"))

// specifying types
val map = TransactionalMap[String, User]
val vector = TransactionalVector[Address]
```

`TransactionalMap` and `TransactionalVector` wrap persistent datastructures with transactional references and provide a standard Scala interface. This makes them convenient to use.

Here is an example of using a `Ref` and a `HashMap` directly:

```
import akka.stm._
import scala.collection.immutable.HashMap

case class User(name: String)

val ref = Ref(HashMap[String, User]())

atomic {
  val users = ref.get
  val newUsers = users + ("bill" -> User("bill")) // creates a new HashMap
  ref.swap(newUsers)
}

atomic {
  ref.get.apply("bill")
}
// -> User("bill")
```

Here is the same example using `TransactionalMap`:

```
import akka.stm._

case class User(name: String)

val users = TransactionalMap[String, User]

atomic {
  users += "bill" -> User("bill")
}

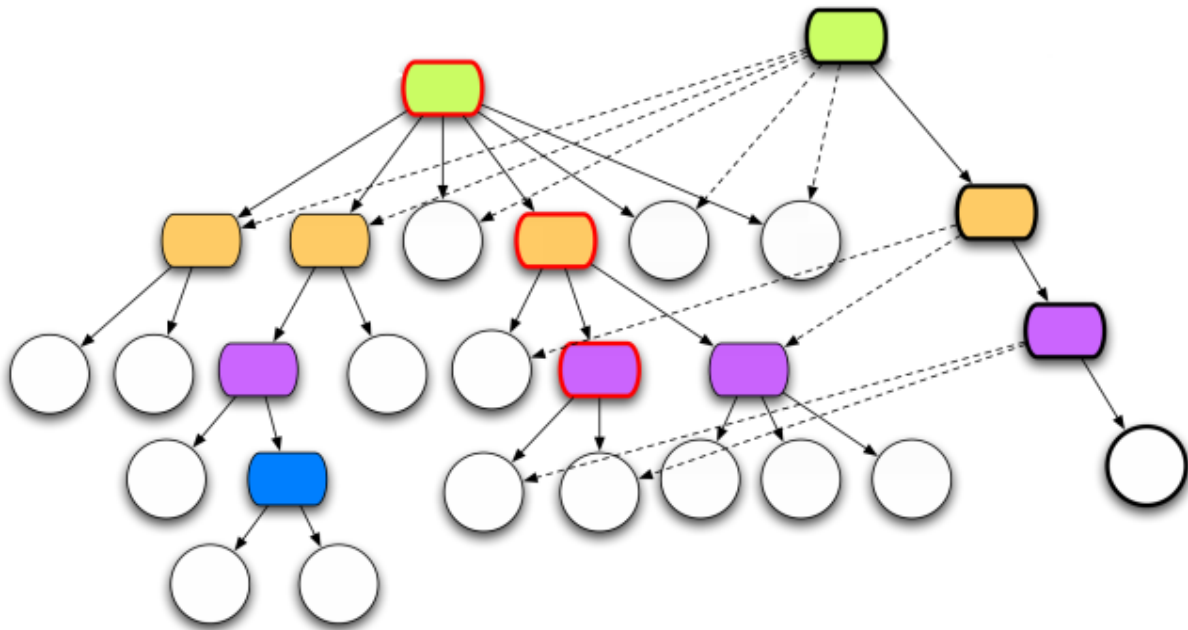
atomic {
  users("bill")
}
// -> User("bill")
```

4.7.6 Persistent datastructures

Akka's STM should only be used with immutable data. This can be costly if you have large datastructures and are using a naive copy-on-write. In order to make working with immutable datastructures fast enough Scala provides what are called Persistent Datastructures. There are currently two different ones: * [HashMap \(scaladoc\)](#) * [Vector \(scaladoc\)](#)

They are immutable and each update creates a completely new version but they are using clever structural sharing in order to make them almost as fast, for both read and update, as regular mutable datastructures.

This illustration is taken from Rich Hickey's presentation. Copyright Rich Hickey 2009.



4.7.7 Ants simulation sample

One fun and very enlightening visual demo of STM, actors and transactional references is the [Ant simulation sample](#). I encourage you to run it and read through the code since it's a good example of using actors with STM.

4.8 Transactors (Scala)

Contents

- [Why Transactors?](#)
 - [Actors and STM](#)
- [Coordinated transactions](#)
- [Transactor](#)
- [Coordinating Typed Actors](#)

Module stability: **SOLID**

4.8.1 Why Transactors?

Actors are excellent for solving problems where you have many independent processes that can work in isolation and only interact with other Actors through message passing. This model fits many problems. But the actor model

is unfortunately a terrible model for implementing truly shared state. E.g. when you need to have consensus and a stable view of state across many components. The classic example is the bank account where clients can deposit and withdraw, in which each operation needs to be atomic. For detailed discussion on the topic see [this JavaOne presentation](#).

STM on the other hand is excellent for problems where you need consensus and a stable view of the state by providing compositional transactional shared state. Some of the really nice traits of STM are that transactions compose, and it raises the abstraction level from lock-based concurrency.

Akka's Transactors combine Actors and STM to provide the best of the Actor model (concurrency and asynchronous event-based programming) and STM (compositional transactional shared state) by providing transactional, compositional, asynchronous, event-based message flows.

If you need Durability then you should not use one of the in-memory data structures but one of the persistent ones.

Generally, the STM is not needed very often when working with Akka. Some use-cases (that we can think of) are:

- When you really need composable message flows across many actors updating their **internal local** state but need them to do that atomically in one big transaction. Might not often, but when you do need this then you are screwed without it.
- When you want to share a datastructure across actors.
- When you need to use the persistence modules.

Actors and STM

You can combine Actors and STM in several ways. An Actor may use STM internally so that particular changes are guaranteed to be atomic. Actors may also share transactional datastructures as the STM provides safe shared state across threads.

It's also possible to coordinate transactions across Actors or threads so that either the transactions in a set all commit successfully or they all fail. This is the focus of Transactors and the explicit support for coordinated transactions in this section.

4.8.2 Coordinated transactions

Akka provides an explicit mechanism for coordinating transactions across Actors. Under the hood it uses a `CountDownCommitBarrier`, similar to a `CountDownLatch`.

Here is an example of coordinating two simple counter Actors so that they both increment together in coordinated transactions. If one of them was to fail to increment, the other would also fail.

```
import akka.transactor.Coordinated
import akka.stm.Ref
import akka.actor.{Actor, ActorRef}

case class Increment(friend: Option[ActorRef] = None)
case object GetCount

class Counter extends Actor {
  val count = Ref(0)

  def receive = {
    case coordinated @ Coordinated(Increment(friend)) => {
      friend foreach (_ ! coordinated(Increment()))
      coordinated atomic {
        count alter (_ + 1)
      }
    }
    case GetCount => self.reply(count.get)
  }
}
```

```
val counter1 = Actor.actorOf[Counter].start()
val counter2 = Actor.actorOf[Counter].start()

counter1 ! Coordinated(Increment(Some(counter2)))

...

(counter1 ? GetCount).as[Int] // Some(1)

counter1.stop()
counter2.stop()
```

To start a new coordinated transaction that you will also participate in, just create a `Coordinated` object:

```
val coordinated = Coordinated()
```

To start a coordinated transaction that you won't participate in yourself you can create a `Coordinated` object with a message and send it directly to an actor. The recipient of the message will be the first member of the coordination set:

```
actor ! Coordinated(Message)
```

To receive a coordinated message in an actor simply match it in a case statement:

```
def receive = {
  case coordinated @ Coordinated(Message) => ...
}
```

To include another actor in the same coordinated transaction that you've created or received, use the `apply` method on that object. This will increment the number of parties involved by one and create a new `Coordinated` object to be sent.

```
actor ! coordinated(Message)
```

To enter the coordinated transaction use the `atomic` method of the coordinated object:

```
coordinated atomic {
  // do something in transaction ...
}
```

The coordinated transaction will wait for the other transactions before committing. If any of the coordinated transactions fail then they all fail.

4.8.3 Transactor

Transactors are actors that provide a general pattern for coordinating transactions, using the explicit coordination described above.

Here's an example of a simple transactor that will join a coordinated transaction:

```
import akka.transactor.Transactor
import akka.stm.Ref

case object Increment

class Counter extends Transactor {
  val count = Ref(0)

  override def atomically = {
    case Increment => count alter (_ + 1)
  }
}
```


You could send this Counter transactor a `Coordinated(Increment)` message. If you were to send it just an `Increment` message it will create its own `Coordinated` (but in this particular case wouldn't be coordinating transactions with any other transactors).

To coordinate with other transactors override the `coordinate` method. The `coordinate` method maps a message to a set of `SendTo` objects, pairs of `ActorRef` and a message. You can use the `include` and `sendTo` methods to easily coordinate with other transactors. The `include` method will send on the same message that was received to other transactors. The `sendTo` method allows you to specify both the actor to send to, and the message to send.

Example of coordinating an increment:

```
import akka.transactor.Transactor
import akka.stm.Ref
import akka.actor.ActorRef

case object Increment

class FriendlyCounter(friend: ActorRef) extends Transactor {
  val count = Ref(0)

  override def coordinate = {
    case Increment => include(friend)
  }

  override def atomically = {
    case Increment => count alter (_ + 1)
  }
}
```

Using `include` to include more than one transactor:

```
override def coordinate = {
  case Message => include(actor1, actor2, actor3)
}
```

Using `sendTo` to coordinate transactions but pass-on a different message than the one that was received:

```
override def coordinate = {
  case Message => sendTo(someActor -> SomeOtherMessage)
  case SomeMessage => sendTo(actor1 -> Message1, actor2 -> Message2)
}
```

To execute directly before or after the coordinated transaction, override the `before` and `after` methods. These methods also expect partial functions like the `receive` method. They do not execute within the transaction.

To completely bypass coordinated transactions override the `normally` method. Any message matched by `normally` will not be matched by the other methods, and will not be involved in coordinated transactions. In this method you can implement normal actor behavior, or use the normal STM atomic for local transactions.

4.8.4 Coordinating Typed Actors

It's also possible to use coordinated transactions with typed actors. You can explicitly pass around `Coordinated` objects, or use built-in support with the `@Coordinated` annotation and the `Coordination.coordinate` method.

To specify a method should use coordinated transactions add the `@Coordinated` annotation. **Note:** the `@Coordinated` annotation only works with methods that return `Unit` (one-way methods).

```
trait Counter {
  @Coordinated def increment()
  def get: Int
}
```

To coordinate transactions use a `coordinate` block:

```
coordinate {  
  counter1.increment()  
  counter2.increment()  
}
```

Here's an example of using `@Coordinated` with a `TypedActor` to coordinate increments.

```
import akka.actor.TypedActor  
import akka.stm.Ref  
import akka.transactor.annotation.Coordinated  
import akka.transactor.Coordination._  
  
trait Counter {  
  @Coordinated def increment()  
  def get: Int  
}  
  
class CounterImpl extends TypedActor with Counter {  
  val ref = Ref(0)  
  def increment() { ref alter (_ + 1) }  
  def get = ref.get  
}  
  
...  
  
val counter1 = TypedActor.newInstance(classOf[Counter], classOf[CounterImpl])  
val counter2 = TypedActor.newInstance(classOf[Counter], classOf[CounterImpl])  
  
coordinate {  
  counter1.increment()  
  counter2.increment()  
}  
  
TypedActor.stop(counter1)  
TypedActor.stop(counter2)
```

The `coordinate` block will wait for the transactions to complete. If you do not want to wait then you can specify this explicitly:

```
coordinate(wait = false) {  
  counter1.increment()  
  counter2.increment()  
}
```

4.9 Remote Actors (Scala)

Contents

- Starting up the remote service
 - Starting remote service in user code as a library
 - Starting remote service as part of the stand-alone Kernel
 - Stopping a RemoteNode or RemoteServer
 - Connecting and shutting down a client explicitly
 - Remote Client message frame size configuration
 - Remote Client reconnect configuration
 - Remote Client message buffering and send retry on failure
- Running Remote Server in untrusted mode
- Using secure cookie for remote client authentication
 - Enabling secure cookie authentication
 - Generating and using the secure cookie
- Client-managed Remote Actors
- Server-managed Remote Actors
 - Server side setup
 - Session bound server side setup
 - Client side usage
 - Running sample
- Automatic remote 'sender' reference management
- Identifying remote actors
- Client-managed Remote Typed Actors
- Server-managed Remote Typed Actors
 - Server side setup
 - Session bound server side setup
 - Client side usage
- Data Compression Configuration
- Code provisioning
- Subscribe to Remote Client events
- Subscribe to Remote Server events
- Message Serialization
 - Scala JSON
 - Protobuf

Module stability: **SOLID**

Akka supports starting and interacting with Actors and Typed Actors on remote nodes using a very efficient and scalable NIO implementation built upon [JBoss Netty](#) and [Google Protocol Buffers](#) .

The usage is completely transparent with local actors, both in regards to sending messages and error handling and propagation as well as supervision, linking and restarts. You can send references to other Actors as part of the message.

You can find a runnable sample [here](#).

4.9.1 Starting up the remote service

Starting remote service in user code as a library

Here is how to start up the RemoteNode and specify the hostname and port programmatically:

```
import akka.actor.Actor._

remote.start("localhost", 2552)
```

```
// Specify the classloader to use to load the remote class (actor)
remote.start("localhost", 2552, classLoader)
```

Here is how to start up the RemoteNode and specify the hostname and port in the ‘akka.conf’ configuration file (see the section below for details):

```
import akka.actor.Actor._

remote.start()

// Specify the classloader to use to load the remote class (actor)
remote.start(classLoader)
```

Starting remote service as part of the stand-alone Kernel

You simply need to make sure that the service is turned on in the external ‘akka.conf’ configuration file.

```
akka {
  remote {
    server {
      service = on
      hostname = "localhost"
      port = 2552
      connection-timeout = 1000 # in millis
    }
  }
}
```

Stopping a RemoteNode or RemoteServer

If you invoke ‘shutdown’ on the server then the connection will be closed.

```
import akka.actor.Actor._

remote.shutdown()
```

Connecting and shutting down a client explicitly

Normally you should not have to start and stop the client connection explicitly since that is handled by Akka on a demand basis. But if you for some reason want to do that then you can do it like this:

```
import akka.actor.Actor._
import java.net.InetSocketAddress

remote.shutdownClientConnection(new InetSocketAddress("localhost", 6666)) //Returns true if success
remote.restartClientConnection(new InetSocketAddress("localhost", 6666)) //Returns true if success
```

Remote Client message frame size configuration

You can define the max message frame size for the remote messages:

```
akka {
  remote {
    client {
      message-frame-size = 1048576
    }
  }
}
```

Remote Client reconnect configuration

The Remote Client automatically performs reconnection upon connection failure.

You can configure it like this:

```
akka {
  remote {
    client {
      reconnect-delay = 5           # in seconds (5 sec default)
      read-timeout = 10            # in seconds (10 sec default)
      reconnection-time-window = 600 # the maximum time window that a client should try to reconnect
    }
  }
}
```

The RemoteClient is automatically trying to reconnect to the server if the connection is broken. By default it has a reconnection window of 10 minutes (600 seconds).

If it has not been able to reconnect during this period of time then it is shut down and further attempts to use it will yield a 'RemoteClientException'. The 'RemoteClientException' contains the message as well as a reference to the RemoteClient that is not yet connect in order for you to retrieve it and do an explicit connect if needed.

You can also register a listener that will listen for example the 'RemoteClientStopped' event, retrieve the 'RemoteClient' from it and reconnect explicitly.

See the section on RemoteClient listener and events below for details.

Remote Client message buffering and send retry on failure

The Remote Client implements message buffering on network failure. This feature has zero overhead (even turned on) in the successful scenario and a queue append operation in case of unsuccessful send. So it is really really fast.

The default behavior is that the remote client will maintain a transaction log of all messages that it has failed to send due to network problems (not other problems like serialization errors etc.). The client will try to resend these messages upon first successful reconnect and the message ordering is maintained. This means that the remote client will swallow all exceptions due to network failure and instead queue remote messages in the transaction log. The failures will however be reported through the remote client life-cycle events as well as the regular Akka event handler. You can turn this behavior on and off in the configuration file. It gives 'at-least-once' semantics, use a message id/counter for discarding potential duplicates (or use idempotent messages).

```
akka {
  remote {
    client {
      buffering {
        retry-message-send-on-failure = on
        capacity = -1                # If negative (or zero) then an unbounded mailbox is used
                                    # If positive then a bounded mailbox is used and the capacity is the limit
      }
    }
  }
}
```

If you choose a capacity higher than 0, then a bounded queue will be used and if the limit of the queue is reached then a 'RemoteClientMessageBufferException' will be thrown.

4.9.2 Running Remote Server in untrusted mode

You can run the remote server in untrusted mode. This means that the server will not allow any client-managed remote actors or any life-cycle messages and methods. This is useful if you want to let untrusted clients use server-managed actors in a safe way. This can optionally be combined with the secure cookie authentication mechanism described below as well as the SSL support for remote actor communication.

If the client is trying to perform one of these unsafe actions then a `java.lang.SecurityException` is thrown on the server as well as transferred to the client and thrown there as well.

Here is how you turn it on:

```
akka {
  remote {
    server {
      untrusted-mode = on # the default is 'off'
    }
  }
}
```

The messages that it prevents are all that extends `LifeCycleMessage`: `* class HotSwap(..)` `* class RevertHotSwap(..)` `* class Restart(..)` `* class Exit(..)` `* class Link(..)` `* class Unlink(..)` `* class UnlinkAndStop(..)` `* class ReceiveTimeout(..)`

It also prevents the client from invoking any life-cycle and side-effecting methods, such as: `* start` `* stop` `* link` `* unlink` `* spawnLink` `* etc.`

4.9.3 Using secure cookie for remote client authentication

Akka is using a similar scheme for remote client node authentication as Erlang; using secure cookies. In order to use this authentication mechanism you have to do two things:

- Enable secure cookie authentication in the remote server
- Use the same secure cookie on all the trusted peer nodes

Enabling secure cookie authentication

The first one is done by enabling the secure cookie authentication in the remote server section in the configuration file:

```
akka {
  remote {
    server {
      require-cookie = on
    }
  }
}
```

Now if you have try to connect to a server with a client then it will first try to authenticate the client by comparing the secure cookie for the two nodes. If they are the same then it allows the client to connect and use the server freely but if they are not the same then it will throw a `java.lang.SecurityException` and not allow the client to connect.

Generating and using the secure cookie

The secure cookie can be any string value but in order to ensure that it is secure it is best to randomly generate it. This can be done by invoking the `generate_config_with_secure_cookie.sh` script which resides in the `$AKKA_HOME/scripts` folder. This script will generate and print out a complete `akka.conf` configuration file with the generated secure cookie defined that you can either use as-is or cut and paste the `secure-cookie` snippet. Here is an example of its generated output:

```
# This config imports the Akka reference configuration.
include "akka-reference.conf"

# In this file you can override any option defined in the 'akka-reference.conf' file.
# Copy in all or parts of the 'akka-reference.conf' file and modify as you please.
```

```
akka {
  remote {
    secure-cookie = "000E02050F0300040C050C0D060A040306090B0C"
  }
}
```

The simplest way to use it is to have it create your ‘akka.conf’ file like this:

```
cd $AKKA_HOME
./scripts/generate_config_with_secure_cookie.sh > ./config/akka.conf
```

Now it is good to make sure that the configuration file is only accessible by the owner of the file. On Unix-style file system this can be done like this:

```
chmod 400 ./config/akka.conf
```

Running this script requires having ‘scala’ on the path (and will take a couple of seconds to run since it is using Scala and has to boot up the JVM to run).

You can also generate the secure cookie by using the ‘Crypt’ object and its ‘generateSecureCookie’ method.

```
import akka.util.Crypt

val secureCookie = Crypt.generateSecureCookie
```

The secure cookie is a cryptographically secure randomly generated byte array turned into a SHA-1 hash.

4.9.4 Client-managed Remote Actors

DEPRECATED AS OF 1.1

The client creates the remote actor and “moves it” to the server.

When you define an actor as being remote it is instantiated as on the remote host and your local actor becomes a proxy, it works as a handle to the remote actor. The real execution is always happening on the remote node.

Actors can be made remote by calling `remote.actorOf[MyActor](host, port)`

Here is an example:

```
import akka.actor.Actor

class MyActor extends Actor {
  def receive = {
    case "hello" => self.reply("world")
  }
}

val remoteActor = Actor.remote.actorOf[MyActor]("192.68.23.769", 2552)
```

An Actor can also start remote child Actors through one of the ‘spawn/link’ methods. These will start, link and make the Actor remote atomically.

```
...
self.spawnRemote[MyActor](hostname, port, timeout)
self.spawnLinkRemote[MyActor](hostname, port, timeout)
...
```

4.9.5 Server-managed Remote Actors

Here it is the server that creates the remote actor and the client can ask for a handle to this actor.

Server side setup

The API for server managed remote actors is really simple. 2 methods only:

```
class HelloWorldActor extends Actor {
  def receive = {
    case "Hello" => self.reply("World")
  }
}

remote.start("localhost", 2552) //Start the server
remote.register("hello-service", actorOf[HelloWorldActor]) //Register the actor with the specified
```

Actors created like this are automatically started.

You can also register an actor by its UUID rather than ID or handle. This is done by prefixing the handle with the “uuid:” protocol.

```
remote.register("uuid:" + actor.uuid, actor)

remote.unregister("uuid:" + actor.uuid)
```

Session bound server side setup

Session bound server managed remote actors work by creating and starting a new actor for every client that connects. Actors are stopped automatically when the client disconnects. The client side is the same as regular server managed remote actors. Use the function `registerPerSession` instead of `register`.

Session bound actors are useful if you need to keep state per session, e.g. username. They are also useful if you need to perform some cleanup when a client disconnects by overriding the `postStop` method as described here

```
class HelloWorldActor extends Actor {
  def receive = {
    case "Hello" => self.reply("World")
  }
}

remote.start("localhost", 2552)
remote.registerPerSession("hello-service", actorOf[HelloWorldActor])
```

Note that the second argument in `registerPerSession` is an implicit function. It will be called to create an actor every time a session is established.

Client side usage

```
val actor = remote.actorFor("hello-service", "localhost", 2552)
val result = (actor ? "Hello").as[String]
```

There are many variations on the ‘`remote#actorFor`’ method. Here are some of them:

```
... = remote.actorFor(className, hostname, port)
... = remote.actorFor(className, timeout, hostname, port)
... = remote.actorFor(uuid, className, hostname, port)
... = remote.actorFor(uuid, className, timeout, hostname, port)
... // etc
```

All of these also have variations where you can pass in an explicit ‘`ClassLoader`’ which can be used when deserializing messages sent from the remote actor.

Running sample

Here is a complete running sample (also available [here](#)):

Paste in the code below into two sbt console shells. Then run:

- `ServerInitiatedRemoteActorServer.run()` in one shell
- `ServerInitiatedRemoteActorClient.run()` in the other shell

```
import akka.actor.Actor
import Actor._
import akka.event.EventHandlerler

class HelloWorldActor extends Actor {
  def receive = {
    case "Hello" => self.reply("World")
  }
}

object ServerInitiatedRemoteActorServer {

  def run() {
    remote.start("localhost", 2552)
    remote.register("hello-service", actorOf[HelloWorldActor])
  }

  def main(args: Array[String]) { run() }
}

object ServerInitiatedRemoteActorClient {

  def run() {
    val actor = remote.actorFor("hello-service", "localhost", 2552)
    val result = (actor ? "Hello").as[AnyRef]
    EventHandlerler.info("Result from Remote Actor: %s", result)
  }

  def main(args: Array[String]) { run() }
}
```

4.9.6 Automatic remote ‘sender’ reference management

The sender of a remote message will be reachable with a reply through the remote server on the node that the actor is residing, automatically. Please note that firewalled clients won’t work right now. [2011-01-05]

4.9.7 Identifying remote actors

The ‘id’ field in the ‘Actor’ class is of importance since it is used as identifier for the remote actor. If you want to create a brand new actor every time you instantiate a remote actor then you have to set the ‘id’ field to a unique ‘String’ for each instance. If you want to reuse the same remote actor instance for each new remote actor (of the same class) you create then you don’t have to do anything since the ‘id’ field by default is equal to the name of the actor class.

Here is an example of overriding the ‘id’ field:

```
import akka.actor.newUuid

class MyActor extends Actor {
  self.id = newUuid.toString
  def receive = {
    case "hello" => self.reply("world")
  }
}

val actor = remote.actorOf[MyActor] ("192.68.23.769", 2552)
```

4.9.8 Client-managed Remote Typed Actors

DEPRECATED AS OF 1.1

You can define the Typed Actor to be a remote service by adding the ‘RemoteAddress’ configuration element in the declarative supervisor configuration:

```
new Component (
  Foo.class,
  new Lifecycle(new Permanent(), 1000),
  1000,
  new RemoteAddress("localhost", 2552))
```

You can also define an Typed Actor to be remote programmatically when creating it explicitly:

```
TypedActorFactory factory = new TypedActorFactory();

POJO pojo = (POJO) factory.newRemoteInstance(POJO.class, 1000, "localhost", 2552)

... // use pojo as usual
```

4.9.9 Server-managed Remote Typed Actors

WARNING: Remote TypedActors do not work with overloaded methods on your TypedActor, refrain from using overloading.

Server side setup

The API for server managed remote typed actors is nearly the same as for untyped actor

```
class RegistrationServiceImpl extends TypedActor with RegistrationService {
  def registerUser(user: User): Unit = {
    ... // register user
  }
}

remote.start("localhost", 2552)

val typedActor = TypedActor.newInstance(classOf[RegistrationService], classOf[RegistrationService])
remote.registerTypedActor("user-service", typedActor)
```

Actors created like this are automatically started.

Session bound server side setup

Session bound server managed remote actors work by creating and starting a new actor for every client that connects. Actors are stopped automatically when the client disconnects. The client side is the same as regular server managed remote actors. Use the function registerTypedPerSessionActor instead of registerTypedActor.

Session bound actors are useful if you need to keep state per session, e.g. username. They are also useful if you need to perform some cleanup when a client disconnects.

```
class RegistrationServiceImpl extends TypedActor with RegistrationService {
  def registerUser(user: User): Unit = {
    ... // register user
  }
}
```

```
remote.start("localhost", 2552)

remote.registerTypedPerSessionActor("user-service",
  TypedActor.newInstance(classOf[RegistrationService],
    classOf[RegistrationServiceImpl], 2000))
```

Note that the second argument in `registerTypedPerSessionActor` is an implicit function. It will be called to create an actor every time a session is established.

Client side usage

```
val actor = remote.typedActorFor(classOf[RegistrationService], "user-service", 5000L, "localhost")
actor.registerUser(...)
```

There are variations on the ‘`remote#typedActorFor`’ method. Here are some of them:

```
... = remote.typedActorFor(interfaceClazz, serviceIdOrClassName, hostname, port)
... = remote.typedActorFor(interfaceClazz, serviceIdOrClassName, timeout, hostname, port)
... = remote.typedActorFor(interfaceClazz, serviceIdOrClassName, timeout, hostname, port, classLoader)
```

4.9.10 Data Compression Configuration

Akka uses compression to minimize the size of the data sent over the wire. Currently it only supports ‘zlib’ compression but more will come later.

You can configure it like this:

```
akka {
  remote {
    compression-scheme = "zlib" # Options: "zlib" (lzf to come), leave out for no compression
    zlib-compression-level = 6 # Options: 0-9 (1 being fastest and 9 being the most compressed),
  }
}
```

4.9.11 Code provisioning

Akka does currently not support automatic code provisioning but requires you to have the remote actor class files available on both the “client” the “server” nodes. This is something that will be addressed soon. Until then, sorry for the inconvenience.

4.9.12 Subscribe to Remote Client events

Akka has a subscription API for the client event. You can register an Actor as a listener and this actor will have to be able to process these events:

```
sealed trait RemoteClientLifecycleEvent
case class RemoteClientError(
  @BeanProperty cause: Throwable,
  @BeanProperty client: RemoteClientModule,
  @BeanProperty remoteAddress: InetSocketAddress) extends RemoteClientLifecycleEvent

case class RemoteClientDisconnected(
  @BeanProperty client: RemoteClientModule,
  @BeanProperty remoteAddress: InetSocketAddress) extends RemoteClientLifecycleEvent

case class RemoteClientConnected(
  @BeanProperty client: RemoteClientModule,
  @BeanProperty remoteAddress: InetSocketAddress) extends RemoteClientLifecycleEvent
```

```

case class RemoteClientStarted(
  @BeanProperty client: RemoteClientModule,
  @BeanProperty remoteAddress: InetSocketAddress) extends RemoteClientLifecycleEvent

case class RemoteClientShutdown(
  @BeanProperty client: RemoteClientModule,
  @BeanProperty remoteAddress: InetSocketAddress) extends RemoteClientLifecycleEvent

case class RemoteClientWriteFailed(
  @BeanProperty request: AnyRef,
  @BeanProperty cause: Throwable,
  @BeanProperty client: RemoteClientModule,
  @BeanProperty remoteAddress: InetSocketAddress) extends RemoteClientLifecycleEvent

```

So a simple listener actor can look like this:

```

import akka.actor.Actor
import akka.actor.Actor._
import akka.remoteinterface._

val listener = actorOf(new Actor {
  def receive = {
    case RemoteClientError(cause, client, address) => //... act upon error
    case RemoteClientDisconnected(client, address) => //... act upon disconnection
    case RemoteClientConnected(client, address) => //... act upon connection
    case RemoteClientStarted(client, address) => //... act upon client shutdown
    case RemoteClientShutdown(client, address) => //... act upon client shutdown
    case RemoteClientWriteFailed(request, cause, client, address) => //... act upon write failure
    case _ => // ignore other
  }
}).start()

```

Registration and de-registration can be done like this:

```

remote.addListener(listener)
...
remote.removeListener(listener)

```

4.9.13 Subscribe to Remote Server events

Akka has a subscription API for the ‘RemoteServer’. You can register an Actor as a listener and this actor will have to be able to process these events:

```

sealed trait RemoteServerLifecycleEvent
case class RemoteServerStarted(
  @BeanProperty val server: RemoteServerModule) extends RemoteServerLifecycleEvent
case class RemoteServerShutdown(
  @BeanProperty val server: RemoteServerModule) extends RemoteServerLifecycleEvent
case class RemoteServerError(
  @BeanProperty val cause: Throwable,
  @BeanProperty val server: RemoteServerModule) extends RemoteServerLifecycleEvent
case class RemoteServerClientConnected(
  @BeanProperty val server: RemoteServerModule,
  @BeanProperty val clientAddress: Option[InetSocketAddress]) extends RemoteServerLifecycleEvent
case class RemoteServerClientDisconnected(
  @BeanProperty val server: RemoteServerModule,
  @BeanProperty val clientAddress: Option[InetSocketAddress]) extends RemoteServerLifecycleEvent
case class RemoteServerClientClosed(
  @BeanProperty val server: RemoteServerModule,
  @BeanProperty val clientAddress: Option[InetSocketAddress]) extends RemoteServerLifecycleEvent
case class RemoteServerWriteFailed(

```

```
@BeanProperty request: AnyRef,
@BeanProperty cause: Throwable,
@BeanProperty server: RemoteServerModule,
@BeanProperty clientAddress: Option[InetSocketAddress]) extends RemoteServerLifecycleEvent
```

So a simple listener actor can look like this:

```
import akka.actor.Actor
import akka.actor.Actor._
import akka.remoteinterface._

val listener = actorOf(new Actor {
  def receive = {
    case RemoteServerStarted(server)           => //... act upon server start
    case RemoteServerShutdown(server)          => //... act upon server shutdown
    case RemoteServerError(cause, server)       => //... act upon server error
    case RemoteServerClientConnected(server, clientAddress) => //... act upon client connection
    case RemoteServerClientDisconnected(server, clientAddress) => //... act upon client disconnection
    case RemoteServerClientClosed(server, clientAddress) => //... act upon client connection
    case RemoteServerWriteFailed(request, cause, server, clientAddress) => //... act upon server write failed
  }
}).start()
```

Registration and de-registration can be done like this:

```
remote.addListener(listener)
...
remote.removeListener(listener)
```

4.9.14 Message Serialization

All messages that are sent to remote actors need to be serialized to binary format to be able to travel over the wire to the remote node. This is done by letting your messages extend one of the traits in the ‘akka.serialization.Serializable’ object. If the messages don’t implement any specific serialization trait then the runtime will try to use standard Java serialization.

Here are some examples, but full documentation can be found in the [Serialization \(Scala\)](#).

Scala JSON

```
case class MyMessage(id: String, value: Tuple2[String, Int]) extends Serializable.ScalaJSON[MyMessage]
```

Protobuf

Protobuf message specification needs to be compiled with ‘protoc’ compiler.

```
message ProtobufPOJO {
  required uint64 id = 1;
  required string name = 2;
  required bool status = 3;
}
```

Using the generated message builder to send the message to a remote actor:

```
val resultFuture = actor ? ProtobufPOJO.newBuilder
  .setId(11)
  .setStatus(true)
  .setName("Coltrane")
  .build
```

4.10 Serialization (Scala)

Contents

- **Serialization of ActorRef**
 - Deep serialization of an Actor and ActorRef
 - Helper Type Class for Stateless Actors
 - Helper Type Class for actors with external serializer
- **Serialization of a RemoteActorRef**
- **Deep serialization of a TypedActor**
- **Serialization of a remote typed ActorRef**
- **Compression**
- **Using the Serializable trait and Serializer class for custom serialization**
- **Protobuf**
- **JSON: Scala**
 - Serializer API using type classes
 - Serializer API using reflection
- **SJSON: Scala**
 - Serialization of Embedded Objects
 - Changing property names during serialization
 - Serialization with ignore properties
 - Serialization with Type Hints for Generic Data Members
 - Fighting Type Erasure
 - Type class based Serialization
- **JSON: Java**

Module stability: **SOLID**

4.10.1 Serialization of ActorRef

An Actor can be serialized in two different ways:

- **Serializable RemoteActorRef** - Serialized to an immutable, network-aware Actor reference that can be freely shared across the network. They “remember” and stay mapped to their original Actor instance and host node, and will always work as expected.
- **Serializable LocalActorRef** - Serialized by doing a deep copy of both the ActorRef and the Actor instance itself. Can be used to physically move an Actor from one node to another and continue the execution there.

Both of these can be sent as messages over the network and/or store them to disk, in a persistent storage backend etc.

Actor serialization in Akka is implemented through a type class `Format[T <: Actor]` which publishes the `fromBinary` and `toBinary` methods for serialization. Here’s the complete definition of the type class:

```
/**
 * Type class definition for Actor Serialization
 */
trait FromBinary[T <: Actor] {
  def fromBinary(bytes: Array[Byte], act: T): T
}

trait ToBinary[T <: Actor] {
  def toBinary(t: T): Array[Byte]
}
```

```
// client needs to implement Format[] for the respective actor
trait Format[T <: Actor] extends FromBinary[T] with ToBinary[T]
```

Deep serialization of an Actor and ActorRef

You can serialize the whole actor deeply, e.g. both the `ActorRef` and then instance of its `Actor`. This can be useful if you want to move an actor from one node to another, or if you want to store away an actor, with its state, into a database.

Here is an example of how to serialize an `Actor`.

Step 1: Define the actor

```
class MyActor extends Actor {
  var count = 0

  def receive = {
    case "hello" =>
      count = count + 1
      self.reply("world " + count)
  }
}
```

Step 2: Implement the type class for the actor. `ProtobufProtocol.Counter` is something you need to define yourself, as explained in the `Protobuf` section.

```
import akka.serialization.{Serializer, Format}
import akka.actor.Actor
import akka.actor.Actor._

object BinaryFormatMyActor {
  implicit object MyActorFormat extends Format[MyActor] {
    def fromBinary(bytes: Array[Byte], act: MyActor) = {
      val p = Serializer.ProtoBuf.fromBinary(bytes, Some(classOf[ProtobufProtocol.Counter])).asInstanceOf[ProtobufProtocol.Counter]
      act.count = p.getCount
      act
    }
    def toBinary(ac: MyActor) =
      ProtobufProtocol.Counter.newBuilder.setCount(ac.count).build.toByteArray
  }
}
```

Step 3: Import the type class module definition and serialize / de-serialize

```
it("should be able to serialize and de-serialize a stateful actor") {
  import akka.serialization.ActorSerialization._
  import BinaryFormatMyActor._

  val actor1 = actorOf[MyActor].start()
  (actor1 ? "hello").as[String].getOrElse("_") should equal("world 1")
  (actor1 ? "hello").as[String].getOrElse("_") should equal("world 2")

  val bytes = toBinary(actor1)
  val actor2 = fromBinary(bytes)
  actor2.start()
  (actor2 ? "hello").as[String].getOrElse("_") should equal("world 3")
}
```

Helper Type Class for Stateless Actors

If your actor is stateless, then you can use the helper trait that Akka provides to serialize / de-serialize. Here's the definition:

```
trait StatelessActorFormat[T <: Actor] extends Format[T] {
  def fromBinary(bytes: Array[Byte], act: T) = act
  def toBinary(ac: T) = Array.empty[Byte]
}
```

Then you use it as follows:

```
class MyStatelessActor extends Actor {
  def receive = {
    case "hello" =>
      self.reply("world")
  }
}
```

Just create an object for the helper trait for your actor:

```
object BinaryFormatMyStatelessActor {
  implicit object MyStatelessActorFormat extends StatelessActorFormat[MyStatelessActor]
}
```

and use it for serialization:

```
it("should be able to serialize and de-serialize a stateless actor") {
  import akka.serialization.ActorSerialization._
  import BinaryFormatMyStatelessActor._

  val actor1 = actorOf[MyStatelessActor].start()
  (actor1 ? "hello").as[String].getOrElse("_") should equal("world")
  (actor1 ? "hello").as[String].getOrElse("_") should equal("world")

  val bytes = toBinary(actor1)
  val actor2 = fromBinary(bytes)
  actor2.start()
  (actor2 ? "hello").as[String].getOrElse("_") should equal("world")
}
```

Helper Type Class for actors with external serializer

Use the trait `SerializerBasedActorFormat` for specifying serializers.

```
trait SerializerBasedActorFormat[T <: Actor] extends Format[T] {
  val serializer: Serializer
  def fromBinary(bytes: Array[Byte], act: T) = serializer.fromBinary(bytes, Some(act).self.actorClass)
  def toBinary(ac: T) = serializer.toBinary(ac)
}
```

For a Java serializable actor:

```
class MyJavaSerializableActor extends Actor with scala.Serializable {
  var count = 0

  def receive = {
    case "hello" =>
      count = count + 1
      self.reply("world " + count)
  }
}
```

Create a module for the type class ..

```
import akka.serialization.{SerializerBasedActorFormat, Serializer}

object BinaryFormatMyJavaSerializableActor {
```



```
implicit object MyJavaSerializableActorFormat extends SerializerBasedActorFormat[MyJavaSerializableActor] {
  val serializer = Serializer.Java
}
}
```

and serialize / de-serialize ..

```
it("should be able to serialize and de-serialize a stateful actor with a given serializer") {
  import akka.actor.Actor._
  import akka.serialization.ActorSerialization._
  import BinaryFormatMyJavaSerializableActor._

  val actor1 = actorOf[MyJavaSerializableActor].start()
  (actor1 ? "hello").as[String].getOrElse("_") should equal("world 1")
  (actor1 ? "hello").as[String].getOrElse("_") should equal("world 2")

  val bytes = toBinary(actor1)
  val actor2 = fromBinary(bytes)
  actor2.start()
  (actor2 ? "hello").as[String].getOrElse("_") should equal("world 3")
}
```

4.10.2 Serialization of a RemoteActorRef

You can serialize an ActorRef to an immutable, network-aware Actor reference that can be freely shared across the network, a reference that “remembers” and stay mapped to its original Actor instance and host node, and will always work as expected.

The RemoteActorRef serialization is based upon Protobuf (Google Protocol Buffers) and you don’t need to do anything to use it, it works on any ActorRef.

Currently Akka will **not** autodetect an ActorRef as part of your message and serialize it for you automatically, so you have to do that manually or as part of your custom serialization mechanisms.

Here is an example of how to serialize an Actor.

```
import akka.serialization.RemoteActorSerialization._

val actor1 = actorOf[MyActor]

val bytes = toRemoteActorRefProtocol(actor1).toByteArray
```

To deserialize the ActorRef to a RemoteActorRef you need to use the fromBinaryToRemoteActorRef(bytes: Array[Byte]) method on the ActorRef companion object:

```
import akka.serialization.RemoteActorSerialization._
val actor2 = fromBinaryToRemoteActorRef(bytes)
```

You can also pass in a class loader to load the ActorRef class and dependencies from:

```
import akka.serialization.RemoteActorSerialization._
val actor2 = fromBinaryToRemoteActorRef(bytes, classLoader)
```

4.10.3 Deep serialization of a TypedActor

Serialization of typed actors works almost the same way as untyped actors. You can serialize the whole actor deeply, e.g. both the ‘proxied ActorRef’ and the instance of its TypedActor.

Here is the example from above implemented as a TypedActor.

Step 1: Define the actor

```
import akka.actor.TypedActor

trait MyTypedActor {
  def requestReply(s: String) : String
  def oneWay() : Unit
}

class MyTypedActorImpl extends TypedActor with MyTypedActor {
  var count = 0

  override def requestReply(message: String) : String = {
    count = count + 1
    "world " + count
  }

  override def oneWay() {
    count = count + 1
  }
}
```

Step 2: Implement the type class for the actor

```
import akka.serialization.{Serializer, Format}

class MyTypedActorFormat extends Format[MyTypedActorImpl] {
  def fromBinary(bytes: Array[Byte], act: MyTypedActorImpl) = {
    val p = Serializer.ProtoBuf.fromBinary(bytes, Some(classOf[ProtobufProtocol.Counter])).asInstance
    act.count = p.getCount
    act
  }
  def toBinary(ac: MyTypedActorImpl) = ProtobufProtocol.Counter.newBuilder.setCount(ac.count).build
}
```

Step 3: Import the type class module definition and serialize / de-serialize

```
import akka.serialization.TypedActorSerialization._

val typedActor1 = TypedActor.newInstance(classOf[MyTypedActor], classOf[MyTypedActorImpl], 1000)

val f = new MyTypedActorFormat
val bytes = toBinaryJ(typedActor1, f)

val typedActor2: MyTypedActor = fromBinaryJ(bytes, f) //type hint needed
typedActor2.requestReply("hello")
```

4.10.4 Serialization of a remote typed ActorRef

To deserialize the `TypedActor` to a `RemoteTypedActorRef` (an aspectwerkz proxy to a `RemoteActorRef`) you need to use the `fromBinaryToRemoteTypedActorRef(bytes: Array[Byte])` method on `RemoteTypedActorSerialization` object:

```
import akka.serialization.RemoteTypedActorSerialization._
val typedActor = fromBinaryToRemoteTypedActorRef(bytes)

// you can also pass in a class loader
val typedActor2 = fromBinaryToRemoteTypedActorRef(bytes, classLoader)
```

4.10.5 Compression

Akka has a helper class for doing compression of binary data. This can be useful for example when storing data in one of the backing storages. It currently supports LZF which is a very fast compression algorithm suited for

runtime dynamic compression.

Here is an example of how it can be used:

```
import akka.serialization.Compression

val bytes: Array[Byte] = ...
val compressBytes = Compression.LZF.compress(bytes)
val uncompressBytes = Compression.LZF.uncompress(compressBytes)
```

4.10.6 Using the Serializable trait and Serializer class for custom serialization

If you are sending messages to a remote Actor and these messages implement one of the predefined interfaces/traits in the `akka.serialization.Serializable.*` object, then Akka will transparently detect which serialization format it should use as wire protocol and will automatically serialize and deserialize the message according to this protocol.

Each serialization interface/trait in

- `akka.serialization.Serializable.*`

has a matching serializer in

- `akka.serialization.Serializer.*`

Note however that if you are using one of the Serializable interfaces then you don't have to do anything else in regard to sending remote messages.

The ones currently supported are (besides the default which is regular Java serialization):

- ScalaJSON (Scala only)
- JavaJSON (Java but some Scala structures)
- Protobuf (Scala and Java)

Apart from the above, Akka also supports Scala object serialization through [SJSON](#) that implements APIs similar to `akka.serialization.Serializer.*`. See the section on SJSON below for details.

4.10.7 Protobuf

Akka supports using [Google Protocol Buffers](#) to serialize your objects. Protobuf is a very efficient network serialization protocol which is also used internally by Akka. The remote actors understand Protobuf messages so if you just send them as they are they will be correctly serialized and unserialized.

Here is an example.

Let's say you have this Protobuf message specification that you want to use as message between remote actors. First you need to compile it with 'protoc' compiler.

```
message ProtobufPOJO {
  required uint64 id = 1;
  required string name = 2;
  required bool status = 3;
}
```

When you compile the spec you will among other things get a message builder. You then use this builder to create the messages to send over the wire:

```
val resultFuture = remoteActor ? ProtobufPOJO.newBuilder
  .setId(11)
  .setStatus(true)
  .setName("Coltrane")
  .build
```

The remote Actor can then receive the Protobuf message typed as-is:

```
class MyRemoteActor extends Actor {
  def receive = {
    case pojo: ProtobufPOJO =>
      val id = pojo.getId
      val status = pojo.getStatus
      val name = pojo.getName
      ...
  }
}
```

4.10.8 JSON: Scala

Use the `akka.serialization.Serializable.ScalaJSON` base class with its `toJSON` method. Akka's Scala JSON is based upon the SJSON library.

For your POJOs to be able to serialize themselves you have to extend the `ScalaJSON[]` trait as follows. JSON serialization is based on a type class protocol which you need to define for your own abstraction. The instance of the type class is defined as an implicit object which is used for serialization and de-serialization. You also need to implement the methods in terms of the APIs which sjson publishes.

```
import akka.serialization._
import akka.serialization.Serializable.ScalaJSON
import akka.serialization.JsonSerialization._
import akka.serialization.DefaultProtocol._

case class MyMessage(val id: String, val value: Tuple2[String, Int]) extends ScalaJSON[MyMessage]
// type class instance
implicit val MyMessageFormat: sjson.json.Format[MyMessage] =
  asProduct2("id", "value") (MyMessage) (MyMessage.unapply(_).get)

def toJSON: String = JsValue.toJson(tojson(this))
def toBytes: Array[Byte] = tobinary(this)
def fromBytes(bytes: Array[Byte]) = frombinary[MyMessage](bytes)
def fromJSON(js: String) = fromjson[MyMessage](Js(js))
}

// sample test case
it("should be able to serialize and de-serialize MyMessage") {
  val s = MyMessage("Target", ("cooker", 120))
  s.fromBytes(s.toBytes) should equal(s)
  s.fromJSON(s.toJSON) should equal(s)
}
```

Use `akka.serialization.Serializer.ScalaJSON` to do generic JSON serialization, e.g. serialize object that does not extend `ScalaJSON` using the JSON serializer. Serialization using `Serializer` can be done in two ways :-

1. Type class based serialization (recommended)
2. Reflection based serialization

We will discuss both of these techniques in this section. For more details refer to the discussion in the next section SJSON: Scala.

Serializer API using type classes

Here are the steps that you need to follow:

1. Define your class

```
case class MyMessage(val id: String, val value: Tuple2[String, Int])
```

2. Define the type class instance

```
import akka.serialization.DefaultProtocol._
implicit val MyMessageFormat: sjson.json.Format[MyMessage] =
  asProduct2("id", "value") (MyMessage) (MyMessage.unapply(_).get)
```

3. Serialize

```
import akka.serialization.Serializer.ScalaJSON
import akka.serialization.JsonSerialization._

val o = MyMessage("dg", ("akka", 100))
fromjson[MyMessage](tojson(o)) should equal(o)
frombinary[MyMessage](tobinary(o)) should equal(o)
```

Serializer API using reflection

You can also use the Serializer abstraction to serialize using the reflection based serialization API of sjson. But we recommend using the type class based one, because reflection based serialization has limitations due to type erasure. Here's an example of reflection based serialization:

```
import scala.reflect.BeanInfo
import akka.serialization.Serializer

@BeanInfo case class Foo(name: String) {
  def this() = this(null) // default constructor is necessary for deserialization
}

val foo = Foo("bar")

val json = Serializer.ScalaJSON.toBinary(foo)

val fooCopy = Serializer.ScalaJSON.fromBinary(json) // returns a JsObject as an AnyRef
val fooCopy2 = Serializer.ScalaJSON.fromJSON(new String(json)) // can also take a string as input
val fooCopy3 = Serializer.ScalaJSON.fromBinary[Foo](json).asInstanceOf[Foo]
```

Classes without a @BeanInfo annotation cannot be serialized as JSON. So if you see something like that:

```
scala> Serializer.ScalaJSON.out(bar)
Serializer.ScalaJSON.out(bar)
java.lang.UnsupportedOperationException: Class class Bar not supported for conversion
    at sjson.json.JsBean$class.toJSON(JsBean.scala:210)
    at sjson.json.Serializer$SJSON$.toJSON(Serializer.scala:107)
    at sjson.json.Serializer$SJSON$.out(Serializer.scala:37)
    at sjson.json.Serializer$SJSON$.out(Serializer.scala:107)
    at akka.serialization.Serializer$ScalaJSON...
```

it means, that you haven't got a @BeanInfo annotation on your class.

You may also see this exception when trying to serialize a case class without any attributes, like this:

```
@BeanInfo case class Empty() // cannot be serialized
```

4.10.9 SJSON: Scala

SJJSON supports serialization of Scala objects into JSON. It implements support for built in Scala structures like List, Map or String as well as custom objects. SJJSON is available as an Apache 2 licensed project on Github [here](#).

Example: I have a Scala object as ..

```
val addr = Address("Market Street", "San Francisco", "956871")
```

where Address is a custom class defined by the user. Using SJSON, I can store it as JSON and retrieve as plain old Scala object. Here's the simple assertion that validates the invariant. Note that during de-serialziation, the class name is specified. Hence what it gives back is an instance of Address.

```
val serializer = sjson.json.Serializer.SJSON

addr should equal(
  serializer.in[Address](serializer.out(addr))
```

Note, that the class needs to have a default constructor. Otherwise the deserialization into the specified class will fail.

There are situations, particularly when writing generic persistence libraries in Akka, when the exact class is not known during de-serialization. Using SJSON I can get it as AnyRef or Nothing ..

```
serializer.in[AnyRef](serializer.out(addr))
```

or just as ..

```
serializer.in(serializer.out(addr))
```

What you get back from is a JsValue, an abstraction of the JSON object model. For details of JsValue implementation, refer to [dispatch-json](#) that SJSON uses as the underlying JSON parser implementation. Once I have the JsValue model, I can use use extractors to get back individual attributes ..

```
val serializer = sjson.json.Serializer.SJSON

val a = serializer.in[AnyRef](serializer.out(addr))

// use extractors
val c = 'city ? str
val c(_city) = a
_city should equal("San Francisco")

val s = 'street ? str
val s(_street) = a
_street should equal("Market Street")

val z = 'zip ? str
val z(_zip) = a
_zip should equal("956871")
```

Serialization of Embedded Objects

SJSON supports serialization of Scala objects that have other embedded objects. Suppose you have the following Scala classes .. Here Contact has an embedded Address Map ..

```
@BeanInfo
case class Contact(name: String,
                  @(JSonTypeHint @field)(value = classOf[Address])
                  addresses: Map[String, Address]) {

  override def toString = "name = " + name + " addresses = " +
    addresses.map(a => a._1 + ":" + a._2.toString).mkString(",")
}

@BeanInfo
case class Address(street: String, city: String, zip: String) {
  override def toString = "address = " + street + "/" + city + "/" + zip
}
```

With SJSON, I can do the following:

```
val a1 = Address("Market Street", "San Francisco", "956871")
val a2 = Address("Monroe Street", "Denver", "80231")
val a3 = Address("North Street", "Atlanta", "987671")

val c = Contact("Bob", Map("residence" -> a1, "office" -> a2, "club" -> a3))
val co = serializer.out(c)

val serializer = sjson.json.Serializer.SJSON

// with class specified
c should equal(serializer.in[Contact](co))

// no class specified
val a = serializer.in[AnyRef](co)

// extract name
val n = 'name ? str
val n(_name) = a
"Bob" should equal(_name)

// extract addresses
val addr = 'addresses ? obj
val addr(_addresses) = a

// extract residence from addresses
val res = 'residence ? obj
val res(_raddr) = _addresses

// make an Address bean out of _raddr
val address = JsBean.fromJSON(_raddr, Some(classOf[Address]))
a1 should equal(address)

object r { def ># [T](f: JsF[T]) = f(a.asInstanceOf[JsValue]) }

// still better: chain 'em up
"Market Street" should equal(
  (r ># { ('addresses ? obj) andThen ('residence ? obj) andThen ('street ? str) }) )
```

Changing property names during serialization

```
@BeanInfo
case class Book(id: Number,
  title: String, @(JSONProperty @getter)(value = "ISBN") isbn: String) {

  override def toString = "id = " + id + " title = " + title + " isbn = " + isbn
}
```

When this will be serialized out, the property name will be changed.

```
val b = new Book(100, "A Beautiful Mind", "012-456372")
val jsBook = Js(JsBean.toJSON(b))
val expected_book_map = Map(
  JsString("id") -> JsNumber(100),
  JsString("title") -> JsString("A Beautiful Mind"),
  JsString("ISBN") -> JsString("012-456372")
)
```

Serialization with ignore properties

When serializing objects, some of the properties can be ignored declaratively. Consider the following class declaration:

```
@BeanInfo
case class Journal(id: BigDecimal,
                  title: String,
                  author: String,
                  @(JSONProperty @getter)(ignore = true) issn: String) {

  override def toString =
    "Journal: " + id + "/" + title + "/" + author +
    (issn match {
      case null => ""
      case _ => "/" + issn
    })
}
```

The annotation `@JSONProperty` can be used to selectively ignore fields. When I serialize a `Journal` object out and then back in, the content of `issn` field will be null.

```
val serializer = sjson.json.Serializer.SJSON

it("should ignore issn field") {
  val j = Journal(100, "IEEE Computer", "Alex Payne", "012-456372")
  serializer.in[Journal](serializer.out(j)).asInstanceOf[Journal].issn should equal(null)
}
```

Similarly, we can ignore properties of an object **only** if they are null and not ignore otherwise. Just specify the annotation `@JSONProperty` as `@JSONProperty {val ignoreIfNull = true}`.

Serialization with Type Hints for Generic Data Members

Consider the following Scala class:

```
@BeanInfo
case class Contact(name: String,
                  @(JSONTypeHint @field)(value = classOf[Address])
                  addresses: Map[String, Address]) {

  override def toString = "name = " + name + " addresses = " +
    addresses.map(a => a._1 + ":" + a._2.toString).mkString(",")
}
```

Because of erasure, you need to add the type hint declaratively through the annotation `@JSONTypeHint` that `SJSON` will pick up during serialization. Now we can say:

```
val serializer = sjson.json.Serializer.SJSON

val c = Contact("Bob", Map("residence" -> a1, "office" -> a2, "club" -> a3))
val co = serializer.out(c)

it("should give an instance of Contact") {
  c should equal(serializer.in[Contact](co))
}
```

With optional generic data members, we need to provide the hint to `SJSON` through another annotation `@OptionTypeHint`.

```
@BeanInfo
case class ContactWithOptionalAddr(name: String,
                                   @(JSONTypeHint @field)(value = classOf[Address])
                                   addresses: Map[String, Address]) {
```



```

@OptionTypeHint @field)(value = classOf[Map[_, _]])
addresses: Option[Map[String, Address]]) {

  override def toString = "name = " + name + " " +
    (addresses match {
      case None => ""
      case Some(ad) => " addresses = " + ad.map(a => a._1 + ":" + a._2.toString).mkString(",")
    })
}

```

Serialization works ok with optional members annotated as above.

```

val serializer = sjson.json.Serializer.SJSON

describe("Bean with optional bean member serialization") {
  it("should serialize with Option defined") {
    val c = new ContactWithOptionalAddr("Debasish Ghosh",
      Some(Map("primary" -> new Address("10 Market Street", "San Francisco, CA", "94111"),
        "secondary" -> new Address("3300 Tamarac Drive", "Denver, CO", "98301"))))
    c should equal(
      serializer.in[ContactWithOptionalAddr](serializer.out(c))
    )
  }
}

```

You can also specify a custom `ClassLoader` while using the SJSON serializer:

```

object SJSON {
  val classLoader = //.. specify a custom classloader
}

import SJSON._
serializer.out(..)

//..

```

Fighting Type Erasure

Because of type erasure, it's not always possible to infer the correct type during de-serialization of objects. Consider the following example:

```

abstract class A
@BeanInfo case class B(param1: String) extends A
@BeanInfo case class C(param1: String, param2: String) extends A

@BeanInfo case class D(@JSONTypeHint @field)(value = classOf[A])param1: List[A])

```

and the serialization code like the following:

```

object TestSerialize{
  def main(args: Array[String]) {
    val test1 = new D(List(B("hello1")))
    val json = sjson.json.Serializer.SJSON.out(test1)
    val res = sjson.json.Serializer.SJSON.in[D](json)
    val res1: D = res.asInstanceOf[D]
    println(res1)
  } q
}

```

Note that the type hint on class D says A, but the actual instances that have been put into the object before serialization is one of the derived classes (B). During de-serialization, we have no idea of what can be inside D. The `serializer.in` API will fail since all hint it has is for A, which is abstract. In such cases, we need to handle the de-serialization by using extractors over the underlying data structure that we use for storing JSON objects, which is `JsValue`. Here's an example:

```

val serializer = sjson.json.Serializer.SJSON

val test1 = new D(List(B("hello1")))
val json = serializer.out(test1)

// create a JsValue from the string
val js = Js(new String(json))

// extract the named list argument
val m = (Symbol("param1") ? list)
val m(_m) = js

// extract the string within
val s = (Symbol("param1") ? str)

// form a list of B's
val result = _m.map{ e =>
  val s(_s) = e
  B(_s)
}

// form a D
println("result = " + D(result))

```

The above snippet de-serializes correctly using extractors defined on JsValue. For more details on JsValue and the extractors, please refer to [dispatch-json](#).

NOTE: Serialization with SJSON is based on bean introspection. In the current version of Scala (2.8.0.Beta1 and 2.7.7) there is a bug where bean introspection does not work properly for classes enclosed within another class. Please ensure that the beans are the top level classes in your application. They can be within objects though. A ticket has been filed in the Scala Tracker and also fixed in the trunk. Here's the [ticket](#).

Type class based Serialization

If type erasure hits you, reflection based serialization may not be the right option. In fact the last section shows some of the scenarios which may not be possible to handle using reflection based serialization of sjson. sjson also supports type class based serialization where you can provide a custom protocol for serialization as part of the type class implementation.

Here's a sample session at the REPL which shows the default serialization protocol of sjson:

```

scala> import sjson.json._
import sjson.json._

scala> import DefaultProtocol._
import DefaultProtocol._

scala> val str = "debasish"
str: java.lang.String = debasish

scala> import JsonSerializer._
import JsonSerializer._

scala> toJson(str)
res0: dispatch.json.JsValue = "debasish"

scala> fromjson[String](res0)
res1: String = debasish

```

You can use serialization of generic data types using the default protocol as well:

```
scala> val list = List(10, 12, 14, 18)
list: List[Int] = List(10, 12, 14, 18)

scala> toJson(list)
res2: dispatch.json.JsonValue = [10, 12, 14, 18]

scala> fromJson[List[Int]](res2)
res3: List[Int] = List(10, 12, 14, 18)
```

You can also define your own custom protocol, which as to be an implementation of the following type class:

```
trait Writes[T] {
  def writes(o: T): JsonValue
}

trait Reads[T] {
  def reads(json: JsonValue): T
}

trait Format[T] extends Writes[T] with Reads[T]
```

Consider a case class and a custom protocol to serialize it into JSON. Here's the type class implementation:

```
object Protocols {
  case class Person(lastName: String, firstName: String, age: Int)
  object PersonProtocol extends DefaultProtocol {
    import dispatch.json._
    import JsonSerializer._

    implicit object PersonFormat extends Format[Person] {
      def reads(json: JsonValue): Person = json match {
        case JsonObject(m) =>
          Person(fromJson[String](m(JsonString("lastName"))),
                fromJson[String](m(JsonString("firstName"))), fromJson[Int](m(JsonString("age"))))
        case _ => throw new RuntimeException("JsonObject expected")
      }

      def writes(p: Person): JsonValue =
        JsonObject(List(
          (toJson("lastName").asInstanceOf[JsonString], toJson(p.lastName)),
          (toJson("firstName").asInstanceOf[JsonString], toJson(p.firstName)),
          (toJson("age").asInstanceOf[JsonString], toJson(p.age)) ))
    )
  }
}
```

and the serialization in action in the REPL:

```
scala> import sjson.json._
import sjson.json._

scala> import Protocols._
import Protocols._

scala> import PersonProtocol._
import PersonProtocol._

scala> val p = Person("ghosh", "debasish", 20)
p: sjson.json.Protocols.Person = Person(ghosh,debasish,20)

scala> import JsonSerializer._
import JsonSerializer._

scala> toJson[Person](p)
```

```
res1: dispatch.json.JsValue = {"lastName" : "ghosh", "firstName" : "debasish", "age" : 20}

scala> fromjson[Person](res1)
res2: sjson.json.Protocols.Person = Person(ghosh,debasish,20)
```

There are other nifty ways to implement case class serialization using sjson. For more details, have a look at the [wiki](#) for sjson.

4.10.10 JSON: Java

Use the `akka.serialization.Serializable.JavaJSON` base class with its `toJSON` method. Akka's Java JSON is based upon the Jackson library.

For your POJOs to be able to serialize themselves you have to extend the `JavaJSON` base class.

```
import akka.serialization.Serializable.JavaJSON;
import akka.serialization.SerializerFactory;

class MyMessage extends JavaJSON {
  private String name = null;
  public MyMessage(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
}

MyMessage message = new MyMessage("json");
String json = message.toJSON();
SerializerFactory factory = new SerializerFactory();
MyMessage messageCopy = factory.getJavaJSON().in(json);
```

Use the `akka.serialization.SerializerFactory.getJavaJSON` to do generic JSON serialization, e.g. serialize object that does not extend `JavaJSON` using the JSON serializer.

```
Foo foo = new Foo();
SerializerFactory factory = new SerializerFactory();
String json = factory.getJavaJSON().out(foo);
Foo fooCopy = factory.getJavaJSON().in(json, Foo.class);
```

4.11 Fault Tolerance Through Supervisor Hierarchies (Scala)

Contents

- [Concurrency](#)
- [Distributed actors](#)
- [Supervision](#)
 - [OneForOne](#)
 - [AllForOne](#)
 - [Restart callbacks](#)
 - [Defining a supervisor's restart strategy](#)
 - [Defining actor life-cycle](#)
- [Supervising Actors](#)
 - [Declarative supervisor configuration](#)
 - [Declaratively define actors as remote services](#)
 - [Programmatic linking and supervision of Actors](#)
 - [The supervising actor's side of things](#)
 - [The supervised actor's side of things](#)
 - [Reply to initial senders](#)
 - [Handling too many actor restarts within a specific time limit](#)
- [Supervising Typed Actors](#)
 - [Declarative supervisor configuration](#)
 - [Restart callbacks](#)
 - [Programatic linking and supervision of TypedActors](#)

Module stability: **SOLID**

The “let it crash” approach to fault/error handling, implemented by linking actors, is very different to what Java and most non-concurrency oriented languages/frameworks have adopted. It's a way of dealing with failure that is designed for concurrent and distributed systems.

4.11.1 Concurrency

Throwing an exception in concurrent code (let's assume we are using non-linked actors), will just simply blow up the thread that currently executes the actor.

- There is no way to find out that things went wrong (apart from inspecting the stack trace).
- There is nothing you can do about it.

Here actors provide a clean way of getting notification of the error and do something about it.

Linking actors also allow you to create sets of actors where you can be sure that either:

- All are dead
- None are dead

This is very useful when you have thousands of concurrent actors. Some actors might have implicit dependencies and together implement a service, computation, user session etc.

It encourages non-defensive programming. Don't try to prevent things from go wrong, because they will, whether you want it or not. Instead; expect failure as a natural state in the life-cycle of your app, crash early and let someone else (that sees the whole picture), deal with it.

4.11.2 Distributed actors

You can't build a fault-tolerant system with just one single box - you need at least two. Also, you (usually) need to know if one box is down and/or the service you are talking to on the other box is down. Here actor supervision/linking is a critical tool for not only monitoring the health of remote services, but to actually manage

the service, do something about the problem if the actor or node is down. Such as restarting actors on the same node or on another node.

In short, it is a very different way of thinking, but a way that is very useful (if not critical) to building fault-tolerant highly concurrent and distributed applications, which is as valid if you are writing applications for the JVM or the Erlang VM (the origin of the idea of “let-it-crash” and actor supervision).


4.11.3 Supervision

Supervisor hierarchies originate from [Erlang’s OTP framework](#).


A supervisor is responsible for starting, stopping and monitoring its child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary. This makes for a completely different view on how to write fault-tolerant servers. Instead of trying all things possible to prevent an error from happening, this approach embraces failure. It shifts the view to look at errors as something natural and something that **will** happen, instead of trying to prevent it; embraces it. Just “Let It Crash”, since the components will be reset to a stable state and restarted upon failure.

Akka has two different restart strategies; All-For-One and One-For-One. Best explained using some pictures (referenced from erlang.org):

OneForOne

The OneForOne fault handler will restart only the component that has crashed.  image:http://www.erlang.org/doc/design_principles/sup4.gif

AllForOne

The AllForOne fault handler will restart all the components that the supervisor is managing, including the one that have crashed. This strategy should be used when you have a certain set of components that are coupled in some way that if one is crashing they all need to be reset to a stable state before continuing.  image:http://www.erlang.org/doc/design_principles/sup5.gif

Restart callbacks

There are two different callbacks that the Typed Actor and Actor can hook in to:

- Pre restart
- Post restart

These are called prior to and after the restart upon failure and can be used to clean up and reset/reinitialize state upon restart. This is important in order to reset the component failure and leave the component in a fresh and stable state before consuming further messages.

Defining a supervisor’s restart strategy

Both the Typed Actor supervisor configuration and the Actor supervisor configuration take a ‘FaultHandlingStrategy’ instance which defines the fault management. The different strategies are:

- AllForOne
- OneForOne

These have the semantics outlined in the section above.

Here is an example of how to define a restart strategy:

```
AllForOneStrategy( //FaultHandlingStrategy; AllForOneStrategy or OneForOneStrategy
  List(classOf[Exception]), //What exceptions will be handled
  3,                        // maximum number of restart retries
  5000                      // within time in millis
)
```

Defining actor life-cycle

The other common configuration element is the “LifeCycle” which defines the life-cycle. The supervised actor can define one of two different life-cycle configurations:

- Permanent: which means that the actor will always be restarted.
- Temporary: which means that the actor will **not** be restarted, but it will be shut down through the regular shutdown process so the ‘postStop’ callback function will be called.

Here is an example of how to define the life-cycle:

```
Permanent // means that the component will always be restarted
Temporary // means that it will not be restarted, but it will be shut
           // down through the regular shutdown process so the 'postStop' hook will be called
```

4.11.4 Supervising Actors

Declarative supervisor configuration

The Actor’s supervision can be declaratively defined by creating a “Supervisor” factory object. Here is an example:

```
val supervisor = Supervisor(
  SupervisorConfig(
    AllForOneStrategy(List(classOf[Exception]), 3, 1000),
    Supervise(
      actorOf[MyActor1],
      Permanent) ::
    Supervise(
      actorOf[MyActor2],
      Permanent) ::
    Nil))
```

Supervisors created like this are implicitly instantiated and started.

To configure a handler function for when the actor underlying the supervisor receives a `MaximumNumberOfRestartsWithinTimeRangeReached` message, you can specify a function of type `(ActorRef, MaximumNumberOfRestartsWithinTimeRangeReached) => Unit` when creating the `SupervisorConfig`. This handler will be called with the `ActorRef` of the supervisor and the `MaximumNumberOfRestartsWithinTimeRangeReached` message.

```
val handler = {
  (supervisor:ActorRef,max:MaximumNumberOfRestartsWithinTimeRangeReached) => EventHandler.notify(...)
}

val supervisor = Supervisor(
  SupervisorConfig(
    AllForOneStrategy(List(classOf[Exception]), 3, 1000),
    Supervise(
      actorOf[MyActor1],
      Permanent) ::
    Supervise(
      actorOf[MyActor2],
      Permanent) ::
    Nil), handler)
```

You can link and unlink actors from a declaratively defined supervisor using the ‘link’ and ‘unlink’ methods:

```
val supervisor = Supervisor(...)
supervisor.link(..)
supervisor.unlink(..)
```

You can also create declarative supervisors through the ‘SupervisorFactory’ factory object. Use this factory instead of the ‘Supervisor’ factory object if you want to control instantiation and starting of the Supervisor, if not then it is easier and better to use the ‘Supervisor’ factory object.

Example usage:

```
val factory = SupervisorFactory(
  SupervisorConfig(
    OneForOneStrategy(List(classOf[Exception]), 3, 10),
    Supervise(
      myFirstActor,
      Permanent) ::
    Supervise(
      mySecondActor,
      Permanent) ::
    Nil))
```

Then create a new instance our Supervisor and start it up explicitly.

```
val supervisor = factory.newInstance
supervisor.start // start up all managed servers
```

Declaratively define actors as remote services

You can declaratively define an actor to be available as a remote actor by specifying **true** for `registerAsRemoteService`.

Here is an example:

```
val supervisor = Supervisor(
  SupervisorConfig(
    AllForOneStrategy(List(classOf[Exception]), 3, 1000),
    Supervise(
      actorOf[MyActor1],
      Permanent,
      **true**)
    :: Nil))
```

Programmatic linking and supervision of Actors

Actors can at runtime create, spawn, link and supervise other actors. Linking and unlinking is done using one of the ‘link’ and ‘unlink’ methods available in the ‘ActorRef’ (therefore prefixed with ‘self’ in these examples).

Here is the API and how to use it from within an ‘Actor’:

```
// link and unlink actors
self.link(actorRef)
self.unlink(actorRef)

// starts and links Actors atomically
self.startLink(actorRef)

// spawns (creates and starts) actors
self.spawn[MyActor]
self.spawnRemote[MyActor]
```



```
// spawns and links Actors atomically
self.spawnLink[MyActor]
self.spawnLinkRemote[MyActor]
```

A child actor can tell the supervising actor to unlink him by sending him the ‘Unlink(this)’ message. When the supervisor receives the message he will unlink and shut down the child. The supervisor for an actor is available in the ‘supervisor: Option[Actor]’ method in the ‘ActorRef’ class. Here is how it can be used.

```
if (supervisor.isDefined) supervisor.get ! Unlink(self)

// Or shorter using 'foreach':

supervisor.foreach(_ ! Unlink(self))
```

The supervising actor’s side of things

If a linked Actor is failing and throws an exception then an “Exit(deadActor, cause)” message will be sent to the supervisor (however you should never try to catch this message in your own message handler, it is managed by the runtime).

The supervising Actor also needs to define a fault handler that defines the restart strategy the Actor should accommodate when it traps an “Exit” message. This is done by setting the “faultHandler” field.

```
protected var faultHandler: FaultHandlingStrategy
```

The different options are:

- AllForOneStrategy(trapExit, maxNrOfRetries, withinTimeRange)
 - trapExit is a List or Array of classes inheriting from Throwable, they signal which types of exceptions this actor will handle
- OneForOneStrategy(trapExit, maxNrOfRetries, withinTimeRange)
 - trapExit is a List or Array of classes inheriting from Throwable, they signal which types of exceptions this actor will handle

Here is an example:

```
self.faultHandler = AllForOneStrategy(List(classOf[Throwable]), 3, 1000)
```

Putting all this together it can look something like this:

```
class MySupervisor extends Actor {
  self.faultHandler = OneForOneStrategy(List(classOf[Throwable]), 5, 5000)

  def receive = {
    case Register(actor) =>
      self.link(actor)
  }
}
```

You can also link an actor from outside the supervisor like this:

```
val supervisor = Actor.registry.actorsFor(classOf[MySupervisor]).head
supervisor.link(actor)
```

The supervised actor’s side of things

The supervised actor needs to define a life-cycle. This is done by setting the lifeCycle field as follows:

```
self.lifeCycle = Permanent // Permanent or Temporary or UndefinedLifeCycle
```

In the supervised Actor you can override the “preRestart” and “postRestart” callback methods to add hooks into the restart process. These methods take the reason for the failure, e.g. the exception that caused termination and restart of the actor as argument. It is in these methods that **you** have to add code to do cleanup before termination and initialization after restart. Here is an example:

```
class FaultTolerantService extends Actor {
  override def preRestart(reason: Throwable) {
    ... // clean up before restart
  }

  override def postRestart(reason: Throwable) {
    ... // reinit stable state after restart
  }
}
```

Reply to initial senders

Supervised actors have the option to reply to the initial sender within preRestart, postRestart and postStop. A reply within these methods is possible after receive has thrown an exception. When receive returns normally it is expected that any necessary reply has already been done within receive. Here's an example.

```
class FaultTolerantService extends Actor {
  def receive = {
    case msg => {
      // do something that may throw an exception
      // ...

      self.reply("ok")
    }
  }

  override def preRestart(reason: scala.Throwable) {
    self.tryReply(reason.getMessage)
  }

  override def postStop() {
    self.tryReply("stopped by supervisor")
  }
}
```

- A reply within preRestart or postRestart must be a try reply via *self.tryReply* because an *self.reply* will throw an exception when the actor is restarted without having failed. This can be the case in context of AllForOne restart strategies.
- A reply within postStop must be a reply via *self.tryReply* because an *self.reply* will throw an exception when the actor has been stopped by the application (and not by a supervisor) after successful execution of receive (or no execution at all).

Handling too many actor restarts within a specific time limit

If you remember, when you define the ‘RestartStrategy’ you also defined maximum number of restart retries within time in millis.

```
AllForOneStrategy( //Restart policy, AllForOneStrategy or OneForOneStrategy
  List(classOf[Exception]), //What kinds of exception it will handle
  3, // maximum number of restart retries
  5000 // within time in millis
)
```

Now, what happens if this limit is reached?

What will happen is that the failing actor will send a system message to its supervisor called ‘MaximumNumberOfRestartsWithinTimeRangeReached’ with the following signature:

```
case class MaximumNumberOfRestartsWithinTimeRangeReached(
  victim: ActorRef, maxNrOfRetries: Int, withinTimeRange: Int, lastExceptionCausingRestart: Throwable)
```

If you want to be able to take action upon this event (highly recommended) then you have to create a message handle for it in the supervisor.

Here is an example:

```
val supervisor = actorOf(new Actor{
  self.faultHandler = OneForOneStrategy(List(classOf[Throwable]), 5, 5000)
  protected def receive = {
    case MaximumNumberOfRestartsWithinTimeRangeReached(
      victimActorRef, maxNrOfRetries, withinTimeRange, lastExceptionCausingRestart) =>
      ... // handle the error situation
  }
}).start()
```

You will also get this log warning similar to this:

```
WAR [20100715-14:05:25.821] actor: Maximum number of restarts [5] within time range [5000] reached
WAR [20100715-14:05:25.821] actor: Will *not* restart actor [Actor[akka.actor.SupervisorHierarch...]]
WAR [20100715-14:05:25.821] actor: Last exception causing restart was [akka.actor.SupervisorHierarch...]
```

If you don’t define a message handler for this message then you don’t get an error but the message is simply not sent to the supervisor. Instead you will get a log warning.

4.11.5 Supervising Typed Actors

Declarative supervisor configuration

To configure Typed Actors for supervision you have to consult the ‘TypedActorConfigurator’ and its ‘configure’ method. This method takes a ‘RestartStrategy’ and an array of ‘Component’ definitions defining the Typed Actors and their ‘LifeCycle’. Finally you call the ‘supervise’ method to start everything up. The configuration elements reside in the ‘akka.config.JavaConfig’ class and need to be imported statically.

Here is an example:

```
import akka.config.Supervision._

val manager = new TypedActorConfigurator

manager.configure(
  AllForOneStrategy(List(classOf[Exception]), 3, 1000),
  List(
    SuperviseTypedActor(
      Foo.class,
      FooImpl.class,
      Permanent,
      1000),
    new SuperviseTypedActor(
      Bar.class,
      BarImpl.class,
      Permanent,
      1000)
  )).supervise
```

Then you can retrieve the Typed Actor as follows:

```
Foo foo = manager.getInstance(classOf[Foo])
```

Restart callbacks

Programatic linking and supervision of TypedActors

TypedActors can be linked and unlinked just like actors - in fact the linking is done on the underlying actor:

```
TypedActor.link(supervisor, supervised)

TypedActor.unlink(supervisor, supervised)
```

If the parent TypedActor (supervisor) wants to be able to do handle failing child TypedActors, e.g. be able restart the linked TypedActor according to a given fault handling scheme then it has to set its 'trapExit' flag to an array of Exceptions that it wants to be able to trap:

```
TypedActor.faultHandler(supervisor, AllForOneStrategy(Array(classOf[IOException]), 3, 2000))
```

For convenience there is an overloaded link that takes trapExit and faultHandler for the supervisor as arguments. Here is an example:

```
import akka.actor.TypedActor._

val foo = newInstance(classOf[Foo], 1000)
val bar = newInstance(classOf[Bar], 1000)

link(foo, bar, new AllForOneStrategy(Array(classOf[IOException]), 3, 2000))

// alternative: chaining
bar = faultHandler(foo, new AllForOneStrategy(Array(classOf[IOException]), 3, 2000))
    .newInstance(Bar.class, 1000)

link(foo, bar
```

4.12 Dispatchers (Scala)

Contents

- Default dispatcher
- Setting the dispatcher
- Types of dispatchers
 - Thread-based
 - Event-based
 - Priority event-based
 - Work-stealing event-based
- Making the Actor mailbox bounded
 - Global configuration
 - Per-instance based configuration

Module stability: **SOLID**

The Dispatcher is an important piece that allows you to configure the right semantics and parameters for optimal performance, throughput and scalability. Different Actors have different needs.

Akka supports dispatchers for both event-driven lightweight threads, allowing creation of millions of threads on a single workstation, and thread-based Actors, where each dispatcher is bound to a dedicated OS thread.

The event-based Actors currently consume ~600 bytes per Actor which means that you can create more than 6.5 million Actors on 4 GB RAM.

4.12.1 Default dispatcher

For most scenarios the default settings are the best. Here we have one single event-based dispatcher for all Actors created. The dispatcher used is this one:

```
Dispatchers.globalExecutorBasedEventDrivenDispatcher
```

But if you feel that you are starting to contend on the single dispatcher (the ‘Executor’ and its queue) or want to group a specific set of Actors for a dedicated dispatcher for better flexibility and configurability then you can override the defaults and define your own dispatcher. See below for details on which ones are available and how they can be configured.

4.12.2 Setting the dispatcher

Normally you set the dispatcher from within the Actor itself. The dispatcher is defined by the ‘dispatcher: MessageDispatcher’ member field in ‘ActorRef’.

```
class MyActor extends Actor {
  self.dispatcher = ... // set the dispatcher
  ...
}
```

You can also set the dispatcher for an Actor **before** it has been started:

```
actorRef.dispatcher = dispatcher
```

4.12.3 Types of dispatchers

There are six different types of message dispatchers:

- Thread-based
- Event-based
- Priority event-based
- Work-stealing

Factory methods for all of these, including global versions of some of them, are in the ‘akka.dispatch.Dispatchers’ object.

Let’s now walk through the different dispatchers in more detail.

Thread-based

The ‘ThreadBasedDispatcher’ binds a dedicated OS thread to each specific Actor. The messages are posted to a ‘LinkedBlockingQueue’ which feeds the messages to the dispatcher one by one. A ‘ThreadBasedDispatcher’ cannot be shared between actors. This dispatcher has worse performance and scalability than the event-based dispatcher but works great for creating “daemon” Actors that consumes a low frequency of messages and are allowed to go off and do their own thing for a longer period of time. Another advantage with this dispatcher is that Actors do not block threads for each other.

It would normally be used from within the actor like this:

```
class MyActor extends Actor {
  self.dispatcher = Dispatchers.newThreadBasedDispatcher(self)
  ...
}
```

Event-based

The ‘`ExecutorBasedEventDrivenDispatcher`’ binds a set of Actors to a thread pool backed up by a ‘`BlockingQueue`’. This dispatcher is highly configurable and supports a fluent configuration API to configure the ‘`BlockingQueue`’ (type of queue, max items etc.) as well as the thread pool.

The event-driven dispatchers **must be shared** between multiple Actors. One best practice is to let each top-level Actor, e.g. the Actors you define in the declarative supervisor config, to get their own dispatcher but reuse the dispatcher for each new Actor that the top-level Actor creates. But you can also share dispatcher between multiple top-level Actors. This is very use-case specific and needs to be tried out on a case by case basis. The important thing is that Akka tries to provide you with the freedom you need to design and implement your system in the most efficient way in regards to performance, throughput and latency.

It comes with many different predefined `BlockingQueue` configurations: * `Bounded LinkedBlockingQueue` * `Unbounded LinkedBlockingQueue` * `Bounded ArrayBlockingQueue` * `Unbounded ArrayBlockingQueue` * `SynchronousQueue`

You can also set the rejection policy that should be used, e.g. what should be done if the dispatcher (e.g. the Actor) can’t keep up and the mailbox is growing up to the limit defined. You can choose between four different rejection policies:

- `java.util.concurrent.ThreadPoolExecutor.CallerRuns` - will run the message processing in the caller’s thread as a way to slow him down and balance producer/consumer
- `java.util.concurrent.ThreadPoolExecutor.AbortPolicy` - rejected messages by throwing a ‘`RejectedExecutionException`’
- `java.util.concurrent.ThreadPoolExecutor.DiscardPolicy` - discards the message (throws it away)
- `java.util.concurrent.ThreadPoolExecutor.DiscardOldestPolicy` - discards the oldest message in the mailbox (throws it away)

You can read more about these policies [here](#).

Here is an example:

```
import akka.actor.Actor
import akka.dispatch.Dispatchers
import java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy

class MyActor extends Actor {
  self.dispatcher = Dispatchers.newExecutorBasedEventDrivenDispatcher(name)
    .withNewThreadPoolWithLinkedBlockingQueueWithCapacity(100)
    .setCorePoolSize(16)
    .setMaxPoolSize(128)
    .setKeepAliveTimeInMillis(60000)
    .setRejectionPolicy(new CallerRunsPolicy)
    .build
  ...
}
```

This ‘`ExecutorBasedEventDrivenDispatcher`’ allows you to define the ‘throughput’ it should have. This defines the number of messages for a specific Actor the dispatcher should process in one single sweep. Setting this to a higher number will increase throughput but lower fairness, and vice versa. If you don’t specify it explicitly then it uses the default value defined in the ‘`akka.conf`’ configuration file:

```
actor {
  throughput = 5
}
```

If you don’t define a the ‘throughput’ option in the configuration file then the default value of ‘5’ will be used.

Browse the ScalaDoc or look at the code for all the options available.

Priority event-based

Sometimes it's useful to be able to specify priority order of messages, that is done by using `PriorityExecutorBasedEventDrivenDispatcher` and supply a `java.util.Comparator[MessageInvocation]` or use a `akka.dispatch.PriorityGenerator` (recommended):

Creating a `PriorityExecutorBasedEventDrivenDispatcher` using `PriorityGenerator`:

```
import akka.dispatch._
import akka.actor._

val gen = PriorityGenerator { // Create a new PriorityGenerator, lower prio means more important
  case 'highpriority => 0    // 'highpriority messages should be treated first if possible
  case 'lowpriority  => 100  // 'lowpriority messages should be treated last if possible
  case otherwise     => 50   // We default to 50
}

val a = Actor.actorOf( // We create a new Actor that just prints out what it processes
  new Actor {
    def receive = {
      case x => println(x)
    }
  })

// We create a new Priority dispatcher and seed it with the priority generator
a.dispatcher = new PriorityExecutorBasedEventDrivenDispatcher("foo", gen)
a.start // Start the Actor

a.dispatcher.suspend(a) // Suspending the actor so it doesn't start to treat the messages before

a ! 'lowpriority
a ! 'lowpriority
a ! 'highpriority
a ! 'pigdog
a ! 'pigdog2
a ! 'pigdog3
a ! 'highpriority

a.dispatcher.resume(a) // Resuming the actor so it will start treating its messages
```

Prints:

```
'highpriority 'highpriority 'pigdog 'pigdog2 'pigdog3 'lowpriority 'lowpriority
```

Work-stealing event-based

The `'ExecutorBasedEventDrivenWorkStealingDispatcher` is a variation of the `'ExecutorBasedEventDrivenDispatcher` in which Actors of the same type can be set up to share this dispatcher and during execution time the different actors will steal messages from other actors if they have less messages to process. This can be a great way to improve throughput at the cost of a little higher latency.

Normally the way you use it is to create an Actor companion object to hold the dispatcher and then set in in the Actor explicitly.

```
object MyActor {
  val dispatcher = Dispatchers.newExecutorBasedEventDrivenWorkStealingDispatcher(name).build
}

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

Here is an article with some more information: [Load Balancing Actors with Work Stealing Techniques](#) Here is another article discussing this particular dispatcher: [Flexible load balancing with Akka in Scala](#)

4.12.4 Making the Actor mailbox bounded

Global configuration

You can make the Actor mailbox bounded by a capacity in two ways. Either you define it in the configuration file under ‘default-dispatcher’. This will set it globally.

```
actor {
  default-dispatcher {
    mailbox-capacity = -1          # If negative (or zero) then an unbounded mailbox is used (default)
                                # If positive then a bounded mailbox is used and the capacity is set
  }
}
```

Per-instance based configuration

You can also do it on a specific dispatcher instance.

For the ‘ExecutorBasedEventDrivenDispatcher’ and the ‘ExecutorBasedWorkStealingDispatcher’ you can do it through their constructor

```
class MyActor extends Actor {
  val mailboxCapacity = BoundedMailbox(capacity = 100)
  self.dispatcher = Dispatchers.newExecutorBasedEventDrivenDispatcher(name, throughput, mailboxCapacity)
  ...
}
```

For the ‘ThreadBasedDispatcher’, it is non-shareable between actors, and associates a dedicated Thread with the actor. Making it bounded (by specifying a capacity) is optional, but if you do, you need to provide a pushTimeout (default is 10 seconds). When trying to send a message to the Actor it will throw a MessageQueueAppend-FailedException(“BlockingMessageTransferQueue transfer timed out”) if the message cannot be added to the mailbox within the time specified by the pushTimeout.

```
class MyActor extends Actor {
  import akka.util.duration._
  self.dispatcher = Dispatchers.newThreadBasedDispatcher(self, mailboxCapacity = 100,
    pushTimeout = 10 seconds)
  ...
}
```

4.13 Routing (Scala)

Contents

- [Dispatcher](#)
- [LoadBalancer](#)
- [Actor Pool](#)
 - [Selection](#)
 - * [Partial Fills](#)
 - [Capacity](#)
 - [Filtering](#)
 - [Examples](#)

Akka-core includes some building blocks to build more complex message flow handlers, they are listed and explained below:

4.13.1 Dispatcher

A Dispatcher is an actor that routes incoming messages to outbound actors.

To use it you can either create a Dispatcher through the `dispatcherActor()` factory method

```
import akka.actor.Actor._
import akka.actor.Actor
import akka.routing.Routing._

//Our message types
case object Ping
case object Pong

//Two actors, one named Pinger and one named Ponger
//The actor(pf) method creates an anonymous actor and starts it
val pinger = actorOf(new Actor { def receive = { case x => println("Pinger: " + x) } }).start()
val ponger = actorOf(new Actor { def receive = { case x => println("Ponger: " + x) } }).start()

//A dispatcher that dispatches Ping messages to the pinger
//and Pong messages to the ponger
val d = dispatcherActor {
  case Ping => pinger
  case Pong => ponger
}

d ! Ping //Prints "Pinger: Ping"
d ! Pong //Prints "Ponger: Pong"
```

Or by mixing in `akka.patterns.Dispatcher`:

```
import akka.actor.Actor
import akka.actor.Actor._
import akka.routing.Dispatcher

//Our message types
case object Ping
case object Pong

class MyDispatcher extends Actor with Dispatcher {
  //Our pinger and ponger actors
  val pinger = actorOf(new Actor { def receive = { case x => println("Pinger: " + x) } }).start()
  val ponger = actorOf(new Actor { def receive = { case x => println("Ponger: " + x) } }).start()
  //When we get a ping, we dispatch to the pinger
  //When we get a pong, we dispatch to the ponger
```

```

def routes = {
  case Ping => pinger
  case Pong => ponger
}

//Create an instance of our dispatcher, and start it
val d = actorOf[MyDispatcher].start()

d ! Ping //Prints "Pinger: Ping"
d ! Pong //Prints "Ponger: Pong"

```

4.13.2 LoadBalancer

A LoadBalancer is an actor that forwards messages it receives to a boundless sequence of destination actors.

Example using the `loadBalancerActor()` factory method:

```

import akka.actor.Actor._
import akka.actor.Actor
import akka.routing.Routing._
import akka.routing.CyclicIterator

//Our message types
case object Ping
case object Pong

//Two actors, one named Pinger and one named Ponger
//The actor(pf) method creates an anonymous actor and starts it

val pinger = actorOf(new Actor { def receive = { case x => println("Pinger: " + x) } }).start()
val ponger = actorOf(new Actor { def receive = { case x => println("Ponger: " + x) } }).start()

//A load balancer that given a sequence of actors dispatches them accordingly
//a CyclicIterator works in a round-robin-fashion

val d = loadBalancerActor( new CyclicIterator( List(pinger,ponger) ) )

d ! Pong //Prints "Pinger: Pong"
d ! Pong //Prints "Ponger: Pong"
d ! Ping //Prints "Pinger: Ping"
d ! Ping //Prints "Ponger: Ping"

```

Or by mixing in `akka.routing.LoadBalancer`

```

import akka.actor._
import akka.actor.Actor._
import akka.routing.{ LoadBalancer, CyclicIterator }

//Our message types
case object Ping
case object Pong

//A load balancer that balances between a pinger and a ponger
class MyLoadBalancer extends Actor with LoadBalancer {
  val pinger = actorOf(new Actor { def receive = { case x => println("Pinger: " + x) } }).start()
  val ponger = actorOf(new Actor { def receive = { case x => println("Ponger: " + x) } }).start()

  val seq = new CyclicIterator[ActorRef](List(pinger,ponger))
}

//Create an instance of our loadbalancer, and start it

```

```
val d = actorOf[MyLoadBalancer].start()

d ! Pong //Prints "Pinger: Pong"
d ! Pong //Prints "Ponger: Pong"
d ! Ping //Prints "Pinger: Ping"
d ! Ping //Prints "Ponger: Ping"
```

Also, instead of using the `CyclicIterator`, you can create your own message distribution algorithms, there's already one that dispatches depending on target mailbox size, effectively dispatching to the one that's got fewest messages to process right now.

Example <http://pastie.org/984889>

You can also send a `'Routing.Broadcast(msg)'` message to the router to have it be broadcasted out to all the actors it represents.

```
router ! Routing.Broadcast(PoisonPill)
```

4.13.3 Actor Pool

An actor pool is similar to the load balancer is that it routes incoming messages to other actors. It has different semantics however when it comes to how those actors are managed and selected for dispatch. Therein lies the difference. The pool manages, from start to shutdown, the lifecycle of all delegated actors. The number of actors in a pool can be fixed or grow and shrink over time. Also, messages can be routed to more than one actor in the pool if so desired. This is a useful little feature for accounting for expected failure - especially with remoting - where you can invoke the same request of multiple actors and just take the first, best response.

The actor pool is built around three concepts: capacity, filtering and selection.

Selection

All pools require a *Selector* to be mixed-in. This trait controls how and how many actors in the pool will receive the incoming message. Define *selectionCount* to some positive number greater than one to route to multiple actors. Currently two are provided:

- **SmallestMailboxSelector** - Using the exact same logic as the iterator of the same name, the pooled actor with the fewest number of pending messages will be chosen.
- **RoundRobinSelector** - Performs a very simple index-based selection, wrapping around the end of the list, very much like the `CyclicIterator` does.

Partial Fills

When selecting more than one pooled actor, its possible that in order to fulfill the requested amount, the selection set must contain duplicates. By setting *partialFill* to **true**, you instruct the selector to return only unique actors from the pool.

Capacity

As you'd expect, capacity traits determine how the pool is funded with actors. There are two types of strategies that can be employed:

- **FixedCapacityStrategy** - When you mix this into your actor pool, you define a pool size and when the pool is started, it will have that number of actors within to which messages will be delegated.
- **BoundedCapacityStrategy** - When you mix this into your actor pool, you define upper and lower bounds, and when the pool is started, it will have the minimum number of actors in place to handle messages. You must also mix-in a *Capacitor* and a *Filter* when using this strategy (see below).

The *BoundedCapacityStrategy* requires additional logic to function. Specifically it requires a *Capacitor* and a *Filter*. Capacitors are used to determine the pressure that the pool is under and provide a (usually) raw reading of this information. Currently we provide for the use of either mailbox backlog or active futures count as a means of evaluating pool pressure. Each expresses itself as a simple number - a reading of the number of actors either with mailbox sizes over a certain threshold or blocking a thread waiting on a future to complete or expire.

Filtering

A *Filter* is a trait that modifies the raw pressure reading returned from a Capacitor such that it drives the adjustment of the pool capacity to a desired end. More simply, if we just used the pressure reading alone, we might only ever increase the size of the pool (to respond to overload) or we might only have a single mechanism for reducing the pool size when/if it became necessary. This behavior is fully under your control through the use of *Filters*. Let's take a look at some code to see how this works:

```
trait BoundedCapacitor
{
  def lowerBound:Int
  def upperBound:Int

  def capacity(delegates:Seq[ActorRef]):Int =
  {
    val current = delegates length
    var delta = _eval(delegates)
    val proposed = current + delta

    if (proposed < lowerBound) delta += (lowerBound - proposed)
    else if (proposed > upperBound) delta -= (proposed - upperBound)

    delta
  }

  protected def _eval(delegates:Seq[ActorRef]):Int
}

trait CapacityStrategy
{
  import ActorPool._

  def pressure(delegates:Seq[ActorRef]):Int
  def filter(pressure:Int, capacity:Int):Int

  protected def _eval(delegates:Seq[ActorRef]):Int = filter(pressure(delegates), delegates.size)
}
```

Here we see how the filter function will have the chance to modify the pressure reading to influence the capacity change. You are free to implement filter() however you like. We provide a *Filter* trait that evaluates both a rampup and a backoff subfilter to determine how to use the pressure reading to alter the pool capacity. There are several subfilters available to use, though again you may create whatever makes the most sense for you pool:

- **BasicRampup** - When pressure exceeds current capacity, increase the number of actors in the pool by some factor (*rampupRate*) of the current pool size.
- **BasicBackoff** - When the pressure ratio falls under some predefined amount (*backoffThreshold*), decrease the number of actors in the pool by some factor of the current pool size.
- **RunningMeanBackoff** - This filter tracks the average pressure-to-capacity over the lifetime of the pool (or since the last time the filter was reset) and will begin to reduce capacity once this mean falls below some predefined amount. The number of actors that will be stopped is determined by some factor of the difference between the current capacity and pressure. The idea behind this filter is to reduce the likelihood of “thrashing” (removing then immediately creating...) pool actors by delaying the backoff until some quiescent stage of the pool. Put another way, use this subfilter to allow quick rampup to handle load and more subtle backoff as that decreases over time.

Examples

```
class TestPool extends Actor with DefaultActorPool
    with BoundedCapacityStrategy
    with ActiveFuturesPressureCapacitor
    with SmallestMailboxSelector
    with BasicNoBackoffFilter
{
  def receive = _route
  def lowerBound = 2
  def upperBound = 4
  def rampupRate = 0.1
  def partialFill = true
  def selectionCount = 1
  def instance = actorOf(new Actor {def receive = {case n:Int =>
    Thread.sleep(n)
    counter.incrementAndGet
    latch.countDown()}}})
}
```

```
class TestPool extends Actor with DefaultActorPool
    with BoundedCapacityStrategy
    with MailboxPressureCapacitor
    with SmallestMailboxSelector
    with Filter
    with RunningMeanBackoff
    with BasicRampup
{
  def receive = _route
  def lowerBound = 1
  def upperBound = 5
  def pressureThreshold = 1
  def partialFill = true
  def selectionCount = 1
  def rampupRate = 0.1
  def backoffRate = 0.50
  def backoffThreshold = 0.50
  def instance = actorOf(new Actor {def receive = {case n:Int =>
    Thread.sleep(n)
    latch.countDown()}}})
}
```

Taken from the unit test [spec](#).

4.14 FSM

Contents

- Overview
- A Simple Example
- State Data
- Reference
 - The FSM Trait and Object
 - Defining Timeouts
 - Defining States
 - Defining the Initial State
 - Unhandled Events
 - Initiating Transitions
 - Monitoring Transitions
 - * Internal Monitoring
 - * External Monitoring
 - Timers
 - Termination from Inside
 - Termination from Outside
- Testing and Debugging Finite State Machines
 - Event Tracing
 - Rolling Event Log
- Examples

Platforms: Scala

Module author: Irmo Manie, Roland Kuhn New in version 1.0.Changed in version 1.2: added Tracing and Logging

Module stability: **STABLE**

4.14.1 Overview

The FSM (Finite State Machine) is available as a mixin for the akka Actor and is best described in the [Erlang design principles](#)

A FSM can be described as a set of relations of the form:

State(S) x Event(E) -> Actions (A), State(S')

These relations are interpreted as meaning:

If we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S'.

4.14.2 A Simple Example

To demonstrate the usage of states we start with a simple FSM without state data. The state can be of any type so for this example we create the states A, B and C.

```
sealed trait ExampleState
case object A extends ExampleState
case object B extends ExampleState
case object C extends ExampleState
```

Now lets create an object representing the FSM and defining the behavior.

```
import akka.actor.{Actor, FSM}
import akka.event.EventHandler
import akka.util.duration._
```

```

case object Move

class ABC extends Actor with FSM[ExampleState, Unit] {

  import FSM._

  startWith(A, Unit)

  when(A) {
    case Ev(Move) =>
      EventHandler.info(this, "Go to B and move on after 5 seconds")
      goto(B) forMax (5 seconds)
  }

  when(B) {
    case Ev(StateTimeout) =>
      EventHandler.info(this, "Moving to C")
      goto(C)
  }

  when(C) {
    case Ev(Move) =>
      EventHandler.info(this, "Stopping")
      stop
  }

  initialize // this checks validity of the initial state and sets up timeout if needed
}

```

Each state is described by one or more `when(state)` blocks; if more than one is given for the same state, they are tried in the order given until the first is found which matches the incoming event. Events are matched using either `Ev(msg)` (if no state data are to be extracted) or `Event(msg, data)`, see below. The statements for each case are the actions to be taken, where the final expression must describe the transition into the next state. This can either be `stay` when no transition is needed or `goto(target)` for changing into the target state. The transition may be annotated with additional properties, where this example includes a state timeout of 5 seconds after the transition into state B: `forMax(duration)` arranges for a `StateTimeout` message to be scheduled, unless some other message is received first. The construction of the FSM is finished by calling the `initialize` method as last part of the `ABC` constructor.

4.14.3 State Data

The FSM can also hold state data associated with the internal state of the state machine. The state data can be of any type but to demonstrate let's look at a lock with a `String` as state data holding the entered unlock code. First we need two states for the lock:

```

sealed trait LockState
case object Locked extends LockState
case object Open extends LockState

```

Now we can create a lock FSM that takes `LockState` as a state and a `String` as state data:

```

import akka.actor.{Actor, FSM}

class Lock(code: String) extends Actor with FSM[LockState, String] {

  import FSM._

  val emptyCode = ""

  startWith(Locked, emptyCode)
}

```

```

when(Locked) {
  // receive a digit and the code that we have so far
  case Event(digit: Char, soFar) => {
    // add the digit to what we have
    soFar + digit match {
      case incomplete if incomplete.length < code.length =>
        // not enough digits yet so stay using the incomplete code as the new state data
        stay using incomplete
      case `code` =>
        // code matched the one from the lock so go to Open state and reset the state data
        goto(Open) using emptyCode forMax (1 seconds)
      case wrong =>
        // wrong code, stay Locked and reset the state data
        stay using emptyCode
    }
  }
}

when(Open) {
  case Ev(StateTimeout, _) => {
    // after the timeout, go back to Locked state
    goto(Locked)
  }
}

initialize
}

```

This very simple example shows how the complete state of the FSM is encoded in the `(State, Data)` pair and only explicitly updated during transitions. This encapsulation is what makes state machines a powerful abstraction, e.g. for handling socket states in a network server application.

4.14.4 Reference

This section describes the DSL in a more formal way, refer to [Examples](#) for more sample material.

The FSM Trait and Object

The `FSM` trait may only be mixed into an `Actor`. Instead of extending `Actor`, the self type approach was chosen in order to make it obvious that an actor is actually created. Importing all members of the `FSM` object is recommended to receive useful implicits and directly access the symbols like `StateTimeout`. This import is usually placed inside the state machine definition:

```

class MyFSM extends Actor with FSM[State, Data] {
  import FSM._

  ...
}

```

The `FSM` trait takes two type parameters:

1. the supertype of all state names, usually a sealed trait with case objects extending it,
2. the type of the state data which are tracked by the `FSM` module itself.

Note: The state data together with the state name describe the internal state of the state machine; if you stick to this scheme and do not add mutable fields to the `FSM` class you have the advantage of making all changes of the internal state explicit in a few well-known places.

Defining Timeouts

The `FSM` module uses `Duration` for all timing configuration. Several methods, like `when` and `startWith` take a `FSM.Timeout`, which is an alias for `Option[Duration]`. There is an implicit conversion available in the `FSM` object which makes this transparent, just import it into your FSM body.

Defining States

A state is defined by one or more invocations of the method

```
when(<name>[, stateTimeout = <timeout>])(stateFunction).
```

The given name must be an object which is type-compatible with the first type parameter given to the `FSM` trait. This object is used as a hash key, so you must ensure that it properly implements `equals` and `hashCode`; in particular it must not be mutable. The easiest fit for these requirements are case objects.

If the `stateTimeout` parameter is given, then all transitions into this state, including staying, receive this timeout by default. Initiating the transition with an explicit timeout may be used to override this default, see [Initiating Transitions](#) for more information. The state timeout of any state may be changed during action processing with `setStateTimeout(state, duration)`. This enables runtime configuration e.g. via external message.

The `stateFunction` argument is a `PartialFunction[Event, State]`, which is conveniently given using the partial function literal syntax as demonstrated below:

```
when(Idle) {
  case Ev(Start(msg)) => // convenience extractor when state data not needed
    goto(Timer) using (msg, self.channel)
}

when(Timer, stateTimeout = 12 seconds) {
  case Event(StateTimeout, (msg, channel)) =>
    channel ! msg
    goto(Idle)
}
```

The `Event(msg, data)` case class may be used directly in the pattern as shown in state `Idle`, or you may use the extractor `Ev(msg)` when the state data are not needed.

Defining the Initial State

Each FSM needs a starting point, which is declared using

```
startWith(state, data[, timeout])
```

The optionally given timeout argument overrides any specification given for the desired initial state. If you want to cancel a default timeout, use `Duration.Inf`.

Unhandled Events

If a state doesn't handle a received event a warning is logged. If you want to do something else in this case you can specify that with `whenUnhandled(stateFunction)`:

```
whenUnhandled {
  case Event(x : X, data) =>
    EventHandler.info(this, "Received unhandled event: " + x)
    stay
  case Ev(msg) =>
    EventHandler.warn(this, "Received unknown event: " + x)
    goto(Error)
}
```

IMPORTANT: This handler is not stacked, meaning that each invocation of `whenUnhandled` replaces the previously installed handler.

Initiating Transitions

The result of any `stateFunction` must be a definition of the next state unless terminating the FSM, which is described in [Termination from Inside](#). The state definition can either be the current state, as described by the `stay` directive, or it is a different state as given by `goto (state)`. The resulting object allows further qualification by way of the modifiers described in the following:

forMax(duration) This modifier sets a state timeout on the next state. This means that a timer is started which upon expiry sends a `StateTimeout` message to the FSM. This timer is canceled upon reception of any other message in the meantime; you can rely on the fact that the `StateTimeout` message will not be processed after an intervening message.

This modifier can also be used to override any default timeout which is specified for the target state. If you want to cancel the default timeout, use `Duration.Inf`.

using(data) This modifier replaces the old state data with the new data given. If you follow the advice [above](#), this is the only place where internal state data are ever modified.

replying(msg) This modifier sends a reply to the currently processed message and otherwise does not modify the state transition.

All modifier can be chained to achieve a nice and concise description:

```
when(State) {
  case Ev(msg) =>
    goto(Processing) using (msg) forMax (5 seconds) replying (WillDo)
}
```

The parentheses are not actually needed in all cases, but they visually distinguish between modifiers and their arguments and therefore make the code even more pleasant to read for foreigners.

Note: Please note that the `return` statement may not be used in `when` blocks or similar; this is a Scala restriction. Either refactor your code using `if () ... else ...` or move it into a method definition.

Monitoring Transitions

Transitions occur “between states” conceptually, which means after any actions you have put into the event handling block; this is obvious since the next state is only defined by the value returned by the event handling logic. You do not need to worry about the exact order with respect to setting the internal state variable, as everything within the FSM actor is running single-threaded anyway.

Internal Monitoring

Up to this point, the FSM DSL has been centered on states and events. The dual view is to describe it as a series of transitions. This is enabled by the method

```
onTransition(handler)
```

which associates actions with a transition instead of with a state and event. The handler is a partial function which takes a pair of states as input; no resulting state is needed as it is not possible to modify the transition in progress.

```
onTransition {
  case Idle -> Active => setTimer("timeout")
  case Active -> _ => cancelTimer("timeout")
  case x -> Idle => EventHandler.info("entering Idle from "+x)
}
```

The convenience extractor `->` enables decomposition of the pair of states with a clear visual reminder of the transition's direction. As usual in pattern matches, an underscore may be used for irrelevant parts; alternatively you could bind the unconstrained state to a variable, e.g. for logging as shown in the last case.

It is also possible to pass a function object accepting two states to `onTransition`, in case your transition handling logic is implemented as a method:

```
onTransition(handler _)

private def handler(from: State, to: State) {
  ...
}
```

The handlers registered with this method are stacked, so you can intersperse `onTransition` blocks with `when` blocks as suits your design. It should be noted, however, that *all handlers will be invoked for each transition*, not only the first matching one. This is designed specifically so you can put all transition handling for a certain aspect into one place without having to worry about earlier declarations shadowing later ones; the actions are still executed in declaration order, though.

Note: This kind of internal monitoring may be used to structure your FSM according to transitions, so that for example the cancellation of a timer upon leaving a certain state cannot be forgot when adding new target states.

External Monitoring

External actors may be registered to be notified of state transitions by sending a message `SubscribeTransitionCallback(actorRef)`. The named actor will be sent a `CurrentState(self, stateName)` message immediately and will receive `Transition(actorRef, oldState, newState)` messages whenever a new state is reached. External monitors may be unregistered by sending `UnsubscribeTransitionCallback(actorRef)` to the FSM actor.

Registering a not-running listener generates a warning and fails gracefully. Stopping a listener without unregistering will remove the listener from the subscription list upon the next transition.

Timers

Besides state timeouts, FSM manages timers identified by `String` names. You may set a timer using

```
setTimer(name, msg, interval, repeat)
```

where `msg` is the message object which will be sent after the duration `interval` has elapsed. If `repeat` is `true`, then the timer is scheduled at fixed rate given by the `interval` parameter. Timers may be canceled using

```
cancelTimer(name)
```

which is guaranteed to work immediately, meaning that the scheduled message will not be processed after this call even if the timer already fired and queued it. The status of any timer may be inquired with

```
timerActive_?(name)
```

These named timers complement state timeouts because they are not affected by intervening reception of other messages.

Termination from Inside

The FSM is stopped by specifying the result state as

```
stop([reason[, data]])
```

The reason must be one of `Normal` (which is the default), `Shutdown` or `Failure(reason)`, and the second argument may be given to change the state data which is available during termination handling.

Note: It should be noted that `stop` does not abort the actions and stop the FSM immediately. The stop action must be returned from the event handler in the same way as a state transition (but note that the `return` statement may not be used within a `when` block).

```
when(A) {
  case Ev(Stop) =>
    doCleanup()
    stop()
}
```

You can use `onTermination(handler)` to specify custom code that is executed when the FSM is stopped. The handler is a partial function which takes a `StopEvent(reason, stateName, stateData)` as argument:

```
onTermination {
  case StopEvent(Normal, s, d)      => ...
  case StopEvent(Shutdown, _, _)    => ...
  case StopEvent(Failure(cause), s, d) => ...
}
```

As for the `whenUnhandled` case, this handler is not stacked, so each invocation of `onTermination` replaces the previously installed handler.

Termination from Outside

When an `ActorRef` associated to a FSM is stopped using the `stop` method, its `postStop` hook will be executed. The default implementation by the `FSM` trait is to execute the `onTermination` handler if that is prepared to handle a `StopEvent(Shutdown, ...)`.

Warning: In case you override `postStop` and want to have your `onTermination` handler called, do not forget to call `super.postStop`.

4.14.5 Testing and Debugging Finite State Machines

During development and for trouble shooting FSMs need care just as any other actor. There are specialized tools available as described in *Testing Finite State Machines* and in the following.

Event Tracing

The setting `akka.actor.debug.fsm` in `akka.conf` enables logging of an event trace by `LoggingFSM` instances:

```
class MyFSM extends Actor with LoggingFSM[X, Z] {
  ...
}
```

This FSM will log at `DEBUG` level:

- all processed events, including `StateTimeout` and scheduled timer messages
- every setting and cancellation of named timers
- all state transitions

Life cycle changes and special messages can be logged as described for *Actors*.

Rolling Event Log

The `LoggingFSM` trait adds one more feature to the FSM: a rolling event log which may be used during debugging (for tracing how the FSM entered a certain failure state) or for other creative uses:

```
class MyFSM extends Actor with LoggingFSM[X, Z] {
  override def logDepth = 12
  onTermination {
    case StopEvent(Failure(_), state, data) =>
      EventHandler.warning(this, "Failure in state "+state+" with data "+data+"\n"+
        "Events leading up to this point:\n\t"+getLog.mkString("\n\t"))
  }
  ...
}
```

The `logDepth` defaults to zero, which turns off the event log.

Warning: The log buffer is allocated during actor creation, which is why the configuration is done using a virtual method call. If you want to override with a `val`, make sure that its initialization happens before the initializer of `LoggingFSM` runs, and do not change the value returned by `logDepth` after the buffer has been allocated.

The contents of the event log are available using method `getLog`, which returns an `IndexedSeq[LogEntry]` where the oldest entry is at index zero.

4.14.6 Examples

A bigger FSM example contrasted with Actor's `become/unbecome` can be found in the sources:

- [Dining Hakkers using FSM](#)
- [Dining Hakkers using become](#)

4.15 HTTP

Contents

- When using Akkas embedded servlet container
- Boot configuration class
- When deploying in another servlet container:
- Adapting your own Akka Initializer for the Servlet Container
- Java API: Typed Actors
- Scala API: Actors
- Using Akka with the Pinky REST/MVC framework
- jetty-run in SBT
- Mist - Lightweight Asynchronous HTTP
 - Endpoints
 - Preparations
 - An Example
 - * Startup
 - * URI Handling
 - * Plumbing
 - * Handling requests
 - Another Example - multiplexing handlers
 - Examples
 - * Using the Akka Mist module with OAuth
 - * Using the Akka Mist module with the Facebook Graph API and WebGL

Module stability: **SOLID**

4.15.1 When using Akkas embedded servlet container

Akka supports the JSR for REST called JAX-RS (JSR-311). It allows you to create interaction with your actors through HTTP + REST

You can deploy your REST services directly into the Akka kernel. All you have to do is to drop the JAR with your application containing the REST services into the '\$AKKA_HOME/deploy' directory and specify in your akka.conf what resource packages to scan for (more on that below) and optionally define a "boot class" (if you need to create any actors or do any config). WAR deployment is coming soon.

4.15.2 Boot configuration class

The boot class is needed for Akka to bootstrap the application and should contain the initial supervisor configuration of any actors in the module.

The boot class should be a regular POJO with a default constructor in which the initial configuration is done. The boot class then needs to be defined in the '\$AKKA_HOME/config/akka.conf' config file like this:

```
akka {
  boot = ["sample.java.Boot", "sample.scala.Boot"] # FQN to the class doing initial actor
                                                    # supervisor bootstrap, should be defined in
  ...
}
```

After you've placed your service-jar into the \$AKKA_HOME/deploy directory, you'll need to tell Akka where to look for your services, and you do that by specifying what packages you want Akka to scan for services, and that's done in akka.conf in the http-section:

```
akka {
  http {
    ...
    resource-packages = ["com.bar", "com.foo.bar"] # List with all resource packages for your Jersey
```

```
...
}
```

4.15.3 When deploying in another servlet container:

If you deploy Akka in another JEE container, don't forget to create an Akka initialization and cleanup hook:

```
package com.my //<--- your own package
import akka.util.AkkaLoader
import akka.remote.BootableRemoteActorService
import akka.actor.BootableActorLoaderService
import javax.servlet.{ServletContextListener, ServletContextEvent}

/**
 * This class can be added to web.xml mappings as a listener to start and postStop Akka.
 * <web-app>
 * ...
 * <listener>
 *   <listener-class>com.my.Initializer</listener-class>
 * </listener>
 * ...
 * </web-app>
 */
class Initializer extends ServletContextListener {
  lazy val loader = new AkkaLoader
  def contextDestroyed(e: ServletContextEvent): Unit = loader.shutdown
  def contextInitialized(e: ServletContextEvent): Unit =
    loader.boot(true, new BootableActorLoaderService with BootableRemoteActorService) //<--- Impo
//    loader.boot(true, new BootableActorLoaderService {}) // If you don't need akka-remote
}
```

For Java users, it's currently only possible to use `BootableActorLoaderService`, but you'll need to use: `akka.actor.DefaultBootableActorLoaderService`

Then you just declare it in your web.xml:

```
<web-app>
...
  <listener>
    <listener-class>your.package.Initializer</listener-class>
  </listener>
...
</web-app>
```

Also, you need to map the servlet that will handle your Jersey/JAX-RS calls, you use Jersey's `ServletContainer` servlet.

```
<web-app>
...
  <servlet>
    <servlet-name>Akka</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <!-- And you want to configure your services -->
    <init-param>
      <param-name>com.sun.jersey.config.property.resourceConfigClass</param-name>
      <param-value>com.sun.jersey.api.core.PackagesResourceConfig</param-value>
    </init-param>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>your.resource.package.here;and.another.here;and.so.on</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
```

```

    <url-pattern>*</url-pattern>
    <servlet-name>Akka</servlet-name>
  </servlet-mapping>
  ...
</web-app>

```

4.15.4 Adapting your own Akka Initializer for the Servlet Container

If you want to use akka-camel or any other modules that have their own “Bootable”s you’ll need to write your own Initializer, which is `_ultra_` simple, see below for an example on how to include Akka-camel.

```

package com.my //<--- your own package
import akka.remote.BootableRemoteActorService
import akka.actor.BootableActorLoaderService
import akka.camel.CamelService
import javax.servlet.{ServletContextListener, ServletContextEvent}

/**
 * This class can be added to web.xml mappings as a listener to start and postStop Akka.
 * <web-app>
 *   ...
 *   <listener>
 *     <listener-class>com.my.Initializer</listener-class>
 *   </listener>
 *   ...
 * </web-app>
 */
class Initializer extends ServletContextListener {
  lazy val loader = new AkkaLoader
  def contextDestroyed(e: ServletContextEvent): Unit = loader.shutdown
  def contextInitialized(e: ServletContextEvent): Unit =
    loader.boot(true, new BootableActorLoaderService with BootableRemoteActorService with CamelS
}

```

4.15.5 Java API: Typed Actors

Sample module for REST services with Actors in Java

4.15.6 Scala API: Actors

Sample module for REST services with Actors in Scala

4.15.7 Using Akka with the Pinky REST/MVC framework

Pinky has a slick Akka integration. Read more [here](#)

4.15.8 jetty-run in SBT

If you want to use jetty-run in SBT you need to exclude the version of Jetty that is bundled in akka-http:

```

override def ivyXML =
  <dependencies>
    <dependency org="se.scalablesolutions.akka" name="akka-http" rev="AKKA_VERSION_GOES_HERE">
      <exclude module="jetty"/>
    </dependency>
  </dependencies>

```


4.15.9 Mist - Lightweight Asynchronous HTTP

The *Mist* layer was developed to provide a direct connection between the servlet container and Akka actors with the goal of handling the incoming HTTP request as quickly as possible in an asynchronous manner. The motivation came from the simple desire to treat REST calls as completable futures, that is, effectively passing the request along an actor message chain to be resumed at the earliest possible time. The primary constraint was to not block any existing threads and secondarily, not create additional ones. Mist is very simple and works both with Jetty Continuations as well as with Servlet API 3.0 (tested using Jetty-8.0.0.M1). When the servlet handles a request, a message is created typed to represent the method (e.g. Get, Post, etc.), the request is suspended and the message is sent (fire-and-forget) to the *root endpoint* actor. That's it. There are no POJOs required to host the service endpoints and the request is treated as any other. The message can be resumed (completed) using a number of helper methods that set the proper HTTP response status code.

Complete runnable example can be found here: <https://github.com/buka/akka-mist-sample>

Endpoints

Endpoints are actors that handle request messages. Minimally there must be an instance of the *RootEndpoint* and then at least one more (to implement your services).

Preparations

In order to use Mist you have to register the *MistServlet* in *web.xml* or do the analogous for the embedded server if running in Akka Microkernel:

```
<servlet>
  <servlet-name>akkaMistServlet</servlet-name>
  <servlet-class>akka.http.AkkaMistServlet</servlet-class>
  <!-- <async-supported>true</async-supported> Enable this for Servlet 3.0 support -->
</servlet>

<servlet-mapping>
  <servlet-name>akkaMistServlet</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Then you also have to add the following dependencies to your SBT build definition:

```
val jettyWebapp = "org.eclipse.jetty" % "jetty-webapp" % "8.0.0.M2" % "test"
val javaxServlet30 = "org.mortbay.jetty" % "servlet-api" % "3.0.20100224" % "provided"
```

Attention: You have to use SBT 0.7.5.RC0 or higher in order to be able to work with that Jetty version.

An Example

Startup

In this example, we'll use the built-in *RootEndpoint* class and implement our own service from that. Here the services are started in the boot loader and attached to the top level supervisor.

```
class Boot {
  val factory = SupervisorFactory(
    SupervisorConfig(
      OneForOneStrategy(List(classOf[Exception]), 3, 100),
      //
      // in this particular case, just boot the built-in default root endpoint
      //
      Supervise(
        actorOf[RootEndpoint],
```

```

    Permanent) ::
    Supervise(
      actorOf[SimpleAkkaAsyncHttpService],
      Permanent)
    :: Nil))
  factory.newInstance.start
}

```

Defining the Endpoint The service is an actor that mixes in the *Endpoint* trait. Here the dispatcher is taken from the Akka configuration file which allows for custom tuning of these actors, though naturally, any dispatcher can be used.

URI Handling

Rather than use traditional annotations to pair HTTP request and class methods, Mist uses hook and provide functions. This offers a great deal of flexibility in how a given endpoint responds to a URI. A hook function is simply a filter, returning a Boolean to indicate whether or not the endpoint will handle the URI. This can be as simple as a straight match or as fancy as you need. If a hook for a given URI returns true, the matching provide function is called to obtain an actor to which the message can be delivered. Notice in the example below, in one case, the same actor is returned and in the other, a new actor is created and returned. Note that URI hooking is non-exclusive and a message can be delivered to multiple actors (see next example).

Plumbing

Hook and provider functions are attached to a parent endpoint, in this case the root, by sending it the **Endpoint.Attach** message. Finally, bind the *handleHttpRequest* function of the *Endpoint* trait to the actor's *receive* function and we're done.

```

class SimpleAkkaAsyncHttpService extends Actor with Endpoint {
  final val ServiceRoot = "/simple/"
  final val ProvideSameActor = ServiceRoot + "same"
  final val ProvideNewActor = ServiceRoot + "new"

  //
  // use the configurable dispatcher
  //
  self.dispatcher = Endpoint.Dispatcher

  //
  // there are different ways of doing this - in this case, we'll use a single hook function
  // and discriminate in the provider; alternatively we can pair hooks & providers
  //
  def hook(uri: String): Boolean = ((uri == ProvideSameActor) || (uri == ProvideNewActor))
  def provide(uri: String): ActorRef = {
    if (uri == ProvideSameActor) same
    else actorOf[BoringActor].start()
  }

  //
  // this is where you want attach your endpoint hooks
  //
  override def preStart() = {
    //
    // we expect there to be one root and that it's already been started up
    // obviously there are plenty of other ways to obtaining this actor
    // the point is that we need to attach something (for starters anyway)
    // to the root
    //
    val root = Actor.registry.actorsFor(classOf[RootEndpoint]).head
  }
}

```

```

    root ! Endpoint.Attach(hook, provide)
  }

  //
  // since this actor isn't doing anything else (i.e. not handling other messages)
  // just assign the receive func like so...
  // otherwise you could do something like:
  //   def myrecv = {...}
  //   def receive = myrecv orElse _recv
  //
  def receive = handleHttpRequest

  //
  // this will be our "same" actor provided with ProvideSameActor endpoint is hit
  //
  lazy val same = actorOf[BoringActor].start()
}

```

Handling requests

Messages are handled just as any other that are received by your actor. The servlet requests and response are not hidden and can be accessed directly as shown below.

```

/**
 * Define a service handler to respond to some HTTP requests
 */
class BoringActor extends Actor {
  import java.util.Date
  import javax.ws.rs.core.MediaType

  var gets = 0
  var posts = 0
  var lastget: Option[Date] = None
  var lastpost: Option[Date] = None

  def receive = {
    // handle a get request
    case get: Get =>
      // the content type of the response.
      // similar to @Produces annotation
      get.response.setContentType(MediaType.TEXT_HTML)

      //
      // "work"
      //
      gets += 1
      lastget = Some(new Date)

      //
      // respond
      //
      val res = "<p>Gets: "+gets+" Posts: "+posts+"</p><p>Last Get: "+lastget.getOrElse("Never").toString()+"</p>"
      get.OK(res)

    // handle a post request
    case post: Post =>
      // the expected content type of the request
      // similar to @Consumes
      if (post.request.getContentType.startsWith(MediaType.APPLICATION_FORM_URLENCODED)) {
        // the content type of the response.
        // similar to @Produces annotation

```

```

    post.response.setContentType(MediaType.TEXT_HTML)

    // "work"
    posts += 1
    lastpost = Some(new Date)

    // respond
    val res = "<p>Gets: "+gets+" Posts: "+posts+"</p><p>Last Get: "+lastget.getOrElse("Never")
    post.OK(res)
  } else {
    post.UnsupportedMediaType("Content-Type request header missing or incorrect (was '" + post
  }
}

case other: RequestMethod =>
  other.NotAllowed("Invalid method for this endpoint")
}
}

```

Timeouts Messages will expire according to the default timeout (specified in akka.conf). Individual messages can also be updated using the *timeout* method. One thing that may seem unexpected is that when an expired request returns to the caller, it will have a status code of OK (200). Mist will add an HTTP header to such responses to help clients, if applicable. By default, the header will be named “Async-Timeout” with a value of “expired” - both of which are configurable.

Another Example - multiplexing handlers

As noted above, hook functions are non-exclusive. This means multiple actors can handle the same request if desired. In this next example, the hook functions are identical (yes, the same one could have been reused) and new instances of both A and B actors will be created to handle the Post. A third mediator is inserted to coordinate the results of these actions and respond to the caller.

```

package sample.mist

import akka.actor._
import akka.actor.Actor._
import akka.http._

import javax.servlet.http.HttpServletResponse

class InterestingService extends Actor with Endpoint {
  final val ServiceRoot = "/interesting/"
  final val Multi = ServiceRoot + "multi/"
  // use the configurable dispatcher
  self.dispatcher = Endpoint.Dispatcher

  //
  // The "multi" endpoint shows forking off multiple actions per request
  // It is triggered by POSTing to http://localhost:9998/interesting/multi/{foo}
  // Try with/without a header named "Test-Token"
  // Try with/without a form parameter named "Data"
  def hookMultiActionA(uri: String): Boolean = uri startsWith Multi
  def provideMultiActionA(uri: String): ActorRef = actorOf(new ActionAActor(complete)).start()

  def hookMultiActionB(uri: String): Boolean = uri startsWith Multi
  def provideMultiActionB(uri: String): ActorRef = actorOf(new ActionBActor(complete)).start()

  //
  // this is where you want attach your endpoint hooks
  //
  override def preStart() = {

```

```

//
// we expect there to be one root and that it's already been started up
// obviously there are plenty of other ways to obtaining this actor
// the point is that we need to attach something (for starters anyway)
// to the root
//
val root = Actor.registry.actorsFor(classOf[RootEndpoint]).head
root ! Endpoint.Attach(hookMultiActionA, provideMultiActionA)
root ! Endpoint.Attach(hookMultiActionB, provideMultiActionB)
}

//
// since this actor isn't doing anything else (i.e. not handling other messages)
// just assign the receive func like so...
// otherwise you could do something like:
// def myrecv = {...}
// def receive = myrecv orElse handleHttpRequest
//
def receive = handleHttpRequest

//
// this guy completes requests after other actions have occurred
//
lazy val complete = actorOf[ActionCompleteActor].start()
}

class ActionAActor(complete:ActorRef) extends Actor {
  import javax.ws.rs.core.MediaType

  def receive = {
    // handle a post request
    case post: Post =>
      // the expected content type of the request
      // similar to @Consumes
      if (post.request.getContentType startsWith MediaType.APPLICATION_FORM_URLENCODED) {
        // the content type of the response.
        // similar to @Produces annotation
        post.response.setContentType(MediaType.TEXT_HTML)

        // get the resource name
        val name = post.request.getRequestURI.substring("/interesting/multi/".length)
        if (name.length % 2 == 0) post.response.getWriter.write("<p>Action A verified request.</p>")
        else post.response.getWriter.write("<p>Action A could not verify request.</p>")

        // notify the next actor to coordinate the response
        complete ! post
      } else post.UnsupportedMediaType("Content-Type request header missing or incorrect (was '"
    }
  }
}

class ActionBActor(complete:ActorRef) extends Actor {
  import javax.ws.rs.core.MediaType

  def receive = {
    // handle a post request
    case post: Post =>
      // the expected content type of the request
      // similar to @Consumes
      if (post.request.getContentType startsWith MediaType.APPLICATION_FORM_URLENCODED) {
        // pull some headers and form params
        def default(any: Any): String = ""

```

```

val token = post.getHeaderOrElse("Test-Token", default)
val data = post.getParameterOrElse("Data", default)

val (resp, status) = (token, data) match {
  case ("", _) => ("No token provided", HttpServletResponse.SC_FORBIDDEN)
  case (_, "") => ("No data", HttpServletResponse.SC_ACCEPTED)
  case _ => ("Data accepted", HttpServletResponse.SC_OK)
}

// update the response body
post.response.getWriter.write(resp)

// notify the next actor to coordinate the response
complete ! (post, status)
} else post.UnsupportedMediaType("Content-Type request header missing or incorrect (was '"

case other: RequestMethod =>
  other.NotAllowed("Invalid method for this endpoint")
}
}

class ActionCompleteActor extends Actor {
  import collection.mutable.HashMap

  val requests = HashMap.empty[Int, Int]

  def receive = {
    case req: RequestMethod =>
      if (requests contains req.hashCode) complete(req)
      else requests += (req.hashCode -> 0)

    case t: Tuple2[RequestMethod, Int] =>
      if (requests contains t._1.hashCode) complete(t._1)
      else requests += (t._1.hashCode -> t._2)
  }

  def complete(req: RequestMethod) = requests.remove(req.hashCode) match {
    case Some(HttpServletResponse.SC_FORBIDDEN) => req.Forbidden("")
    case Some(HttpServletResponse.SC_ACCEPTED) => req.Accepted("")
    case Some(_) => req.OK("")
    case _ => {}
  }
}

```

Examples

Using the Akka Mist module with OAuth

<https://gist.github.com/759501>

Using the Akka Mist module with the Facebook Graph API and WebGL

Example project using Akka Mist with the Facebook Graph API and WebGL <https://github.com/buka/fbgl1>

4.16 HTTP Security

Contents

- [Setup](#)
- [Security Samples](#)
- [Kerberos/SPNEGO Authentication](#)
- [How to setup kerberos on localhost for Ubuntu](#)

Module stability: **IN PROGRESS**

Akka supports security for access to RESTful Actors through [HTTP Authentication](#). The security is implemented as a jersey ResourceFilter which delegates the actual authentication to an authentication actor.

Akka provides authentication via the following authentication schemes:

- [Basic Authentication](#)
- [Digest Authentication](#)
- [Kerberos SPNEGO Authentication](#)

The authentication is performed by implementations of akka.security.AuthenticationActor.

Akka provides a trait for each authentication scheme:

- BasicAuthenticationActor
- DigestAuthenticationActor
- SpnegoAuthenticationActor

4.16.1 Setup

To secure your RESTful actors you need to perform the following steps:

1. configure the resource filter factory 'akka.security.AkkaSecurityFilterFactory' in the 'akka.conf' like this:

```
akka {
  ...
  rest {
    filters="akka.security.AkkaSecurityFilterFactory"
  }
  ...
}
```

2. Configure an implementation of an authentication actor in 'akka.conf':

```
akka {
  ...
  rest {
    filters= ...
    authenticator = "akka.security.samples.BasicAuthenticationService"
  }
  ...
}
```

3. Start your authentication actor in your 'Boot' class. The security package consists of the following parts:
4. Secure your RESTful actors using class or resource level annotations:
 - @DenyAll
 - @RolesAllowed(listOfRoles)
 - @PermitAll

4.16.2 Security Samples

The akka-samples-security module contains a small sample application with sample implementations for each authentication scheme. You can start the sample app using the jetty plugin: `mvn jetty:run`.

The RESTful actor can then be accessed using your browser of choice under:

- permit access only to users having the “chef” role: <http://localhost:8080//secureticker/chef>
- public access: <http://localhost:8080//secureticker/public>

You can access the secured resource using any user for basic authentication (which is the default authenticator in the sample app).

Digest authentication can be directly enabled in the sample app. Kerberos/SPNEGO authentication is a bit more involved and is described below.

4.16.3 Kerberos/SPNEGO Authentication

Kerberos is a network authentication protocol, (see <http://www.ietf.org/rfc/rfc1510.txt>). It provides strong authentication for client/server applications. In a kerberos enabled environment a user will need to sign on only once. Subsequent authentication to applications is handled transparently by kerberos.

Most prominently the kerberos protocol is used to authenticate users in a windows network. When deploying web applications to a corporate intranet an important feature will be to support the single sign on (SSO), which comes to make the application kerberos aware.

How does it work (at least for REST actors)?

- When accessing a secured resource the server will check the request for the *Authorization* header as with basic or digest authentication.
- If it is not set, the server will respond with a challenge to “Negotiate”. The negotiation is in fact the NEGO part of the SPNEGO specification
- The browser will then try to acquire a so called *service ticket* from a ticket granting service, i.e. the kerberos server
- The browser will send the *service ticket* to the web application encoded in the header value of the *Authorization* header
- The web application must validate the ticket based on a shared secret between the web application and the kerberos server. As a result the web application will know the name of the user

To activate the kerberos/SPNEGO authentication for your REST actor you need to enable the kerberos/SPNEGOauthentication actor in the akka.conf like this:

```
akka {
  ...
  rest {
    filters= ...
    authenticator = "akka.security.samples.SpnegoAuthenticationService"
  }
  ...
}
```

Furthermore you must provide the SpnegoAuthenticator with the following information.

- Service principal name: the name of your web application in the kerberos servers user database. This name is always has the form `HTTP/{server}@{realm}`
- Path to the keytab file: this is a kind of certificate for your web application to acquire tickets from the kerberos server


```
akka {
  ...
  rest {
    filters= ...
    authenticator = "akka.security.samples.SpnegoAuthenticationService"
    kerberos {
      servicePrincipal = "HTTP/{server}@{realm}"
      keyTabLocation   = "URL to keytab"
#      kerberosDebug   = "true"
    }
  }
  ...
}
```

4.16.4 How to setup kerberos on localhost for Ubuntu

This is a short step by step description of howto set up a kerberos server on an ubuntu system.

1. Install the Heimdal Kerberos Server and Client

```
sudo apt-get install heimdal-clients heimdal-clients-x heimdal-kdc krb5-config
...
```

2. Set up your kerberos realm. In this example the realm is of course ... EXAMPLE.COM

```
eckart@dilbert:~$ sudo kadmin -l
kadmin> init EXAMPLE.COM
Realm max ticket life [unlimited]:
Realm max renewable ticket life [unlimited]:
kadmin> quit
```

3. Tell your kerberos clients what your realm is and where to find the kerberos server (aka the Key Distribution Centre or KDC)

Edit the kerberos config file: /etc/krb5.conf and configurethe default realm:

```
[libdefaults]
default_realm = EXAMPLE.COM
```

... where to find the KDC for your realm

```
[realms]
EXAMPLE.COM = {
    kdc = localhost
}
```

... which hostnames or domains map to which realm (a kerberos realm is **not** a DNS domain):

```
[domain_realm]
localhost = EXAMPLE.COM
```

4. Add the principals The user principal:

```
eckart@dilbert:~$ sudo kadmin -l
kadmin> add zaphod
Max ticket life [1 day]:
Max renewable life [1 week]:
Principal expiration time [never]:
Password expiration time [never]:
Attributes []:
zaphod@EXAMPLE.COM's Password:
Verifying - zaphod@EXAMPLE.COM's Password:
kadmin> quit
```

The service principal:

```
eckart@dilbert:~$ sudo kadmin -l
kadmin> add HTTP/localhost@EXAMPLE.COM
Max ticket life [1 day]:
Max renewable life [1 week]:
Principal expiration time [never]:
Password expiration time [never]:
Attributes []:
HTTP/localhost@EXAMPLE.COM's Password:
Verifying - HTTP/localhost@EXAMPLE.COM's Password:
kadmin> quit
```

We can now try to acquire initial tickets for the principals to see if everything worked.

```
eckart@dilbert:~$ kinit zaphod
zaphod@EXAMPLE.COM's Password:
```

If this method returns without error we have a success. We can additionally list the acquired tickets:

```
eckart@dilbert:~$ klist
Credentials cache: FILE:/tmp/krb5cc_1000
Principal: zaphod@EXAMPLE.COM

    Issued                Expires               Principal
Oct 24 21:51:59  Oct 25 06:51:59  krbtgt/EXAMPLE.COM@EXAMPLE.COM
```

This seems correct. To remove the ticket cache simply type `kdestroy`.

5. Create a keytab for your service principal

```
eckart@dilbert:~$ ktutil -k http.keytab add -p HTTP/localhost@EXAMPLE.COM -V 1 -e aes256-cts-hmac-sha1-96
Password:
Verifying - Password:
eckart@dilbert:~$
```

This command will create a keytab file for the service principal named `http.keytab` in the current directory. You can specify other encryption methods than ‘`aes256-cts-hmac-sha1-96`’, but this is the default encryption method for the heimdal client, so there is no additional configuration needed. You can specify other encryption types in the `krb5.conf`.

Note that you might need to install the unlimited strength policy files for java from here: http://java.sun.com/javase/downloads/index_jdk5.jsp to use the aes256 encryption from your application.

Again we can test if the keytab generation worked with the `kinit` command:

```
eckart@dilbert:~$ kinit -t http.keytab HTTP/localhost@EXAMPLE.COM
eckart@dilbert:~$ klist
Credentials cache: FILE:/tmp/krb5cc_1000
Principal: HTTP/localhost@EXAMPLE.COM

    Issued                Expires               Principal
Oct 24 21:59:20  Oct 25 06:59:20  krbtgt/EXAMPLE.COM@EXAMPLE.COM
```

Now point the configuration of the key in ‘`akka.conf`’ to the correct location and set the correct service principal name. The web application should now startup and produce at least a 401 response with a header `WWW-Authenticate = "Negotiate"`. The last step is to configure the browser.

6. Set up Firefox to use Kerberos/SPNEGO This is done by typing `about:config`. Filter the config entries for `network.negotiate-auth` and set the config entries `network.negotiate-auth.delegation-uris` and `network.negotiate-auth.trusted-uris` to `localhost`. and now ...

7. Access the RESTful Actor.

8. Have fun ... but acquire an initial ticket for the user principal first: `kinit zaphod`

4.17 Testing Actor Systems

4.17.1 TestKit Example

Ray Roostenburg's example code from [his blog](#) adapted to work with Akka 1.1.

```
package unit.akka

import org.scalatest.matchers.ShouldMatchers
import org.scalatest.{WordSpec, BeforeAndAfterAll}
import akka.actor.Actor._
import akka.util.duration._
import akka.testkit.TestKit
import java.util.concurrent.TimeUnit
import akka.actor.{ActorRef, Actor}
import util.Random

/**
 * a Test to show some TestKit examples
 */

class TestKitUsageSpec extends WordSpec with BeforeAndAfterAll with ShouldMatchers with TestKit {
  val echoRef = actorOf(new EchoActor).start()
  val forwardRef = actorOf(new ForwardingActor(testActor)).start()
  val filterRef = actorOf(new FilteringActor(testActor)).start()
  val randomHead = Random.nextInt(6)
  val randomTail = Random.nextInt(10)
  val headList = List().padTo(randomHead, "0")
  val tailList = List().padTo(randomTail, "1")
  val seqRef = actorOf(new SequencingActor(testActor, headList, tailList)).start()

  override protected def afterAll(): scala.Unit = {
    stopTestActor
    echoRef.stop()
    forwardRef.stop()
    filterRef.stop()
    seqRef.stop()
  }

  "An EchoActor" should {
    "Respond with the same message it receives" in {
      within(100 millis) {
        echoRef ! "test"
        expectMsg("test")
      }
    }
  }

  "A ForwardingActor" should {
    "Forward a message it receives" in {
      within(100 millis) {
        forwardRef ! "test"
        expectMsg("test")
      }
    }
  }

  "A FilteringActor" should {
    "Filter all messages, except expected messagetypes it receives" in {
      var messages = List[String]()
      within(100 millis) {
        filterRef ! "test"
        expectMsg("test")
        filterRef ! 1
      }
    }
  }
}
```

```

    expectNoMsg
    filterRef ! "some"
    filterRef ! "more"
    filterRef ! 1
    filterRef ! "text"
    filterRef ! 1

    receiveWhile(500 millis) {
      case msg: String => messages = msg :: messages
    }
    messages.length should be(3)
    messages.reverse should be(List("some", "more", "text"))
  }
}

"A SequencingActor" should {
  "receive an interesting message at some point " in {
    within(100 millis) {
      seqRef ! "something"
      ignoreMsg {
        case msg: String => msg != "something"
      }
      expectMsg("something")
      ignoreMsg {
        case msg: String => msg == "1"
      }
      expectNoMsg
    }
  }
}

/**
 * An Actor that echoes everything you send to it
 */
class EchoActor extends Actor {
  def receive = {
    case msg => {
      self.reply(msg)
    }
  }
}

/**
 * An Actor that forwards every message to a next Actor
 */
class ForwardingActor(next: ActorRef) extends Actor {
  def receive = {
    case msg => {
      next ! msg
    }
  }
}

/**
 * An Actor that only forwards certain messages to a next Actor
 */
class FilteringActor(next: ActorRef) extends Actor {
  def receive = {
    case msg: String => {
      next ! msg
    }
    case _ => None
  }
}

```

```

    }
  }

  /**
   * An actor that sends a sequence of messages with a random head list, an interesting value and a
   * The idea is that you would like to test that the interesting value is received and that you can
   */
  class SequencingActor(next: ActorRef, head: List[String], tail: List[String]) extends Actor {
    def receive = {
      case msg => {
        head map (next ! _)
        next ! msg
        tail map (next ! _)
      }
    }
  }
}

```

Contents

- [Unit Testing with TestActorRef](#)
 - [Obtaining a Reference to an Actor](#)
 - [Testing Finite State Machines](#)
 - [Testing the Actor's Behavior](#)
 - [The Way In-Between](#)
 - [Use Cases](#)
- [Integration Testing with TestKit](#)
 - [Overview](#)
 - [Built-In Assertions](#)
 - [Expecting Exceptions](#)
 - [Timing Assertions](#)
 - * [Accounting for Slow Test Systems](#)
 - [Resolving Conflicts with Implicit ActorRef](#)
 - [Using Multiple Probe Actors](#)
 - * [Replying to Messages Received by Probes](#)
 - * [Forwarding Messages Received by Probes](#)
 - * [Caution about Timing Assertions](#)
- [CallingThreadDispatcher](#)
 - [How to use it](#)
 - [How it works](#)
 - [Limitations](#)
 - [Benefits](#)
- [Tracing Actor Invocations](#)

Module author: Roland Kuhn New in version 1.0.Changed in version 1.1: added `TestActorRefChanged` in version 1.2: added `TestFSMRef` As with any piece of software, automated tests are a very important part of the development cycle. The actor model presents a different view on how units of code are delimited and how they interact, which has an influence on how to perform tests.

Akka comes with a dedicated module `akka-testkit` for supporting tests at different levels, which fall into two clearly distinct categories:

- Testing isolated pieces of code without involving the actor model, meaning without multiple threads; this implies completely deterministic behavior concerning the ordering of events and no concurrency concerns and will be called **Unit Testing** in the following.
- Testing (multiple) encapsulated actors including multi-threaded scheduling; this implies non-deterministic order of events but shielding from concurrency concerns by the actor model and will be called **Integration Testing** in the following.

There are of course variations on the granularity of tests in both categories, where unit testing reaches down to white-box tests and integration testing can encompass functional tests of complete actor networks. The important distinction lies in whether concurrency concerns are part of the test or not. The tools offered are described in detail in the following sections.

Note: Be sure to add the module `akka-testkit` to your dependencies.

4.17.2 Unit Testing with `TestActorRef`

Testing the business logic inside `Actor` classes can be divided into two parts: first, each atomic operation must work in isolation, then sequences of incoming events must be processed correctly, even in the presence of some possible variability in the ordering of events. The former is the primary use case for single-threaded unit testing, while the latter can only be verified in integration tests.

Normally, the `ActorRef` shields the underlying `Actor` instance from the outside, the only communications channel is the actor's mailbox. This restriction is an impediment to unit testing, which led to the inception of the `TestActorRef`. This special type of reference is designed specifically for test purposes and allows access to the actor in two ways: either by obtaining a reference to the underlying actor instance, or by invoking or querying the actor's behaviour (`receive`). Each one warrants its own section below.

Obtaining a Reference to an Actor

Having access to the actual `Actor` object allows application of all traditional unit testing techniques on the contained methods. Obtaining a reference is done like this:

```
import akka.testkit.TestActorRef

val actorRef = TestActorRef[MyActor]
val actor = actorRef.underlyingActor
```

Since `TestActorRef` is generic in the actor type it returns the underlying actor with its proper static type. From this point on you may bring any unit testing tool to bear on your actor as usual.

Testing Finite State Machines

If your actor under test is a `FSM`, you may use the special `TestFSMRef` which offers all features of a normal `TestActorRef` and in addition allows access to the internal state:

```
import akka.testkit.TestFSMRef
import akka.util.duration._

val fsm = TestFSMRef(new Actor with FSM[Int, String] {
  startWith(1, "")
  when (1) {
    case Ev("go") => goto(2) using "go"
  }
  when (2) {
    case Ev("back") => goto(1) using "back"
  }
}).start()

assert (fsm.stateName == 1)
assert (fsm.stateData == "")
fsm ! "go" // being a TestActorRef, this runs also on the CallingThreadDispatcher
assert (fsm.stateName == 2)
assert (fsm.stateData == "go")

fsm.setState(stateName = 1)
```

```
assert (fsm.stateName == 1)

assert (fsm.timerActive_?("test") == false)
fsm.setTimer("test", 12, 10 millis, true)
assert (fsm.timerActive_?("test") == true)
fsm.cancelTimer("test")
assert (fsm.timerActive_?("test") == false)
```

Due to a limitation in Scala's type inference, there is only the factory method shown above, so you will probably write code like `TestFSMRef(new MyFSM)` instead of the hypothetical `ActorRef-inspired TestFSMRef[MyFSM]`. All methods shown above directly access the FSM state without any synchronization; this is perfectly alright if the `CallingThreadDispatcher` is used (which is the default for `TestFSMRef`) and no other threads are involved, but it may lead to surprises if you were to actually exercise timer events, because those are executed on the `Scheduler` thread.

Testing the Actor's Behavior

When the dispatcher invokes the processing behavior of an actor on a message, it actually calls `apply` on the current behavior registered for the actor. This starts out with the return value of the declared `receive` method, but it may also be changed using `become` and `unbecome`, both of which have corresponding message equivalents, meaning that the behavior may be changed from the outside. All of this contributes to the overall actor behavior and it does not lend itself to easy testing on the `Actor` itself. Therefore the `TestActorRef` offers a different mode of operation to complement the `Actor` testing: it supports all operations also valid on normal `ActorRef`. Messages sent to the actor are processed synchronously on the current thread and answers may be sent back as usual. This trick is made possible by the `CallingThreadDispatcher` described below; this dispatcher is set implicitly for any actor instantiated into a `TestActorRef`.

```
val actorRef = TestActorRef(new MyActor)
val result = (actorRef ? Say42).as[Int] // hypothetical message stimulating a '42' answer
result must be (Some(42))
```

As the `TestActorRef` is a subclass of `LocalActorRef` with a few special extras, also aspects like linking to a supervisor and restarting work properly, as long as all actors involved use the `CallingThreadDispatcher`. As soon as you add elements which include more sophisticated scheduling you leave the realm of unit testing as you then need to think about proper synchronization again (in most cases the problem of waiting until the desired effect had a chance to happen).

One more special aspect which is overridden for single-threaded tests is the `receiveTimeout`, as including that would entail asynchronous queuing of `ReceiveTimeout` messages, violating the synchronous contract.

Warning: To summarize: `TestActorRef` overwrites two fields: it sets the dispatcher to `CallingThreadDispatcher.global` and it sets the `receiveTimeout` to zero.

The Way In-Between

If you want to test the actor behavior, including hotswapping, but without involving a dispatcher and without having the `TestActorRef` swallow any thrown exceptions, then there is another mode available for you: just use the `TestActorRef` as a partial function, the calls to `isDefinedAt` and `apply` will be forwarded to the underlying actor:

```
val ref = TestActorRef[MyActor]
ref.isDefinedAt('unknown) must be (false)
intercept[IllegalActorStateException] { ref(RequestReply) }
```

Use Cases

You may of course mix and match both modi operandi of `TestActorRef` as suits your test needs:

- one common use case is setting up the actor into a specific internal state before sending the test message
- another is to verify correct internal state transitions after having sent the test message

Feel free to experiment with the possibilities, and if you find useful patterns, don't hesitate to let the Akka forums know about them! Who knows, common operations might even be worked into nice DSLs.

4.17.3 Integration Testing with `TestKit`

When you are reasonably sure that your actor's business logic is correct, the next step is verifying that it works correctly within its intended environment (if the individual actors are simple enough, possibly because they use the `FSM` module, this might also be the first step). The definition of the environment depends of course very much on the problem at hand and the level at which you intend to test, ranging for functional/integration tests to full system tests. The minimal setup consists of the test procedure, which provides the desired stimuli, the actor under test, and an actor receiving replies. Bigger systems replace the actor under test with a network of actors, apply stimuli at varying injection points and arrange results to be sent from different emission points, but the basic principle stays the same in that a single procedure drives the test.

Overview

The `TestKit` trait contains a collection of tools which makes this common task easy:

```
import akka.testkit.TestKit
import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers

class MySpec extends WordSpec with MustMatchers with TestKit {

  "An Echo actor" must {

    "send back messages unchanged" in {

      val echo = Actor.actorOf[EchoActor].start()
      echo ! "hello world"
      expectMsg("hello world")

    }

  }

}
```

The `TestKit` contains an actor named `testActor` which is implicitly used as sender reference when dispatching messages from the test procedure. This enables replies to be received by this internal actor, whose only function is to queue them so that interrogation methods like `expectMsg` can examine them. The `testActor` may also be passed to other actors as usual, usually subscribing it as notification listener. There is a whole set of examination methods, e.g. receiving all consecutive messages matching certain criteria, receiving a whole sequence of fixed messages or classes, receiving nothing for some time, etc.

Note: The test actor shuts itself down by default after 5 seconds (configurable) of inactivity, relieving you of the duty of explicitly managing it.

Built-In Assertions

The abovementioned `expectMsg` is not the only method for formulating assertions concerning received messages. Here is the full list:

- `expectMsg[T](d: Duration, msg: T): T`

The given message object must be received within the specified time; the object will be returned.

- `expectMsgPF[T](d: Duration)(pf: PartialFunction[Any, T]): T`

Within the given time period, a message must be received and the given partial function must be defined for that message; the result from applying the partial function to the received message is returned. The duration may be left unspecified (empty parentheses are required in this case) to use the deadline from the innermost enclosing *within* block instead.

- `expectMsgClass[T](d: Duration, c: Class[T]): T`

An object which is an instance of the given `Class` must be received within the allotted time frame; the object will be returned. Note that this does a conformance check; if you need the class to be equal, have a look at `expectMsgAllClassOf` with a single given class argument.

- `expectMsgType[T: Manifest](d: Duration)`

An object which is an instance of the given type (after erasure) must be received within the allotted time frame; the object will be returned. This method is approximately equivalent to `expectMsgClass(manifest[T].erasure)`.

- `expectMsgAnyOf[T](d: Duration, obj: T*): T`

An object must be received within the given time, and it must be equal (compared with `==`) to at least one of the passed reference objects; the received object will be returned.

- `expectMsgAnyClassOf[T](d: Duration, obj: Class[_ <: T]*): T`

An object must be received within the given time, and it must be an instance of at least one of the supplied `Class` objects; the received object will be returned.

- `expectMsgAllOf[T](d: Duration, obj: T*): Seq[T]`

A number of objects matching the size of the supplied object array must be received within the given time, and for each of the given objects there must exist at least one among the received ones which equals (compared with `==`) it. The full sequence of received objects is returned.

- `expectMsgAllClassOf[T](d: Duration, c: Class[_ <: T]*): Seq[T]`

A number of objects matching the size of the supplied `Class` array must be received within the given time, and for each of the given classes there must exist at least one among the received objects whose class equals (compared with `==`) it (this is *not* a conformance check). The full sequence of received objects is returned.

- `expectMsgAllConformingOf[T](d: Duration, c: Class[_ <: T]*): Seq[T]`

A number of objects matching the size of the supplied `Class` array must be received within the given time, and for each of the given classes there must exist at least one among the received objects which is an instance of this class. The full sequence of received objects is returned.

- `expectNoMsg(d: Duration)`

No message must be received within the given time. This also fails if a message has been received before calling this method which has not been removed from the queue using one of the other methods.

- `receiveN(n: Int, d: Duration): Seq[AnyRef]`

`n` messages must be received within the given time; the received messages are returned.

In addition to message reception assertions there are also methods which help with message flows:

- `receiveOne(d: Duration): AnyRef`

Tries to receive one message for at most the given time interval and returns `null` in case of failure. If the given `Duration` is zero, the call is non-blocking (polling mode).

- `receiveWhile[T](max: Duration, idle: Duration)(pf: PartialFunction[Any, T]): Seq[T]`

Collect messages as long as

- they are matching the given partial function
- the given time interval is not used up
- the next message is received within the idle timeout

All collected messages are returned. The maximum duration defaults to the time remaining in the innermost enclosing *within* block and the idle duration defaults to infinity (thereby disabling the idle timeout feature).

- `awaitCond(p: => Boolean, max: Duration, interval: Duration)`

Poll the given condition every `interval` until it returns `true` or the `max` duration is used up. The interval defaults to 100 ms and the maximum defaults to the time remaining in the innermost enclosing *within* block.

- `ignoreMsg(pf: PartialFunction[AnyRef, Boolean])`

`ignoreNoMsg`

The internal `testActor` contains a partial function for ignoring messages: it will only enqueue messages which do not match the function or for which the function returns `false`. This function can be set and reset using the methods given above; each invocation replaces the previous function, they are not composed.

This feature is useful e.g. when testing a logging system, where you want to ignore regular messages and are only interested in your specific ones.

Expecting Exceptions

One case which is not handled by the `testActor` is if an exception is thrown while processing the message sent to the actor under test. This can be tested by using a `Future` based invocation:

```
// assuming ScalaTest ShouldMatchers

evaluating {
  (someActor ? badOperation).await.get
} should produce [UnhandledMessageException]
```

Timing Assertions

Another important part of functional testing concerns timing: certain events must not happen immediately (like a timer), others need to happen before a deadline. Therefore, all examination methods accept an upper time limit within the positive or negative result must be obtained. Lower time limits need to be checked external to the examination, which is facilitated by a new construct for managing time constraints:

```
within([min, ]max) {
  ...
}
```

The block given to `within` must complete after a *Duration* which is between `min` and `max`, where the former defaults to zero. The deadline calculated by adding the `max` parameter to the block's start time is implicitly available within the block to all examination methods, if you do not specify it, is inherited from the innermost enclosing `within` block.

It should be noted that if the last message-receiving assertion of the block is `expectNoMsg` or `receiveWhile`, the final check of the `within` is skipped in order to avoid false positives due to wake-up latencies. This means that while individual contained assertions still use the maximum time bound, the overall block may take arbitrarily longer in this case.

```
class SomeSpec extends WordSpec with MustMatchers with TestKit {
  "A Worker" must {
    "send timely replies" in {
      val worker = actorOf(...)
      within (500 millis) {
        worker ! "some work"
        expectMsg("some result")
      }
    }
  }
}
```

```

    expectNoMsg          // will block for the rest of the 500ms
    Thread.sleep(1000)    // will NOT make this block fail
  }
}
}
}

```

Note: All times are measured using `System.nanoTime`, meaning that they describe wall time, not CPU time.

Ray Roestenburg has written a great article on using the `TestKit`: http://roestenburg.agilesquad.com/2011/02/unit-testing-akka-actors-with-testkit_12.html. His full example is also available [here](#).

Accounting for Slow Test Systems

The tight timeouts you use during testing on your lightning-fast notebook will invariably lead to spurious test failures on the heavily loaded Jenkins server (or similar). To account for this situation, all maximum durations are internally scaled by a factor taken from `akka.conf`, `akka.test.timefactor`, which defaults to 1.

Resolving Conflicts with Implicit ActorRef

The `TestKit` trait contains an implicit value of type `ActorRef` to enable the magic reply handling. This value is named `self` so that e.g. anonymous actors may be declared within a test class without having to care about the ambiguous implicit issues which would otherwise arise. If you find yourself in a situation where the implicit you need comes from a different trait than `TestKit` and is not named `self`, then use `TestKitLight`, which differs only in not having any implicit members. You would then need to make an implicit available in locally confined scopes which need it, e.g. different test cases. If this cannot be done, you will need to resort to explicitly specifying the sender reference:

```

val actor = actorOf[MyWorker].start()
actor.!(msg)(testActor)

```

Using Multiple Probe Actors

When the actors under test are supposed to send various messages to different destinations, it may be difficult distinguishing the message streams arriving at the `testActor` when using the `TestKit` as a mixin. Another approach is to use it for creation of simple probe actors to be inserted in the message flows. To make this more powerful and convenient, there is a concrete implementation called `TestProbe`. The functionality is best explained using a small example:

```

class MyDoubleEcho extends Actor {
  var dest1 : ActorRef = _
  var dest2 : ActorRef = _
  def receive = {
    case (d1: ActorRef, d2: ActorRef) =>
      dest1 = d1
      dest2 = d2
    case x =>
      dest1 ! x
      dest2 ! x
  }
}

val probe1 = TestProbe()
val probe2 = TestProbe()
val actor = Actor.actorOf[MyDoubleEcho].start()
actor ! (probe1.ref, probe2.ref)
actor ! "hello"

```

```
probe1.expectMsg(50 millis, "hello")
probe2.expectMsg(50 millis, "hello")
```

Probes may also be equipped with custom assertions to make your test code even more concise and clear:

```
case class Update(id : Int, value : String)

val probe = new TestProbe {
  def expectUpdate(x : Int) = {
    expectMsg {
      case Update(id, _) if id == x => true
    }
    reply("ACK")
  }
}
```

You have complete flexibility here in mixing and matching the `TestKit` facilities with your own checks and choosing an intuitive name for it. In real life your code will probably be a bit more complicated than the example given above; just use the power!

Replying to Messages Received by Probes

The probes keep track of the communications channel for replies, if possible, so they can also reply:

```
val probe = TestProbe()
val future = probe.ref ? "hello"
probe.expectMsg(0 millis, "hello") // TestActor runs on CallingThreadDispatcher
probe.reply("world")
assert (future.isCompleted && future.as[String] == "world")
```

Forwarding Messages Received by Probes

Given a destination actor `dest` which in the nominal actor network would receive a message from actor `source`. If you arrange for the message to be sent to a `TestProbe` `probe` instead, you can make assertions concerning volume and timing of the message flow while still keeping the network functioning:

```
val probe = TestProbe()
val source = Actor.actorOf(new Source(probe)).start()
val dest = Actor.actorOf[Destination].start()
source ! "start"
probe.expectMsg("work")
probe.forward(dest)
```

The `dest` actor will receive the same message invocation as if no test probe had intervened.

Caution about Timing Assertions

The behavior of `within` blocks when using test probes might be perceived as counter-intuitive: you need to remember that the nicely scoped deadline as described *above* is local to each probe. Hence, probes do not react to each other's deadlines or to the deadline set in an enclosing `TestKit` instance:

```
class SomeTest extends TestKit {

  val probe = TestProbe()

  within(100 millis) {
    probe.expectMsg("hallo") // Will hang forever!
  }
}
```

This test will hang indefinitely, because the `expectMsg` call does not see any deadline. Currently, the only option is to use `probe.within` in the above code to make it work; later versions may include lexically scoped deadlines using implicit arguments.

4.17.4 CallingThreadDispatcher

The `CallingThreadDispatcher` serves good purposes in unit testing, as described above, but originally it was conceived in order to allow contiguous stack traces to be generated in case of an error. As this special dispatcher runs everything which would normally be queued directly on the current thread, the full history of a message's processing chain is recorded on the call stack, so long as all intervening actors run on this dispatcher.

How to use it

Just set the dispatcher as you normally would, either from within the actor

```
import akka.testkit.CallingThreadDispatcher

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

or from the client code

```
val ref = Actor.actorOf[MyActor]
ref.dispatcher = CallingThreadDispatcher.global
ref.start()
```

As the `CallingThreadDispatcher` does not have any configurable state, you may always use the (lazily) preallocated one as shown in the examples.

How it works

When receiving an invocation, the `CallingThreadDispatcher` checks whether the receiving actor is already active on the current thread. The simplest example for this situation is an actor which sends a message to itself. In this case, processing cannot continue immediately as that would violate the actor model, so the invocation is queued and will be processed when the active invocation on that actor finishes its processing; thus, it will be processed on the calling thread, but simply after the actor finishes its previous work. In the other case, the invocation is simply processed immediately on the current thread. Futures scheduled via this dispatcher are also executed immediately.

This scheme makes the `CallingThreadDispatcher` work like a general purpose dispatcher for any actors which never block on external events.

In the presence of multiple threads it may happen that two invocations of an actor running on this dispatcher happen on two different threads at the same time. In this case, both will be processed directly on their respective threads, where both compete for the actor's lock and the loser has to wait. Thus, the actor model is left intact, but the price is loss of concurrency due to limited scheduling. In a sense this is equivalent to traditional mutex style concurrency.

The other remaining difficulty is correct handling of suspend and resume: when an actor is suspended, subsequent invocations will be queued in thread-local queues (the same ones used for queuing in the normal case). The call to `resume`, however, is done by one specific thread, and all other threads in the system will probably not be executing this specific actor, which leads to the problem that the thread-local queues cannot be emptied by their native threads. Hence, the thread calling `resume` will collect all currently queued invocations from all threads into its own queue and process them.

Limitations

If an actor's behavior blocks on something which would normally be affected by the calling actor after having sent the message, this will obviously dead-lock when using this dispatcher. This is a common scenario in actor tests based on `CountDownLatch` for synchronization:

```
val latch = new CountDownLatch(1)
actor ! startWorkAfter(latch)    // actor will call latch.await() before proceeding
doSomeSetupStuff()
latch.countDown()
```

The example would hang indefinitely within the message processing initiated on the second line and never reach the fourth line, which would unblock it on a normal dispatcher.

Thus, keep in mind that the `CallingThreadDispatcher` is not a general-purpose replacement for the normal dispatchers. On the other hand it may be quite useful to run your actor network on it for testing, because if it runs without dead-locking chances are very high that it will not dead-lock in production.

Warning: The above sentence is unfortunately not a strong guarantee, because your code might directly or indirectly change its behavior when running on a different dispatcher. If you are looking for a tool to help you debug dead-locks, the `CallingThreadDispatcher` may help with certain error scenarios, but keep in mind that it may give false negatives as well as false positives.

Benefits

To summarize, these are the features with the `CallingThreadDispatcher` has to offer:

- Deterministic execution of single-threaded tests while retaining nearly full actor semantics
- Full message processing history leading up to the point of failure in exception stack traces
- Exclusion of certain classes of dead-lock scenarios

4.17.5 Tracing Actor Invocations

The testing facilities described up to this point were aiming at formulating assertions about a system's behavior. If a test fails, it is usually your job to find the cause, fix it and verify the test again. This process is supported by debuggers as well as logging, where the Akka toolkit offers the following options:

- *Logging of exceptions thrown within Actor instances*

This is always on; in contrast to the other logging mechanisms, this logs at `ERROR` level.

- *Logging of message invocations on certain actors*

This is enabled by a setting in `akka.conf` — namely `akka.actor.debug.receive` — which enables the `loggable` statement to be applied to an actor's `receive` function:

```
def receive = Actor.loggable(this) { // 'Actor' unnecessary with import Actor._
  case msg => ...
}
```

The first argument to `loggable` defines the source to be used in the logging events, which should be the current actor.

If the abovementioned setting is not given in `akka.conf`, this method will pass through the given `Receive` function unmodified, meaning that there is no runtime cost unless actually enabled.

The logging feature is coupled to this specific local mark-up because enabling it uniformly on all actors is not usually what you need, and it would lead to endless loops if it were applied to `EventHandler` listeners.

- *Logging of special messages*

Actors handle certain special messages automatically, e.g. `Kill`, `PoisonPill`, etc. Tracing of these message invocations is enabled by the setting `akka.actor.debug.autoreceive`, which enables this on all actors.

- *Logging of the actor lifecycle*

Actor creation, start, restart, link, unlink and stop may be traced by enabling the setting `akka.actor.debug.lifecycle`; this, too, is enabled uniformly on all actors.

All these messages are logged at `DEBUG` level. To summarize, you can enable full logging of actor activities using this configuration fragment:

```
akka {
  event-handler-level = "DEBUG"
  actor {
    debug {
      receive = "true"
      autoreceive = "true"
      lifecycle = "true"
    }
  }
}
```

4.18 Tutorial: write a scalable, fault-tolerant, network chat server and client (Scala)

Contents

- Introduction
- Actors
- Creating Actors
- Sample application
- Creating an Akka SBT project
- Creating messages
- Client: Sending messages
- Session: Receiving messages
- Let it crash: Implementing fault-tolerance
- Supervisor hierarchies
- Chat server: Supervision, Traits and more
- Session management
- Chat message management
- STM and Transactors
- Chat storage: Backed with simple in-memory
- Composing the full Chat Service
- Creating a remote server service
- Sample client chat session
- Sample code
- Run it

4.18.1 Introduction

[Tutorial source code.](#)

Writing correct concurrent, fault-tolerant and scalable applications is too hard. Most of the time it's because we are using the wrong tools and the wrong level of abstraction.

Akka is an attempt to change that.

Akka uses the Actor Model together with Software Transactional Memory to raise the abstraction level and provide a better platform to build correct concurrent and scalable applications.

For fault-tolerance Akka adopts the “Let it crash”, also called “Embrace failure”, model which has been used with great success in the telecom industry to build applications that self-heal, systems that never stop.

Actors also provides the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications.

Akka is Open Source and available under the Apache 2 License.

In this article we will introduce you to Akka and see how we can utilize it to build a highly concurrent, scalable and fault-tolerant network server.

But first let's take a step back and discuss what Actors really are and what they are useful for.

4.18.2 Actors

The **Actor Model** provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management. It makes it easier to write correct concurrent and parallel systems. Actors are really nothing new, they were defined in the 1963 paper by Carl Hewitt and have been popularized by the Erlang language which emerged in the mid 80s. It has been used by for example at Ericsson with great success to build highly concurrent and extremely reliable (99.9999999 % availability - 31 ms/year downtime) telecom systems.

Actors encapsulate state and behavior into a lightweight process/thread. In a sense they are like OO objects but with a major semantic difference; they *do not* share state with any other Actor. Each Actor has its own view of the world and can only have impact on other Actors by sending messages to them. Messages are sent asynchronously and non-blocking in a so-called “fire-and-forget” manner where the Actor sends off a message to some other Actor and then do not wait for a reply but goes off doing other things or are suspended by the runtime. Each Actor has a mailbox (ordered message queue) in which incoming messages are processed one by one. Since all processing is done asynchronously and Actors do not block and consume any resources while waiting for messages, Actors tend to give very good concurrency and scalability characteristics and are excellent for building event-based systems.

4.18.3 Creating Actors

Akka has both a Scala API (*Actors (Scala)*) and a Java API (*Actors (Java)*). In this article we will only look at the Scala API since that is the most expressive one. The article assumes some basic Scala knowledge, but even if you don't know Scala I don't think it will not be too hard to follow along anyway.

Akka has adopted the same style of writing Actors as Erlang in which each Actor has an explicit message handler which does pattern matching to match on the incoming messages.

Actors can be created either by: * Extending the ‘Actor’ class and implementing the ‘receive’ method. * Create an anonymous Actor using one of the ‘actor’ methods.

Here is a little example before we dive into a more interesting one.

```
import akka.actor.Actor

class MyActor extends Actor {
  def receive = {
    case "test" => println("received test")
    case _ =>      println("received unknown message")
  }
}
```



```
val myActor = Actor.actorOf[MyActor]
myActor.start()
```

From this call we get a handle to the ‘Actor’ called ‘ActorRef’, which we can use to interact with the Actor

The ‘actorOf’ factory method can be imported like this:

```
import akka.actor.Actor.actorOf

val a = actorOf[MyActor]
```

From now on we will assume that it is imported like this and can use it directly.

Akka Actors are extremely lightweight. Each Actor consume ~600 bytes, which means that you can create 6.5 million on 4 GB RAM.

Messages are sent using the ‘!’ operator:

```
myActor ! "test"
```

4.18.4 Sample application

We will try to write a simple chat/IM system. It is client-server based and uses remote Actors to implement remote clients. Even if it is not likely that you will ever write a chat system I think that it can be a useful exercise since it uses patterns and idioms found in many other use-cases and domains.

We will use many of the features of Akka along the way. In particular; Actors, fault-tolerance using Actor supervision, remote Actors, Software Transactional Memory (STM) and persistence.

4.18.5 Creating an Akka SBT project

First we need to create an SBT project for our tutorial. You do that by stepping into the directory you want to create your project in and invoking the `sbt` command answering the questions for setting up your project:

```
$ sbt
Project does not exist, create new project? (y/N/s) y
Name: Chat
Organization: Hakkers Inc
Version [1.0]:
Scala version [2.9.1]:
sbt version [0.7.6.RC0]:
```

Add the Akka SBT plugin definition to your SBT project by creating a `Plugins.scala` file in the `project/plugins` directory containing:

```
import sbt._

class Plugins(info: ProjectInfo) extends PluginDefinition(info) {
  val akkaRepo = "Akka Repo" at "http://akka.io/repository"
  val akkaPlugin = "se.scalablesolutions.akka" % "akka-sbt-plugin" % "1.2"
}
```

Create a project definition `project/build/Project.scala` file containing:

```
import sbt._

class ChatProject(info: ProjectInfo) extends DefaultProject(info) with AkkaProject {
  val akkaRepo = "Akka Repo" at "http://akka.io/repository"
  val akkaSTM = akkaModule("stm")
  val akkaRemote = akkaModule("remote")
}
```

Make SBT download the dependencies it needs. That is done by invoking:

```
> reload
> update
```

From the SBT project you can generate files for your IDE:

- [SbtEclipsify](#) to generate Eclipse project. Detailed instructions are available in *Getting Started Tutorial (Scala with Eclipse): First Chapter*.
- [sbt-idea](#) to generate IntelliJ IDEA project.

4.18.6 Creating messages

Let's start by defining the messages that will flow in our system. It is very important that all messages that will be sent around in the system are immutable. The Actor model relies on the simple fact that no state is shared between Actors and the only way to guarantee that is to make sure we don't pass mutable state around as part of the messages.

In Scala we have something called [case classes](#). These make excellent messages since they are both immutable and great to pattern match on.

Let's now start by creating the messages that will flow in our system.

```
sealed trait Event
case class Login(user: String) extends Event
case class Logout(user: String) extends Event
case class GetChatLog(from: String) extends Event
case class ChatLog(log: List[String]) extends Event
case class ChatMessage(from: String, message: String) extends Event
```

As you can see with these messages we can log in and out, send a chat message and ask for and get a reply with all the messages in the chat log so far.

4.18.7 Client: Sending messages

Our client wraps each message send in a function, making it a bit easier to use. Here we assume that we have a reference to the chat service so we can communicate with it by sending messages. Messages are sent with the `!` operator (pronounced "bang"). This sends a message asynchronously and do not wait for a reply.

Sometimes however, there is a need for sequential logic, sending a message and wait for the reply before doing anything else. In Akka we can achieve that using the `?` operator. When sending a message with `?` we get back a [Future](#). A 'Future' is a promise that we will get a result later but with the difference from regular method dispatch that the OS thread we are running on is put to sleep while waiting and that we can set a time-out for how long we wait before bailing out, retrying or doing something else. This waiting is achieved with the `Future.as[T]` method, which returns a [scala.Option](#) which implements the [Null Object pattern](#). It has two subclasses; `None` which means no result and `Some(value)` which means that we got a reply. The 'Option' class has a lot of great methods to work with the case of not getting a defined result. F.e. as you can see below we are using the `getOrElse` method which will try to return the result and if there is no result defined invoke the `"...OrElse"` statement.

```
class ChatClient(val name: String) {
  val chat = Actor.remote.actorFor("chat:service", "localhost", 2552)

  def login                = chat ! Login(name)
  def logout               = chat ! Logout(name)
  def post(message: String) = chat ! ChatMessage(name, name + ": " + message)
  def chatLog               = (chat ? GetChatLog(name)).as[ChatLog]
                           .getOrElse(throw new Exception("Couldn't get the chat log from ChatService"))
}
```

As you can see, we are using the ‘Actor.remote.actorFor’ to lookup the chat server on the remote node. From this call we will get a handle to the remote instance and can use it as it is local.

4.18.8 Session: Receiving messages

Now we are done with the client side and let’s dig into the server code. We start by creating a user session. The session is an Actor and is defined by extending the ‘Actor’ trait. This trait has one abstract method that we have to define; ‘receive’ which implements the message handler for the Actor.

In our example the session has state in the form of a ‘List’ with all the messages sent by the user during the session. It takes two parameters in its constructor; the user name and a reference to an Actor implementing the persistent message storage. For both of the messages it responds to, ‘ChatMessage’ and ‘GetChatLog’, it passes them on to the storage Actor.

If you look closely (in the code below) you will see that when passing on the ‘GetChatLog’ message we are not using ‘!’ but ‘forward’. This is similar to ‘!’ but with the important difference that it passes the original sender reference, in this case to the storage Actor. This means that the storage can use this reference to reply to the original sender (our client) directly.

```
class Session(user: String, storage: ActorRef) extends Actor {
  private val loginTime = System.currentTimeMillis
  private var userLog: List[String] = Nil

  EventHandler.info(this, "New session for user [%s] has been created at [%s]".format(user, loginTime))

  def receive = {
    case msg @ ChatMessage(from, message) =>
      userLog ::= message
      storage ! msg

    case msg @ GetChatLog(_) =>
      storage forward msg
  }
}
```

4.18.9 Let it crash: Implementing fault-tolerance

Akka’s approach to fault-tolerance; the “let it crash” model, is implemented by linking Actors. It is very different to what Java and most non-concurrency oriented languages/frameworks have adopted. It’s a way of dealing with failure that is designed for concurrent and distributed systems.

If we look at concurrency first. Now let’s assume we are using non-linked Actors. Throwing an exception in concurrent code, will just simply blow up the thread that currently executes the Actor. There is no way to find out that things went wrong (apart from see the stack trace in the log). There is nothing you can do about it. Here linked Actors provide a clean way of both getting notification of the error so you know what happened, as well as the Actor that crashed, so you can do something about it.

Linking Actors allow you to create sets of Actors where you can be sure that either:

- All are dead
- All are alive

This is very useful when you have hundreds of thousands of concurrent Actors. Some Actors might have implicit dependencies and together implement a service, computation, user session etc. for these being able to group them is very nice.

Akka encourages non-defensive programming. Don’t try to prevent things from go wrong, because they will, whether you want it or not. Instead; expect failure as a natural state in the life-cycle of your app, crash early and let someone else (that sees the whole picture), deal with it.

Now let's look at distributed Actors. As you probably know, you can't build a fault-tolerant system with just one single node, but you need at least two. Also, you (usually) need to know if one node is down and/or the service you are talking to on the other node is down. Here Actor supervision/linking is a critical tool for not only monitoring the health of remote services, but to actually manage the service, do something about the problem if the Actor or node is down. This could be restarting him on the same node or on another node.

To sum things up, it is a very different way of thinking but a way that is very useful (if not critical) to building fault-tolerant highly concurrent and distributed applications.

4.18.10 Supervisor hierarchies

A supervisor is a regular Actor that is responsible for starting, stopping and monitoring its child Actors. The basic idea of a supervisor is that it should keep its child Actors alive by restarting them when necessary. This makes for a completely different view on how to write fault-tolerant servers. Instead of trying all things possible to prevent an error from happening, this approach embraces failure. It shifts the view to look at errors as something natural and something that will happen and instead of trying to prevent it; embrace it. Just "let it crash" and reset the service to a stable state through restart.

Akka has two different restart strategies; All-For-One and One-For-One.

- OneForOne: Restart only the component that has crashed.
- AllForOne: Restart all the components that the supervisor is managing, including the one that have crashed.

The latter strategy should be used when you have a certain set of components that are coupled in some way that if one is crashing they all need to be reset to a stable state before continuing.

4.18.11 Chat server: Supervision, Traits and more

There are two ways you can define an Actor to be a supervisor; declaratively and dynamically. In this example we use the dynamic approach. There are two things we have to do:

- Define the fault handler by setting the 'faultHandler' member field to the strategy we want.
- Define the exceptions we want to "trap", e.g. which exceptions should be handled according to the fault handling strategy we have defined. This is done by setting the 'trapExit' member field to a 'List' with all exceptions we want to trap.

The last thing we have to do to supervise Actors (in our example the storage Actor) is to 'link' the Actor. Invoking 'link(actor)' will create a link between the Actor passed as argument into 'link' and ourselves. This means that we will now get a notification if the linked Actor is crashing and if the cause of the crash, the exception, matches one of the exceptions in our 'trapExit' list then the crashed Actor is restarted according to the fault handling strategy defined in our 'faultHandler'. We also have the 'unlink(actor)' function which disconnects the linked Actor from the supervisor.

In our example we are using a method called 'spawnLink(actor)' which creates, starts and links the Actor in an atomic operation. The linking and unlinking is done in 'preStart' and 'postStop' callback methods which are invoked by the runtime when the Actor is started and shut down (shutting down is done by invoking 'actor.stop()'). In these methods we initialize our Actor, by starting and linking the storage Actor and clean up after ourselves by shutting down all the user session Actors and the storage Actor.

That is it. Now we have implemented the supervising part of the fault-tolerance for the storage Actor. But before we dive into the 'ChatServer' code there are some more things worth mentioning about its implementation.

It defines an abstract member field holding the 'ChatStorage' implementation the server wants to use. We do not define that in the 'ChatServer' directly since we want to decouple it from the actual storage implementation.

The 'ChatServer' is a 'trait', which is Scala's version of mixins. A mixin can be seen as an interface with an implementation and is a very powerful tool in Object-Oriented design that makes it possible to design the system into small, reusable, highly cohesive, loosely coupled parts that can be composed into larger object and components structures.

I'll try to show you how we can make use of Scala's mixins to decouple the Actor implementation from the business logic of managing the user sessions, routing the chat messages and storing them in the persistent storage. Each of these separate parts of the server logic will be represented by its own trait; giving us four different isolated mixins: 'Actor', 'SessionManagement', 'ChatManagement' and 'ChatStorageFactory'. This will give us a loosely coupled system with high cohesion and reusability. At the end of the article I'll show you how you can compose these mixins into the complete runtime component we like.

```
/**
 * Chat server. Manages sessions and redirects all other messages to the Session for the client.
 */
trait ChatServer extends Actor {
  self.faultHandler = OneForOneStrategy(List(classOf[Exception]), 5, 5000)
  val storage: ActorRef

  EventHandler.info(this, "Chat server is starting up...")

  // actor message handler
  def receive: Receive = sessionManagement orElse chatManagement

  // abstract methods to be defined somewhere else
  protected def chatManagement: Receive
  protected def sessionManagement: Receive
  protected def shutdownSessions(): Unit

  override def postStop() = {
    EventHandler.info(this, "Chat server is shutting down...")
    shutdownSessions
    self.unlink(storage)
    storage.stop()
  }
}
```

If you look at the 'receive' message handler function you can see that we have defined it but instead of adding our logic there we are delegating to two different functions; 'sessionManagement' and 'chatManagement', chaining them with 'orElse'. These two functions are defined as abstract in our 'ChatServer' which means that they have to be provided by some other mixin or class when we instantiate our 'ChatServer'. Naturally we will put the 'sessionManagement' implementation in the 'SessionManagement' trait and the 'chatManagement' implementation in the 'ChatManagement' trait. First let's create the 'SessionManagement' trait.

Chaining partial functions like this is a great way of composing functionality in Actors. You can for example put define one default message handler for generic messages in the base Actor and then let deriving Actors extend that functionality by defining additional message handlers. There is a section on how that is done here.

4.18.12 Session management

The session management is defined in the 'SessionManagement' trait in which we implement the two abstract methods in the 'ChatServer'; 'sessionManagement' and 'shutdownSessions'.

The 'SessionManagement' trait holds a 'HashMap' with all the session Actors mapped by user name as well as a reference to the storage (to be able to pass it in to each newly created 'Session').

The 'sessionManagement' function performs session management by responding to the 'Login' and 'Logout' messages. For each 'Login' message it creates a new 'Session' Actor, starts it and puts it in the 'sessions' Map and for each 'Logout' message it does the opposite; shuts down the user's session and removes it from the 'sessions' Map.

The 'shutdownSessions' function simply shuts all the sessions Actors down. That completes the user session management.

```
/**
 * Implements user session management.
 */
```

```

* Uses self-type annotation (this: Actor =>) to declare that it needs to be mixed in with an Actor
*/
trait SessionManagement { this: Actor =>

  val storage: ActorRef // needs someone to provide the ChatStorage
  val sessions = new HashMap[String, ActorRef]

  protected def sessionManagement: Receive = {
    case Login(username) =>
      EventHandler.info(this, "User [%s] has logged in".format(username))
      val session = actorOf(new Session(username, storage))
      session.start()
      sessions += (username -> session)

    case Logout(username) =>
      EventHandler.info(this, "User [%s] has logged out".format(username))
      val session = sessions(username)
      session.stop()
      sessions -= username
  }

  protected def shutdownSessions =
    sessions.foreach { case (_, session) => session.stop() }
}

```

4.18.13 Chat message management

Chat message management is implemented by the ‘ChatManagement’ trait. It has an abstract ‘HashMap’ session member field with all the sessions. Since it is abstract it needs to be mixed in with someone that can provide this reference. If this dependency is not resolved when composing the final component, you will get a compilation error.

It implements the ‘chatManagement’ function which responds to two different messages; ‘ChatMessage’ and ‘GetChatLog’. It simply gets the session for the user (the sender of the message) and routes the message to this session. Here we also use the ‘forward’ function to make sure the original sender reference is passed along to allow the end receiver to reply back directly.

```

/**
 * Implements chat management, e.g. chat message dispatch.
 * <p/>
 * Uses self-type annotation (this: Actor =>) to declare that it needs to be mixed in with an Actor
 */
trait ChatManagement { this: Actor =>
  val sessions: HashMap[String, ActorRef] // needs someone to provide the Session map

  protected def chatManagement: Receive = {
    case msg @ ChatMessage(from, _) => getSession(from).foreach(_ ! msg)
    case msg @ GetChatLog(from) => getSession(from).foreach(_ forward msg)
  }

  private def getSession(from: String) : Option[ActorRef] = {
    if (sessions.contains(from))
      Some(sessions(from))
    else {
      EventHandler.info(this, "Session expired for %s".format(from))
      None
    }
  }
}

```

Using an Actor as a message broker, as in this example, is a very common pattern with many variations; load-

balancing, master/worker, map/reduce, replication, logging etc. It becomes even more useful with remote Actors when we can use it to route messages to different nodes.

4.18.14 STM and Transactors

Actors are excellent for solving problems where you have many independent processes that can work in isolation and only interact with other Actors through message passing. This model fits many problems. But the Actor model is unfortunately a terrible model for implementing truly shared state. E.g. when you need to have consensus and a stable view of state across many components. The classic example is the bank account where clients can deposit and withdraw, in which each operation needs to be atomic. For detailed discussion on the topic see this [presentation](#).

Software Transactional Memory (STM) on the other hand is excellent for problems where you need consensus and a stable view of the state by providing compositional transactional shared state. Some of the really nice traits of STM are that transactions compose and that it raises the abstraction level from lock-based concurrency.

Akka has a STM implementation that is based on the same ideas as found in the [Clojure language](#); Managed References working with immutable data.

Akka allows you to combine Actors and STM into what we call Transactors (short for Transactional Actors), these allow you to optionally combine Actors and STM provides IMHO the best of the Actor model (simple concurrency and asynchronous event-based programming) and STM (compositional transactional shared state) by providing transactional, compositional, asynchronous, event-based message flows. You don't need Transactors all the time but when you do need them then you *really need* them.

Akka currently provides three different transactional abstractions; 'Map', 'Vector' and 'Ref'. They can be shared between multiple Actors and they are managed by the STM. You are not allowed to modify them outside a transaction, if you do so, an exception will be thrown.

What you get is transactional memory in which multiple Actors are allowed to read and write to the same memory concurrently and if there is a clash between two transactions then both of them are aborted and retried. Aborting a transaction means that the memory is rolled back to the state it were in when the transaction was started.

In database terms STM gives you 'ACI' semantics; 'Atomicity', 'Consistency' and 'Isolation'. The 'D' in 'ACID'; 'Durability', you can't get with an STM since it is in memory. It is possible to implement durable persistence for the transactional data structures, but in this sample we keep them in memory.

4.18.15 Chat storage: Backed with simple in-memory

To keep it simple we implement the persistent storage, with a in-memory Vector, i.e. it will not be persistent. We start by creating a 'ChatStorage' trait allowing us to have multiple different storage backend. For example one in-memory and one persistent.

```
/**
 * Abstraction of chat storage holding the chat log.
 */
trait ChatStorage extends Actor
```

Our 'MemoryChatStorage' extends the 'ChatStorage' trait. The only state it holds is the 'chatLog' which is a transactional 'Vector'.

It responds to two different messages; 'ChatMessage' and 'GetChatLog'. The 'ChatMessage' message handler takes the 'message' attribute and appends it to the 'chatLog' vector. Here you can see that we are using the 'atomic { ... }' block to run the vector operation in a transaction. For this in-memory storage it is not important to use a transactional Vector, since it is not shared between actors, but it illustrates the concept.

The 'GetChatLog' message handler retrieves all the messages in the chat log storage inside an atomic block, iterates over them using the 'map' combinator transforming them from 'Array[Byte]' to 'String'. Then it invokes the 'reply(message)' function that will send the chat log to the original sender; the 'ChatClient'.

You might remember that the ‘ChatServer’ was supervising the ‘ChatStorage’ actor. When we discussed that we showed you the supervising Actor’s view. Now is the time for the supervised Actor’s side of things. First, a supervised Actor need to define a life-cycle in which it declares if it should be seen as a:

- ‘Permanent’: which means that the actor will always be restarted.
- ‘Temporary’: which means that the actor will not be restarted, but it will be shut down through the regular shutdown process so the ‘postStop’ callback function will called.

We define the ‘MemoryChatStorage’ as ‘Permanent’ by setting the ‘lifeCycle’ member field to ‘Permanent’.

The idea with this crash early style of designing your system is that the services should just crash and then they should be restarted and reset into a stable state and continue from there. The definition of “stable state” is domain specific and up to the application developer to define. Akka provides two callback functions; ‘preRestart’ and ‘postRestart’ that are called right *before* and right *after* the Actor is restarted. Both of these functions take a ‘Throwable’, the reason for the crash, as argument. In our case we just need to implement the ‘postRestart’ hook and there re-initialize the ‘chatLog’ member field with a fresh ‘Vector’.

```
/**
 * Memory-backed chat storage implementation.
 */
class MemoryChatStorage extends ChatStorage {
  self.lifeCycle = Permanent

  private var chatLog = TransactionalVector[Array[Byte]]()

  EventHandler.info(this, "Memory-based chat storage is starting up...")

  def receive = {
    case msg @ ChatMessage(from, message) =>
      EventHandler.debug(this, "New chat message [%s]".format(message))
      atomic { chatLog + message.getBytes("UTF-8") }

    case GetChatLog(_) =>
      val messageList = atomic { chatLog.map(bytes => new String(bytes, "UTF-8")).toList }
      self.reply(ChatLog(messageList))
  }

  override def postRestart(reason: Throwable) = chatLog = TransactionalVector()
}
```

The last thing we need to do in terms of persistence is to create a ‘MemoryChatStorageFactory’ that will take care of instantiating and resolving the ‘val storage: ChatStorage’ field in the ‘ChatServer’ with a concrete implementation of our persistence Actor.

```
/**
 * Creates and links a MemoryChatStorage.
 */
trait MemoryChatStorageFactory { this: Actor =>
  val storage = this.self.spawnLink[MemoryChatStorage] // starts and links ChatStorage
}
```

4.18.16 Composing the full Chat Service

We have now created the full functionality for the chat server, all nicely decoupled into isolated and well-defined traits. Now let’s bring all these traits together and compose the complete concrete ‘ChatService’.

```
/**
 * Class encapsulating the full Chat Service.
 * Start service by invoking:
 * <pre>
 * val chatService = Actor.actorOf[ChatService].start()
 * </pre>
```



```

*/
class ChatService extends
  ChatServer with
  SessionManagement with
  ChatManagement with
  MemoryChatStorageFactory {
  override def preStart() = {
    remote.start("localhost", 2552);
    remote.register("chat:service", self) //Register the actor with the specified service id
  }
}

```

4.18.17 Creating a remote server service

As you can see in the section above, we are overriding the Actor's 'start' method and are starting up a remote server node by invoking 'remote.start("localhost", 2552)'. This starts up the remote node on address "localhost" and port 2552 which means that it accepts incoming messages on this address. Then we register the ChatService actor in the remote node by invoking 'remote.register("chat:service", self)'. This means that the ChatService will be available to other actors on this specific id, address and port.

That's it. Were done. Now we have a, very simple, but scalable, fault-tolerant, event-driven, persistent chat server that can without problem serve a million concurrent users on a regular workstation.

Let's use it.

4.18.18 Sample client chat session

Now let's create a simple test runner that logs in posts some messages and logs out.

```

/**
 * Test runner emulating a chat session.
 */
object ClientRunner {

  def run = {
    val client1 = new ChatClient("jonas")
    client1.login
    val client2 = new ChatClient("patrik")
    client2.login

    client1.post("Hi there")
    println("CHAT LOG:\n\t" + client1.chatLog.log.mkString("\n\t"))

    client2.post("Hello")
    println("CHAT LOG:\n\t" + client2.chatLog.log.mkString("\n\t"))

    client1.post("Hi again")
    println("CHAT LOG:\n\t" + client1.chatLog.log.mkString("\n\t"))

    client1.logout
    client2.logout
  }
}

```

4.18.19 Sample code

All this code is available as part of the Akka distribution. It resides in the './akka-samples/akka-sample-chat' module and have a 'README' file explaining how to run it.

Or if you rather browse it [online](#).

4.18.20 Run it

Download and build Akka

1. Check out Akka from <http://github.com/jboner/akka>
2. Set 'AKKA_HOME' environment variable to the root of the Akka distribution.
3. Open up a shell and step into the Akka distribution root folder.
4. Build Akka by invoking:

```
% sbt update
% sbt dist
```

Run a sample chat session

1. Fire up two shells. For each of them:
 - Step down into to the root of the Akka distribution.
 - Set 'export AKKA_HOME=<root of distribution>.
 - Run 'sbt console' to start up a REPL (interpreter).
2. In the first REPL you get execute:

```
import sample.chat._
import akka.actor.Actor._
val chatService = actorOf[ChatService].start()
```

3. In the second REPL you get execute:

```
import sample.chat._
ClientRunner.run
```

4. See the chat simulation run.
5. Run it again to see full speed after first initialization.
6. In the client REPL, or in a new REPL, you can also create your own client

```
import sample.chat._
val myClient = new ChatClient("<your name>")
myClient.login
myClient.post("Can I join?")
println("CHAT LOG:\n\t" + myClient.chatLog.log.mkString("\n\t"))
```

That's it. Have fun.

JAVA API

5.1 Actors (Java)

Contents

- Defining an Actor class
 - Creating Actors
 - Creating Actors with non-default constructor
- UntypedActor context
- Identifying Actors
- Messages and immutability
- Send messages
 - Fire-forget
 - Send-And-Receive-Eventually
 - Send-And-Receive-Future
 - Forward message
- Receive messages
- Reply to messages
 - Reply using the channel
 - Reply using the 'replySafe' and 'replyUnsafe' methods
 - Summary of reply semantics and options
- Starting actors
- Stopping actors
- PoisonPill
- Killing an Actor
- Actor life-cycle

Module stability: **SOLID**

The **Actor Model** provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined in the 1973 paper by Carl Hewitt but have been popularized by the Erlang language, and used for example at Ericsson with great success to build highly concurrent and reliable telecom systems.

5.1.1 Defining an Actor class

Actors in Java are created either by extending the 'UntypedActor' class and implementing the 'onReceive' method. This method takes the message as a parameter.

Here is an example:

```
import akka.actor.UntypedActor;
import akka.event.EventHandler;

public class SampleUntypedActor extends UntypedActor {

    public void onReceive(Object message) throws Exception {
        if (message instanceof String)
            EventHandler.info(this, String.format("Received String message: %s",
                message));
        else
            throw new IllegalArgumentException("Unknown message: " + message);
    }
}
```

Creating Actors

Creating an Actor is done using the ‘akka.actor.ActorOf’ factory method. This method returns a reference to the UntypedActor’s ActorRef. This ‘ActorRef’ is an immutable serializable reference that you should use to communicate with the actor, send messages, link to it etc. This reference also functions as the context for the actor and holds run-time type information such as sender of the last message,

```
ActorRef myActor = Actors.actorOf(SampleUntypedActor.class);
myActor.start();
```

Normally you would want to import the ‘actorOf’ method like this:

```
import static akka.actor.ActorOf.*;
ActorRef myActor = actorOf(SampleUntypedActor.class);
```

To avoid prefix it with ‘ActorOf’ every time you use it.

You can also create & start the actor in one statement:

```
ActorRef myActor = actorOf(SampleUntypedActor.class).start();
```

The call to ‘actorOf’ returns an instance of ‘ActorRef’. This is a handle to the ‘UntypedActor’ instance which you can use to interact with the Actor, like send messages to it etc. more on this shortly. The ‘ActorRef’ is immutable and has a one to one relationship with the Actor it represents. The ‘ActorRef’ is also serializable and network-aware. This means that you can serialize it, send it over the wire and use it on a remote host and it will still be representing the same Actor on the original node, across the network.

Creating Actors with non-default constructor

If your UntypedActor has a constructor that takes parameters then you can’t create it using ‘actorOf(clazz)’. Instead you can use a variant of ‘actorOf’ that takes an instance of an ‘UntypedActorFactory’ in which you can create the Actor in any way you like. If you use this method then you to make sure that no one can get a reference to the actor instance. If they can get a reference it then they can touch state directly in bypass the whole actor dispatching mechanism and create race conditions which can lead to corrupt data.

Here is an example:

```
ActorRef actor = actorOf(new UntypedActorFactory() {
    public UntypedActor create() {
        return new MyUntypedActor("service:name", 5);
    }
});
```

This way of creating the Actor is also great for integrating with Dependency Injection (DI) frameworks like Guice or Spring.

5.1.2 UntypedActor context

The `UntypedActor` base class contains almost no member fields or methods to invoke. It only has the `onReceive(Object message)` method, which is defining the Actor's message handler, and some life-cycle callbacks that you can choose to implement: `## preStart ## postStop ## preRestart ## postRestart`

Most of the API is in the `UntypedActorRef` a reference for the actor. This reference is available in the `getContext()` method in the `UntypedActor` (or you can use its alias, the `context()` method, if you prefer. Here, for example, you find methods to reply to messages, send yourself messages, define timeouts, fault tolerance etc., start and stop etc.

5.1.3 Identifying Actors

Each `ActorRef` has two methods: `* getContext().getUuid(); * getContext().getId();`

The difference is that the `'uuid'` is generated by the runtime, guaranteed to be unique and can't be modified. While the `'id'` can be set by the user (using `'getContext().setId(...)'`, and defaults to Actor class name. You can retrieve Actors by both UUID and ID using the `'ActorRegistry'`, see the section further down for details.

5.1.4 Messages and immutability

IMPORTANT: Messages can be any kind of object but have to be immutable. Akka can't enforce immutability (yet) so this has to be by convention.

5.1.5 Send messages

Messages are sent to an Actor through one of the `'send'` methods. `* 'tell'` means "fire-and-forget", e.g. send a message asynchronously and return immediately. `* 'sendRequestReply'` means "send-and-reply-eventually", e.g. send a message asynchronously and wait for a reply through a `Future`. Here you can specify a timeout. Using timeouts is very important. If no timeout is specified then the actor's default timeout (set by the `'getContext().setTimeout(...)'` method in the `'ActorRef'`) is used. This method throws an `'ActorTimeoutException'` if the call timed out. `* 'sendRequestReplyFuture'` sends a message asynchronously and returns a `'Future'`.

In all these methods you have the option of passing along your `'ActorRef'` context variable. Make it a practice of doing so because it will allow the receiver actors to be able to respond to your message, since the sender reference is sent along with the message.

Fire-forget

This is the preferred way of sending messages. No blocking waiting for a message. Give best concurrency and scalability characteristics.

```
actor.tell("Hello");
```

Or with the sender reference passed along:

```
actor.tell("Hello", getContext());
```

If invoked from within an Actor, then the sending actor reference will be implicitly passed along with the message and available to the receiving Actor in its `'getContext().getSender();'` method. He can use this to reply to the original sender or use the `'getContext().reply(message);'` method.

If invoked from an instance that is **not** an Actor there will be no implicit sender passed along the message and you will get an `'IllegalStateException'` if you call `'getContext().reply(...)'`.

Send-And-Receive-Eventually

Using ‘sendRequestReply’ will send a message to the receiving Actor asynchronously but it will wait for a reply on a ‘Future’, blocking the sender Actor until either:

- A reply is received, or
- The Future times out and an ‘ActorTimeoutException’ is thrown.

You can pass an explicit time-out to the ‘sendRequestReply’ method and if none is specified then the default time-out defined in the sender Actor will be used.

Here are some examples:

```
UntypedActorRef actorRef = ...

try {
    Object result = actorRef.sendRequestReply("Hello", getContext(), 1000);
    ... // handle reply
} catch (ActorTimeoutException e) {
    ... // handle timeout
}
```

Send-And-Receive-Future

Using ‘sendRequestReplyFuture’ will send a message to the receiving Actor asynchronously and will immediately return a ‘Future’.

```
Future future = actorRef.sendRequestReplyFuture("Hello", getContext(), 1000);
```

The ‘Future’ interface looks like this:

```
interface Future<T> {
    void await();
    void awaitBlocking();
    boolean isCompleted();
    boolean isExpired();
    long timeoutInNanos();
    Option<T> result();
    Option<Throwable> exception();
    Future<T> onComplete(Procedure<Future<T>> procedure);
}
```

So the normal way of working with futures is something like this:

```
Future future = actorRef.sendRequestReplyFuture("Hello", getContext(), 1000);
future.await();
if (future.isCompleted()) {
    Option resultOption = future.result();
    if (resultOption.isDefined()) {
        Object result = resultOption.get();
        ...
    }
    ... // whatever
}
```

The ‘onComplete’ callback can be used to register a callback to get a notification when the Future completes. Gives you a way to avoid blocking.

Forward message

You can forward a message from one actor to another. This means that the original sender address/reference is maintained even though the message is going through a ‘mediator’. This can be useful when writing actors that

work as routers, load-balancers, replicators etc. You need to pass along your ActorRef context variable as well.

```
getContext().forward(message, getContext());
```

5.1.6 Receive messages

When an actor receives a message it is passed into the ‘onReceive’ method, this is an abstract method on the ‘UntypedActor’ base class that needs to be defined.

Here is an example:

```
public class SampleUntypedActor extends UntypedActor {

    public void onReceive(Object message) throws Exception {
        if (message instanceof String)
            EventHandler.info(this, String.format("Received String message: %s", message));
        else
            throw new IllegalArgumentException("Unknown message: " + message);
    }
}
```

5.1.7 Reply to messages

Reply using the channel

If you want to have a handle to an object to whom you can reply to the message, you can use the Channel abstraction. Simply call getContext().channel() and then you can forward that to others, store it away or otherwise until you want to reply, which you do by Channel.tell(msg)

```
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if (msg.equals("Hello")) {
            // Reply to original sender of message using the channel
            getContext().channel().tryTell(msg + " from " + getContext().getUuid());
        }
    }
}
```

We recommend that you as first choice use the channel abstraction instead of the other ways described in the following sections.

Reply using the ‘replySafe’ and ‘replyUnsafe’ methods

If you want to send a message back to the original sender of the message you just received then you can use the ‘getContext().replyUnsafe(..)’ method.

```
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if (msg.equals("Hello")) {
            // Reply to original sender of message using the ‘replyUnsafe’ method
            getContext().replyUnsafe(msg + " from " + getContext().getUuid());
        }
    }
}
```

In this case we will a reply back to the Actor that sent the message.

The `replyUnsafe` method throws an `IllegalStateException` if unable to determine what to reply to, e.g. the sender has not been passed along with the message when invoking one of `send*` methods. You can also use the more forgiving `replySafe` method which returns `true` if reply was sent, and `false` if unable to determine what to reply to.

```
public void onReceive(Object message) throws Exception {
    if (message instanceof String) {
        String msg = (String)message;
        if (msg.equals("Hello")) {
            // Reply to original sender of message using the 'replyUnsafe' method
            if (getContext().replySafe(msg + " from " + getContext().getUuid())) ... // success
            else ... // handle failure
        }
    }
}
```

Summary of reply semantics and options

- `getContext().reply(...)` can be used to reply to an Actor or a Future from within an actor; the current actor will be passed as reply channel if the current channel supports this.
- `getContext().channel` is a reference providing an abstraction for the reply channel; this reference may be passed to other actors or used by non-actor code.

Note: There used to be two methods for determining the sending Actor or Future for the current invocation:

- `getContext().sender()` yielded a `Option[ActorRef]`
- `getContext().senderFuture()` yielded a `Option[CompletableFuture[Any]]`

These two concepts have been unified into the `channel`. If you need to know the nature of the channel, you may do so using instance tests:

```
if (getContext().channel() instanceof ActorRef) {
    ...
} else if (getContext().channel() instanceof ActorCompletableFuture) {
    ...
}
```

5.1.8 Starting actors

Actors are started by invoking the `start` method.

```
ActorRef actor = actorOf(SampleUntypedActor.class);
myActor.start();
```

You can create and start the Actor in a one liner like this:

```
ActorRef actor = actorOf(SampleUntypedActor.class).start();
```

When you start the actor then it will automatically call the `preStart` callback method on the `UntypedActor`. This is an excellent place to add initialization code for the actor.

```
@Override
void preStart() {
    ... // initialization code
}
```


5.1.9 Stopping actors

Actors are stopped by invoking the ‘stop’ method.

```
actor.stop();
```

When stop is called then a call to the ‘postStop’ callback method will take place. The Actor can use this callback to implement shutdown behavior.

```
@Override
void postStop() {
    ... // clean up resources
}
```

You can shut down all Actors in the system by invoking:

```
Actors.registry().shutdownAll();
```

5.1.10 PoisonPill

You can also send an actor the akka.actor.PoisonPill message, which will stop the actor when the message is processed. If the sender is a Future, the Future will be completed with an akka.actor.ActorKilledException(“PoisonPill”)

Use it like this:

```
import static akka.actor.Actors.*;

actor.tell(poisonPill());
```

5.1.11 Killing an Actor

You can kill an actor by sending a ‘new Kill()’ message. This will restart the actor through regular supervisor semantics.

Use it like this:

```
import static akka.actor.Actors.*;

// kill the actor called 'victim'
victim.tell(kill());
```

5.1.12 Actor life-cycle

The actor has a well-defined non-circular life-cycle.

```
NEW (newly created actor) - can't receive messages (yet)
=> STARTED (when 'start' is invoked) - can receive messages
=> SHUT DOWN (when 'exit' or 'stop' is invoked) - can't do anything
```

5.2 Typed Actors (Java)

Contents

- [Creating Typed Actors](#)
 - [Creating Typed Actors with non-default constructor](#)
 - [Configuration factory class](#)
- [Sending messages](#)
 - [One-way message send](#)
 - [Request-reply message send](#)
 - [Request-reply-with-future message send](#)
- [Stopping Typed Actors](#)
- [How to use the TypedActorContext for runtime information access](#)
- [Messages and immutability](#)

Module stability: **SOLID**

The Typed Actors are implemented through [Typed Actors](#). It uses AOP through [AspectWerkz](#) to turn regular POJOs into asynchronous non-blocking Actors with semantics of the Actor Model. Each method dispatch is turned into a message that is put on a queue to be processed by the Typed Actor sequentially one by one.

If you are using the [Spring Framework](#) then take a look at Akka's Spring integration.

5.2.1 Creating Typed Actors

IMPORTANT: The Typed Actors class must have access modifier 'public' and can't be a non-static inner class.

Akka turns POJOs with interface and implementation into asynchronous (Typed) Actors. Akka is using [AspectWerkz's Proxy](#) implementation, which is the [most performant](#) proxy implementation there exists.

In order to create a Typed Actor you have to subclass the TypedActor base class.

Here is an example.

If you have a POJO with an interface implementation separation like this:

```
interface RegistrationService {
    void register(User user, Credentials cred);
    User getUserFor(String username);
}

import akka.actor.TypedActor;

public class RegistrationServiceImpl extends TypedActor implements RegistrationService {
    public void register(User user, Credentials cred) {
        ... // register user
    }

    public User getUserFor(String username) {
        ... // fetch user by username
        return user;
    }
}
```

Then you can create an Typed Actor out of it by creating it through the 'TypedActor' factory like this:

```
RegistrationService service =
    (RegistrationService) TypedActor.newInstance(RegistrationService.class, RegistrationServiceImpl
// The last parameter defines the timeout for Future calls
```

Creating Typed Actors with non-default constructor

To create a typed actor that takes constructor arguments use a variant of ‘newInstance’ or ‘newRemoteInstance’ that takes an instance of a ‘TypedActorFactory’ in which you can create the TypedActor in any way you like. If you use this method then make sure that no one can get a reference to the actor instance. Touching actor state directly is bypassing the whole actor dispatching mechanism and create race conditions which can lead to corrupt data.

Here is an example:

```
Service service = TypedActor.newInstance(classOf[Service], new TypedActorFactory() {
    public TypedActor create() {
        return new ServiceWithConstructorArgsImpl("someString", 500L);
    }
});
```

Configuration factory class

Using a configuration object:

```
import static java.util.concurrent.TimeUnit.MILLISECONDS;
import akka.actor.TypedActorConfiguration;
import akka.util.FiniteDuration;

TypedActorConfiguration config = new TypedActorConfiguration()
    .timeout(new FiniteDuration(3000, MILLISECONDS));

RegistrationService service = (RegistrationService) TypedActor.newInstance(RegistrationService.class, config);
```

However, often you will not use these factory methods but declaratively define the Typed Actors as part of a supervisor hierarchy. More on that in the [Fault Tolerance Through Supervisor Hierarchies \(Java\)](#) section.

5.2.2 Sending messages

Messages are sent simply by invoking methods on the POJO, which is proxy to the “real” POJO now. The arguments to the method are bundled up atomically into a message and sent to the receiver (the actual POJO instance).

One-way message send

Methods that return void are turned into ‘fire-and-forget’ semantics by asynchronously firing off the message and return immediately. In the example above it would be the ‘register’ method, so if this method is invoked then it returns immediately:

```
// method invocation returns immediately and method is invoke asynchronously using the Actor Model semantics
service.register(user, creds);
```

Request-reply message send

Methods that return something (e.g. non-void methods) are turned into ‘send-and-receive-eventually’ semantics by asynchronously firing off the message and wait on the reply using a Future.

```
// method invocation is asynchronously dispatched using the Actor Model semantics,
// but it blocks waiting on a Future to be resolved in the background
User user = service.getUser(username);
```

Generally it is preferred to use fire-forget messages as much as possible since they will never block, e.g. consume a resource by waiting. But sometimes they are neat to use since they:

Simulates standard Java method dispatch, which is more intuitive for most Java developers # Are a neat to model request-reply # Are useful when you need to do things in a defined order

The same holds for the ‘request-reply-with-future’ described below.

Request-reply-with-future message send

Methods that return a ‘akka.dispatch.Future<TYPE>’ are turned into ‘send-and-receive-with-future’ semantics by asynchronously firing off the message and returns immediately with a Future. You need to use the ‘future(...)’ method in the TypedActor base class to resolve the Future that the client code is waiting on.

Here is an example:

```
public class MathTypedActorImpl extends TypedActor implements MathTypedActor {
    public Future<Integer> square(int value) {
        return future(value * value);
    }
}

MathTypedActor math = TypedActor.actorOf(MathTypedActor.class, MathTypedActorImpl.class);

// This method will return immediately when called, caller should wait on the Future for the result
Future<Integer> future = math.square(10);
future.await();
Integer result = future.get();
```

5.2.3 Stopping Typed Actors

Once Typed Actors have been created with one of the TypedActor.newInstance methods they need to be stopped with TypedActor.stop to free resources allocated by the created Typed Actor (this is not needed when the Typed Actor is supervised).

```
// Create Typed Actor
RegistrationService service = (RegistrationService) TypedActor.newInstance(RegistrationService.class);

// ...

// Free Typed Actor resources
TypedActor.stop(service);
```

When the Typed Actor defines a shutdown callback method (*Fault Tolerance Through Supervisor Hierarchies (Java)*) it will be invoked on TypedActor.stop.

5.2.4 How to use the TypedActorContext for runtime information access

The ‘akka.actor.TypedActorContext’ class Holds ‘runtime type information’ (RTTI) for the Typed Actor. This context is a member field in the TypedActor base class and holds for example the current sender reference, the current sender future etc.

Here is an example how you can use it to in a ‘void’ (e.g. fire-forget) method to implement request-reply by using the sender reference:

```
class PingImpl implements Ping extends TypedActor {

    public void hit(int count) {
        Pong pong = (Pong) getContext().getSender();
        pong.hit(count++);
    }
}
```

If the sender, sender future etc. is not available, then these methods will return ‘null’ so you should have a way of dealing with that scenario.

5.2.5 Messages and immutability

IMPORTANT: Messages can be any kind of object but have to be immutable (there is a workaround, see next section). Java or Scala can’t enforce immutability (yet) so this has to be by convention. Primitives like String, int, Long are always immutable. Apart from these you have to create your own immutable objects to send as messages. If you pass on a reference to an instance that is mutable then this instance can be modified concurrently by two different Typed Actors and the Actor model is broken leaving you with NO guarantees and most likely corrupt data.

Akka can help you in this regard. It allows you to turn on an option for serializing all messages, e.g. all parameters to the Typed Actor effectively making a deep clone/copy of the parameters. This will make sending mutable messages completely safe. This option is turned on in the ‘\$AKKA_HOME/config/akka.conf’ config file like this:

```
akka {
  actor {
    serialize-messages = on # does a deep clone of messages to ensure immutability
  }
}
```

This will make a deep clone (using Java serialization) of all parameters.

5.3 ActorRegistry (Java)

Module stability: **SOLID**

5.3.1 ActorRegistry: Finding Actors

Actors can be looked up using the ‘akka.actor.Actors.registry()’ object. Through this registry you can look up actors by:

- `uuid com.eaio.uuid.UUID` – this uses the `uuid` field in the Actor class, returns the actor reference for the actor with specified uuid, if one exists, otherwise `None`
- `id string` – this uses the `id` field in the Actor class, which can be set by the user (default is the class name), returns all actor references to actors with specified id
- `parameterized type` - returns a `ActorRef[]` with all actors that are a subtype of this specific type
- `specific actor class` - returns a `ActorRef[]` with all actors of this exact class

Actors are automatically registered in the ActorRegistry when they are started and removed when they are stopped. But you can explicitly register and unregister `ActorRef`’s if you need to using the `register` and `unregister` methods.

Here is a summary of the API for finding actors:

```
import static akka.actor.Actors.*;
Option<ActorRef> actor = registry().actorFor(uuid);
ActorRef[] actors = registry().actors();
ActorRef[] otherActors = registry().actorsFor(id);
ActorRef[] moreActors = registry().actorsFor(clazz);
```

You can shut down all Actors in the system by invoking:

```
registry().shutdownAll();
```

If you want to know when a new Actor is added to or removed from the registry, you can use the subscription API on the registry. You can register an Actor that should be notified when an event happens in the ActorRegistry:

```
void addListener(ActorRef listener);
void removeListener(ActorRef listener);
```

The messages sent to this Actor are:

```
public class ActorRegistered {
    ActorRef actor();
}
public class ActorUnregistered {
    ActorRef actor();
}
```

So your listener Actor needs to be able to handle these two messages. Example:

```
import akka.actor.ActorRegistered;
import akka.actor.ActorUnregistered;
import akka.actor.UntypedActor;
import akka.event.EventHandler;

public class RegistryListener extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        if (message instanceof ActorRegistered) {
            ActorRegistered event = (ActorRegistered) message;
            EventHandler.info(this, String.format("Actor registered: %s - %s",
                event.actor().actorClassName(), event.actor().getUuid()));
            event.actor().actorClassName(), event.actor().getUuid());
        } else if (message instanceof ActorUnregistered) {
            // ...
        }
    }
}
```

The above actor can be added as listener of registry events:

```
import static akka.actor.Actors.*;

ActorRef listener = actorOf(RegistryListener.class).start();
registry().addListener(listener);
```

5.4 Futures (Java)

Contents

- Introduction
- Use with Actors
- Use Directly
- Functional Futures
 - Future is a Monad
 - Composing Futures
- Exceptions

5.4.1 Introduction

In Akka, a `Future` is a data structure used to retrieve the result of some concurrent operation. This operation is usually performed by an `Actor` or by the `Dispatcher` directly. This result can be accessed synchronously (blocking) or asynchronously (non-blocking).

5.4.2 Use with Actors

There are generally two ways of getting a reply from an `UntypedActor`: the first is by a sent message (`actorRef.tell(msg);`), which only works if the original sender was an `UntypedActor` and the second is through a `Future`.

Using the `ActorRef`'s `sendRequestReplyFuture` method to send a message will return a `Future`. To wait for and retrieve the actual result the simplest method is:

```
Future[Object] future = actorRef.sendRequestReplyFuture[Object](msg);
Object result = future.get(); //Block until result is available, usually bad practice
```

This will cause the current thread to block and wait for the `UntypedActor` to 'complete' the `Future` with it's reply. Due to the dynamic nature of Akka's `UntypedActors` this result can be anything. The safest way to deal with this is to specify the result to an `Object` as is shown in the above example. You can also use the expected result type instead of `Any`, but if an unexpected type were to be returned you will get a `ClassCastException`. For more elegant ways to deal with this and to use the result without blocking, refer to [Functional Futures](#).

5.4.3 Use Directly

A common use case within Akka is to have some computation performed concurrently without needing the extra utility of an `UntypedActor`. If you find yourself creating a pool of `UntypedActors` for the sole reason of performing a calculation in parallel, there is an easier (and faster) way:

```
import akka.dispatch.Future;
import static akka.dispatch.Futures.future;
import java.util.concurrent.Callable;

Future<String> f = future(new Callable<String>() {
    public String call() {
        return "Hello" + "World!";
    }
});

String result = f.get(); //Blocks until timeout, default timeout is set in akka.conf, otherwise 5
```

In the above code the block passed to `future` will be executed by the default `Dispatcher`, with the return value of the block used to complete the `Future` (in this case, the result would be the string: "HelloWorld"). Unlike a `Future` that is returned from an `UntypedActor`, this `Future` is properly typed, and we also avoid the overhead of managing an `UntypedActor`.

5.4.4 Functional Futures

A recent addition to Akka's `Future` is several monadic methods that are very similar to the ones used by Scala's collections. These allow you to create 'pipelines' or 'streams' that the result will travel through.

Future is a Monad

The first method for working with `Future` functionally is `map`. This method takes a `Function` which performs some operation on the result of the `Future`, and returning a new result. The return value of the `map` method is another `Future` that will contain the new result:

```
import akka.dispatch.Future;
import static akka.dispatch.Futures.future;
import static akka.japi.Function;
import java.util.concurrent.Callable;

Future<String> f1 = future(new Callable<String>() {
    public String call() {
        return "Hello" + "World";
    }
});

Future<Integer> f2 = f1.map(new Function<String, Integer>() {
    public Integer apply(String s) {
        return s.length();
    }
});

Integer result = f2.get();
```

In this example we are joining two strings together within a `Future`. Instead of waiting for `f1` to complete, we apply our function that calculates the length of the string using the `map` method. Now we have a second `Future`, `f2`, that will eventually contain an `Integer`. When our original `Future`, `f1`, completes, it will also apply our function and complete the second `Future` with it's result. When we finally `get` the result, it will contain the number 10. Our original `Future` still contains the string "HelloWorld" and is unaffected by the `map`.

Something to note when using these methods: if the `Future` is still being processed when one of these methods are called, it will be the completing thread that actually does the work. If the `Future` is already complete though, it will be run in our current thread. For example:

```
import akka.dispatch.Future;
import static akka.dispatch.Futures.future;
import static akka.japi.Function;
import java.util.concurrent.Callable;

Future<String> f1 = future(new Callable<String>() {
    public String call() {
        Thread.sleep(1000);
        return "Hello" + "World";
    }
});

Future<Integer> f2 = f1.map(new Function<String, Integer>() {
    public Integer apply(String s) {
        return s.length();
    }
});

Integer result = f2.get();
```

The original `Future` will take at least 1 second to execute now, which means it is still being processed at the time we call `map`. The function we provide gets stored within the `Future` and later executed automatically by the dispatcher when the result is ready.

If we do the opposite:

```
import akka.dispatch.Future;
import static akka.dispatch.Futures.future;
import static akka.japi.Function;
import java.util.concurrent.Callable;

Future<String> f1 = future(new Callable<String>() {
    public String call() {
        return "Hello" + "World";
    }
});
```



```

        }
    });

Thread.sleep(1000);

Future<Integer> f2 = f1.map(new Function<String, Integer>() {
    public Integer apply(String s) {
        return s.length();
    }
});

Integer result = f2.get();

```

Our little string has been processed long before our 1 second sleep has finished. Because of this, the dispatcher has moved onto other messages that need processing and can no longer calculate the length of the string for us, instead it gets calculated in the current thread just as if we weren't using a Future.

Normally this works quite well as it means there is very little overhead to running a quick function. If there is a possibility of the function taking a non-trivial amount of time to process it might be better to have this done concurrently, and for that we use flatMap:

```

import akka.dispatch.Future;
import static akka.dispatch.Futures.future;
import static akka.japi.Function;
import java.util.concurrent.Callable;

Future<String> f1 = future(new Callable<String>() {
    public String call() {
        return "Hello" + "World";
    }
});

Future<Integer> f2 = f1.flatMap(new Function<String, Future<Integer>>() {
    public Future<Integer> apply(final String s) {
        return future(
            new Callable<Integer>() {
                public Integer call() {
                    return s.length();
                }
            }
        );
    }
});

Integer result = f2.get();

```

Now our second Future is executed concurrently as well. This technique can also be used to combine the results of several Futures into a single calculation, which will be better explained in the following sections.

Composing Futures

It is very often desirable to be able to combine different Futures with each other, below are some examples on how that can be done in a non-blocking fashion.

```

import akka.dispatch.Future;
import static akka.dispatch.Futures.sequence;
import akka.japi.Function;
import java.lang.Iterable;

Iterable<Future<Integer>> listOfFutureInts = ... //Some source generating a sequence of Future<Int>

// now we have a Future[Iterable[Int]]
Future<Iterable<Integer>> futureListOfInts = sequence(listOfFutureInts);

```

```
// Find the sum of the odd numbers
Long totalSum = futureListOfInts.map(
    new Function<LinkedList<Integer>, Long>() {
        public Long apply(LinkedList<Integer> ints) {
            long sum = 0;
            for(Integer i : ints)
                sum += i;
            return sum;
        }
    }).get();
```

To better explain what happened in the example, `Future.sequence` is taking the `Iterable<Future<Integer>>` and turning it into a `Future<Iterable<Integer>>`. We can then use `map` to work with the `Iterable<Integer>` directly, and we aggregate the sum of the `Iterable`.

The `traverse` method is similar to `sequence`, but it takes a sequence of `A`'s and applies a function from `A` to `Future` and returns a `Future<Iterable>`, enabling parallel map over the sequence, if you use `Futures.future` to create the `Future`.

```
import akka.dispatch.Future;
import static akka.dispatch.Futures.traverse;
import static akka.dispatch.Futures.future;
import java.lang.Iterable;
import akka.japi.Function;

Iterable<String> listStrings = ... //Just a sequence of Strings

Future<Iterable<String>> result = traverse(listStrings, new Function<String,Future<String>>(){
    public Future<String> apply(final String r) {
        return future(new Callable<String>() {
            public String call() {
                return r.toUpperCase();
            }
        });
    }
});

result.get(); //Returns the sequence of strings as upper case
```

It's as simple as that!

Then there's a method that's called `fold` that takes a start-value, a sequence of `Future`s and a function from the type of the start-value, a timeout, and the type of the futures and returns something with the same type as the start-value, and then applies the function to all elements in the sequence of futures, non-blockingly, the execution will run on the Thread of the last completing `Future` in the sequence.

```
import akka.dispatch.Future;
import static akka.dispatch.Futures.fold;
import java.lang.Iterable;
import akka.japi.Function2;

Iterable<Future<String>> futures = ... //A sequence of Futures, in this case Strings

Future<String> result = fold("", 15000, futures, new Function2<String, String, String>(){ //Start
    public String apply(String r, String t) {
        return r + t; //Just concatenate
    }
});

result.get(); // Will produce a String that says "testtesttesttest"(... and so on).
```

That's all it takes!

If the sequence passed to `fold` is empty, it will return the start-value, in the case above, that will be 0. In some

cases you don't have a start-value and you're able to use the value of the first completing Future in the sequence as the start-value, you can use `reduce`, it works like this:

```
import akka.dispatch.Future;
import static akka.dispatch.Futures.reduce;
import java.util.Iterable;
import akka.japi.Function2;

Iterable<Future<String>> futures = ... //A sequence of Futures, in this case Strings

Future<String> result = reduce(futures, 15000, new Function2<String, String, String>() { //Timeout
    public String apply(String r, String t) {
        return r + t; //Just concatenate
    }
});

result.get(); // Will produce a String that says "testtesttesttest"(... and so on).
```

Same as with `fold`, the execution will be done by the Thread that completes the last of the Futures, you can also parallelize it by chunking your futures into sub-sequences and reduce them, and then reduce the reduced results again.

This is just a sample of what can be done.

5.4.5 Exceptions

Since the result of a `Future` is created concurrently to the rest of the program, exceptions must be handled differently. It doesn't matter if an `UntypedActor` or the dispatcher is completing the `Future`, if an `Exception` is caught the `Future` will contain it instead of a valid result. If a `Future` does contain an `Exception`, calling `get` will cause it to be thrown again so it can be handled properly.

5.5 Dataflow Concurrency (Java)

Contents

- [Introduction](#)
- [Dataflow Variables](#)
- [Threads](#)
- [Examples](#)
 - [Simple DataFlowVariable example](#)
 - [Example on life-cycle management of DataFlowVariables](#)

5.5.1 Introduction

IMPORTANT: As of Akka 1.1, Akka `Future`, `CompletableFuture` and `DefaultCompletableFuture` have all the functionality of `DataFlowVariables`, they also support non-blocking composition and advanced features like `fold` and `reduce`, Akka `DataFlowVariable` is therefor deprecated and will probably resurface in the following release as a DSL on top of Futures.

Akka implements [Oz-style dataflow concurrency](#) through dataflow (single assignment) variables and lightweight (event-based) processes/threads.

Dataflow concurrency is deterministic. This means that it will always behave the same. If you run it once and it yields output 5 then it will do that **every time**, run it 10 million times, same result. If it on the other hand deadlocks the first time you run it, then it will deadlock **every single time** you run it. Also, there is **no difference**

between sequential code and concurrent code. These properties makes it very easy to reason about concurrency. The limitation is that the code needs to be side-effect free, e.g. deterministic. You can't use exceptions, time, random etc., but need to treat the part of your program that uses dataflow concurrency as a pure function with input and output.

The best way to learn how to program with dataflow variables is to read the fantastic book [Concepts, Techniques, and Models of Computer Programming](#). By Peter Van Roy and Seif Haridi.

The documentation is not as complete as it should be, something we will improve shortly. For now, besides above listed resources on dataflow concurrency, I recommend you to read the documentation for the GPars implementation, which is heavily influenced by the Akka implementation:

- <http://gpars.codehaus.org/Dataflow>
- <http://www.gpars.org/guide/guide/7.%20Dataflow%20Concurrency.html>

5.5.2 Dataflow Variables

Dataflow Variable defines three different operations:

1. Define a Dataflow Variable

```
import static akka.dataflow.DataFlow.*;

DataFlowVariable<int> x = new DataFlowVariable<int>();
```

2. Wait for Dataflow Variable to be bound

```
x.get();
```

3. Bind Dataflow Variable

```
x.set(3);
```

A Dataflow Variable can only be bound once. Subsequent attempts to bind the variable will throw an exception.

You can also shutdown a dataflow variable like this:

```
x.shutdown();
```

5.5.3 Threads

You can easily create millions lightweight (event-driven) threads on a regular workstation.

```
import static akka.dataflow.DataFlow.*;
import akka.japi.Effect;

thread(new Effect() {
    public void apply() { ... }
});
```

You can also set the thread to a reference to be able to control its life-cycle:

```
import static akka.dataflow.DataFlow.*;
import akka.japi.Effect;

ActorRef t = thread(new Effect() {
    public void apply() { ... }
});

... // time passes

t.tell(new Exit()); // shut down the thread
```

5.5.4 Examples

Most of these examples are taken from the [Oz wikipedia page](#)

Simple DataFlowVariable example

This example is from Oz wikipedia page: [http://en.wikipedia.org/wiki/Oz_\(programming_language\)](http://en.wikipedia.org/wiki/Oz_(programming_language)). Sort of the “Hello World” of dataflow concurrency.

Example in Oz:

```
thread
  Z = X+Y      % will wait until both X and Y are bound to a value.
  {Browse Z}   % shows the value of Z.
end
thread X = 40 end
thread Y = 2  end
```

Example in Akka:

```
import static akka.dataflow.DataFlow.*;
import akka.japi.Effect;

DataFlowVariable<int> x = new DataFlowVariable<int>();
DataFlowVariable<int> y = new DataFlowVariable<int>();
DataFlowVariable<int> z = new DataFlowVariable<int>();

thread(new Effect() {
  public void apply() {
    z.set(x.get() + y.get());
    System.out.println("z = " + z.get());
  }
});

thread(new Effect() {
  public void apply() {
    x.set(40);
  }
});

thread(new Effect() {
  public void apply() {
    y.set(40);
  }
});
```

Example on life-cycle management of DataFlowVariables

Shows how to shutdown dataflow variables and bind threads to values to be able to interact with them (exit etc.).

Example in Akka:

```
import static akka.dataflow.DataFlow.*;
import akka.japi.Effect;

// create four 'int' data flow variables
DataFlowVariable<int> x = new DataFlowVariable<int>();
DataFlowVariable<int> y = new DataFlowVariable<int>();
DataFlowVariable<int> z = new DataFlowVariable<int>();
DataFlowVariable<int> v = new DataFlowVariable<int>();

ActorRef main = thread(new Effect() {
```

```

public void apply() {
    System.out.println("Thread 'main'")
    if (x.get() > y.get()) {
        z.set(x);
        System.out.println("'z' set to 'x': " + z.get());
    } else {
        z.set(y);
        System.out.println("'z' set to 'y': " + z.get());
    }

    // main completed, shut down the data flow variables
    x.shutdown();
    y.shutdown();
    z.shutdown();
    v.shutdown();
}
});

ActorRef setY = thread(new Effect() {
    public void apply() {
        System.out.println("Thread 'setY', sleeping...");
        Thread.sleep(5000);
        y.set(2);
        System.out.println("'y' set to: " + y.get());
    }
});

ActorRef setV = thread(new Effect() {
    public void apply() {
        System.out.println("Thread 'setV'");
        y.set(2);
        System.out.println("'v' set to y: " + v.get());
    }
});

// shut down the threads
main.tell(new Exit());
setY.tell(new Exit());
setV.tell(new Exit());

```

5.6 Software Transactional Memory (Java)

Contents

- Overview of STM
- Simple example
- Ref
 - Creating a Ref
 - Accessing the value of a Ref
 - Changing the value of a Ref
- Transactions
 - Retries
 - Unexpected retries
 - Coordinated transactions and Transactors
 - Configuring transactions
 - Transaction lifecycle listeners
 - Blocking transactions
 - Alternative blocking transactions
- Transactional datastructures
- Persistent datastructures

Module stability: **SOLID**

5.6.1 Overview of STM

An [STM](#) turns the Java heap into a transactional data set with begin/commit/rollback semantics. Very much like a regular database. It implements the first three letters in **ACID**: **A**CI: * (failure) **A**tomicity: all changes during the execution of a transaction make it, or none make it. This only counts for transactional datastructures. * **C**onsistency: a transaction gets a consistent of reality (in Akka you get the Oracle version of the **SERIALIZED** isolation level). * **I**solated: changes made by concurrent execution transactions are not visible to each other.

Generally, the STM is not needed that often when working with Akka. Some use-cases (that we can think of) are:

- When you really need composable message flows across many actors updating their **internal local** state but need them to do that atomically in one big transaction. Might not often, but when you do need this then you are screwed without it.
- When you want to share a datastructure across actors.
- When you need to use the persistence modules.

Akka's STM implements the concept in [Clojure's STM](#) view on state in general. Please take the time to read [this excellent document](#) and view [this presentation](#) by Rich Hickey (the genius behind Clojure), since it forms the basis of Akka's view on STM and state in general.

The STM is based on Transactional References (referred to as Refs). Refs are memory cells, holding an (arbitrary) immutable value, that implement CAS (Compare-And-Swap) semantics and are managed and enforced by the STM for coordinated changes across many Refs. They are implemented using the excellent [Multiverse STM](#).

Working with immutable collections can sometimes give bad performance due to extensive copying. Scala provides so-called persistent datastructures which makes working with immutable collections fast. They are immutable but with constant time access and modification. The use of structural sharing and an insert or update does not ruin the old structure, hence "persistent". Makes working with immutable composite types fast. The persistent datastructures currently consist of a Map and Vector.

5.6.2 Simple example

Here is a simple example of an incremental counter using STM. This shows creating a `Ref`, a transactional reference, and then modifying it within a transaction, which is delimited by an `Atomic` anonymous inner class.

```
import akka.stm.*;

final Ref<Integer> ref = new Ref<Integer>(0);

public int counter() {
    return new Atomic<Integer>() {
        public Integer atomically() {
            int inc = ref.get() + 1;
            ref.set(inc);
            return inc;
        }
    }.execute();
}

counter();
// -> 1

counter();
// -> 2
```

5.6.3 Ref

Refs (transactional references) are mutable references to values and through the STM allow the safe sharing of mutable data. To ensure safety the value stored in a Ref should be immutable. The value referenced by a Ref can only be accessed or swapped within a transaction. Refs separate identity from value.

Creating a Ref

You can create a Ref with or without an initial value.

```
import akka.stm.*;

// giving an initial value
final Ref<Integer> ref = new Ref<Integer>(0);

// specifying a type but no initial value
final Ref<Integer> ref = new Ref<Integer>();
```

Accessing the value of a Ref

Use `get` to access the value of a Ref. Note that if no initial value has been given then the value is initially `null`.

```
import akka.stm.*;

final Ref<Integer> ref = new Ref<Integer>(0);

Integer value = new Atomic<Integer>() {
    public Integer atomically() {
        return ref.get();
    }
}.execute();
// -> value = 0
```

Changing the value of a Ref

To set a new value for a Ref you can use `set` (or equivalently `swap`), which sets the new value and returns the old value.


```
import akka.stm.*;

final Ref<Integer> ref = new Ref<Integer>(0);

new Atomic() {
    public Object atomically() {
        return ref.set(5);
    }
}.execute();
```

5.6.4 Transactions

A transaction is delimited using an `Atomic` anonymous inner class.

```
new Atomic() {
    public Object atomically() {
        // ...
    }
}.execute();
```

All changes made to transactional objects are isolated from other changes, all make it or non make it (so failure atomicity) and are consistent. With the AkkaSTM you automatically have the Oracle version of the `SERIALIZED` isolation level, lower isolation is not possible. To make it fully serialized, set the `writeskew` property that checks if a writeskew problem is allowed to happen.

Retries

A transaction is automatically retried when it runs into some read or write conflict, until the operation completes, an exception (throwable) is thrown or when there are too many retries. When a read or write conflict is encountered, the transaction uses a bounded exponential backoff to prevent cause more contention and give other transactions some room to complete.

If you are using non transactional resources in an atomic block, there could be problems because a transaction can be retried. If you are using print statements or logging, it could be that they are called more than once. So you need to be prepared to deal with this. One of the possible solutions is to work with a deferred or compensating task that is executed after the transaction aborts or commits.

Unexpected retries

It can happen for the first few executions that you get a few failures of execution that lead to unexpected retries, even though there is not any read or write conflict. The cause of this is that speculative transaction configuration/selection is used. There are transactions optimized for a single transactional object, for 1..n and for n to unlimited. So based on the execution of the transaction, the system learns; it begins with a cheap one and upgrades to more expensive ones. Once it has learned, it will reuse this knowledge. It can be activated/deactivated using the speculative property on the `TransactionFactoryBuilder`. In most cases it is best use the default value (enabled) so you get more out of performance.

Coordinated transactions and Transactors

If you need coordinated transactions across actors or threads then see [Transactors \(Java\)](#).

Configuring transactions

It's possible to configure transactions. The `Atomic` class can take a `TransactionFactory`, which can determine properties of the transaction. A default transaction factory is used if none is specified. You can create a `TransactionFactory` with a `TransactionFactoryBuilder`.

Configuring transactions with a TransactionFactory:

```
import akka.stm.*;

TransactionFactory txFactory = new TransactionFactoryBuilder()
    .setReadonly(true)
    .build();

new Atomic<Object>(txFactory) {
    public Object atomically() {
        // read only transaction
        return ...;
    }
}.execute();
```

The following settings are possible on a TransactionFactory:

- **familyName** - Family name for transactions. Useful for debugging because the familyName is shown in exceptions, logging and in the future also will be used for profiling.
- **readonly** - Sets transaction as readonly. Readonly transactions are cheaper and can be used to prevent modification to transactional objects.
- **maxRetries** - The maximum number of times a transaction will retry.
- **timeout** - The maximum time a transaction will block for.
- **trackReads** - Whether all reads should be tracked. Needed for blocking operations. Readtracking makes a transaction more expensive, but makes subsequent reads cheaper and also lowers the chance of a readconflict.
- **writeSkew** - Whether writeskew is allowed. Disable with care.
- **blockingAllowed** - Whether explicit retries are allowed.
- **interruptible** - Whether a blocking transaction can be interrupted if it is blocked.
- **speculative** - Whether speculative configuration should be enabled.
- **quickRelease** - Whether locks should be released as quickly as possible (before whole commit).
- **propagation** - For controlling how nested transactions behave.
- **traceLevel** - Transaction trace level.

You can also specify the default values for some of these options in akka.conf. Here they are with their default values:

```
stm {
  fair                = on          # Should global transactions be fair or non-fair (non fair yield better
  max-retries         = 1000
  timeout             = 5           # Default timeout for blocking transactions and transaction set (in u
                                     # the time-unit property)
  write-skew          = true
  blocking-allowed    = false
  interruptible       = false
  speculative         = true
  quick-release       = true
  propagation         = "requires"
  trace-level         = "none"
}
```

Transaction lifecycle listeners

It's possible to have code that will only run on the successful commit of a transaction, or when a transaction aborts. You can do this by adding deferred or compensating blocks to a transaction.

```
import akka.stm.*;
import static akka.stm.StmUtils.deferred;
import static akka.stm.StmUtils.compensating;

new Atomic() {
    public Object atomically() {
        deferred(new Runnable() {
            public void run() {
                // executes when transaction commits
            }
        });
        compensating(new Runnable() {
            public void run() {
                // executes when transaction aborts
            }
        });
        // ...
        return something;
    }
}.execute();
```

Blocking transactions

You can block in a transaction until a condition is met by using an explicit `retry`. To use `retry` you also need to configure the transaction to allow explicit retries.

Here is an example of using `retry` to block until an account has enough money for a withdrawal. This is also an example of using actors and STM together.

```
import akka.stm.*;

public class Transfer {
    private final Ref<Double> from;
    private final Ref<Double> to;
    private final double amount;

    public Transfer(Ref<Double> from, Ref<Double> to, double amount) {
        this.from = from;
        this.to = to;
        this.amount = amount;
    }

    public Ref<Double> getFrom() { return from; }
    public Ref<Double> getTo() { return to; }
    public double getAmount() { return amount; }
}
```

```
import akka.stm.*;
import static akka.stm.StmUtils.retry;
import akka.actor.*;
import akka.util.FiniteDuration;
import java.util.concurrent.TimeUnit;
import akka.event.EventHandler;

public class Transferer extends UntypedActor {
    TransactionFactory txFactory = new TransactionFactoryBuilder()
        .setBlockingAllowed(true)
        .setTrackReads(true)
        .setTimeout(new FiniteDuration(60, TimeUnit.SECONDS))
        .build();

    public void onReceive(Object message) throws Exception {
```

```

        if (message instanceof Transfer) {
            Transfer transfer = (Transfer) message;
            final Ref<Double> from = transfer.getFrom();
            final Ref<Double> to = transfer.getTo();
            final double amount = transfer.getAmount();
            new Atomic(txFactory) {
                public Object atomically() {
                    if (from.get() < amount) {
                        EventHandler.info(this, "not enough money - retrying");
                        retry();
                    }
                    EventHandler.info(this, "transferring");
                    from.set(from.get() - amount);
                    to.set(to.get() + amount);
                    return null;
                }
            }.execute();
        }
    }
}

```

```

import akka.stm.*;
import akka.actor.*;

public class Main {
    public static void main(String...args) throws Exception {
        final Ref<Double> account1 = new Ref<Double>(100.0);
        final Ref<Double> account2 = new Ref<Double>(100.0);

        ActorRef transferer = Actors.actorOf(Transferer.class).start();

        transferer.tell(new Transfer(account1, account2, 500.0));
        // Transferer: not enough money - retrying

        new Atomic() {
            public Object atomically() {
                return account1.set(account1.get() + 2000);
            }
        }.execute();
        // Transferer: transferring

        Thread.sleep(1000);

        Double acc1 = new Atomic<Double>() {
            public Double atomically() {
                return account1.get();
            }
        }.execute();

        Double acc2 = new Atomic<Double>() {
            public Double atomically() {
                return account2.get();
            }
        }.execute();

        System.out.println("Account 1: " + acc1);
        // Account 1: 1600.0

        System.out.println("Account 2: " + acc2);
        // Account 2: 600.0
    }
}

```

```

    transferer.stop();
  }
}

```

Alternative blocking transactions

You can also have two alternative blocking transactions, one of which can succeed first, with `EitherOrElse`.

```

import akka.stm.*;

public class Branch {
  private final Ref<Integer> left;
  private final Ref<Integer> right;
  private final double amount;

  public Branch(Ref<Integer> left, Ref<Integer> right, int amount) {
    this.left = left;
    this.right = right;
    this.amount = amount;
  }

  public Ref<Integer> getLeft() { return left; }

  public Ref<Integer> getRight() { return right; }

  public double getAmount() { return amount; }
}

```

```

import akka.actor.*;
import akka.stm.*;
import static akka.stm.StmUtils.retry;
import akka.util.FiniteDuration;
import java.util.concurrent.TimeUnit;
import akka.event.EventHandler;

public class Brancher extends UntypedActor {
  TransactionFactory txFactory = new TransactionFactoryBuilder()
    .setBlockingAllowed(true)
    .setTrackReads(true)
    .setTimeout(new FiniteDuration(60, TimeUnit.SECONDS))
    .build();

  public void onReceive(Object message) throws Exception {
    if (message instanceof Branch) {
      Branch branch = (Branch) message;
      final Ref<Integer> left = branch.getLeft();
      final Ref<Integer> right = branch.getRight();
      final double amount = branch.getAmount();
      new Atomic<Integer>(txFactory) {
        public Integer atomically() {
          return new EitherOrElse<Integer>() {
            public Integer either() {
              if (left.get() < amount) {
                EventHandler.info(this, "not enough on left - retrying");
                retry();
              }
              EventHandler.info(this, "going left");
              return left.get();
            }
            public Integer orElse() {
              if (right.get() < amount) {
                EventHandler.info(this, "not enough on right - retrying");

```

```

        retry();
    }
    EventHandler.info(this, "going right");
    return right.get();
}
}.execute();
}
}.execute();
}
}
}
}
}

```

```

import akka.stm.*;
import akka.actor.*;

public class Main2 {
    public static void main(String...args) throws Exception {
        final Ref<Integer> left = new Ref<Integer>(100);
        final Ref<Integer> right = new Ref<Integer>(100);

        ActorRef brancher = Actors.actorOf(Brancher.class).start();

        brancher.tell(new Branch(left, right, 500));
        // not enough on left - retrying
        // not enough on right - retrying

        Thread.sleep(1000);

        new Atomic() {
            public Object atomically() {
                return right.set(right.get() + 1000);
            }
        }.execute();
        // going right

        brancher.stop();
    }
}

```

5.6.5 Transactional datastructures

Akka provides two datastructures that are managed by the STM.

- TransactionalMap
- TransactionalVector

TransactionalMap and TransactionalVector look like regular mutable datastructures, they even implement the standard Scala ‘Map’ and ‘RandomAccessSeq’ interfaces, but they are implemented using persistent datastructures and managed references under the hood. Therefore they are safe to use in a concurrent environment. Underlying TransactionalMap is HashMap, an immutable Map but with near constant time access and modification operations. Similarly TransactionalVector uses a persistent Vector. See the Persistent Datastructures section below for more details.

Like managed references, TransactionalMap and TransactionalVector can only be modified inside the scope of an STM transaction.

Here is an example of creating and accessing a TransactionalMap:

```
import akka.stm.*;

// assuming a User class

final TransactionalMap<String, User> users = new TransactionalMap<String, User>();

// fill users map (in a transaction)
new Atomic() {
    public Object atomically() {
        users.put("bill", new User("bill"));
        users.put("mary", new User("mary"));
        users.put("john", new User("john"));
        return null;
    }
}.execute();

// access users map (in a transaction)
User user = new Atomic<User>() {
    public User atomically() {
        return users.get("bill").get();
    }
}.execute();
```

Here is an example of creating and accessing a TransactionalVector:

```
import akka.stm.*;

// assuming an Address class

final TransactionalVector<Address> addresses = new TransactionalVector<Address>();

// fill addresses vector (in a transaction)
new Atomic() {
    public Object atomically() {
        addresses.add(new Address("somewhere"));
        addresses.add(new Address("somewhere else"));
        return null;
    }
}.execute();

// access addresses vector (in a transaction)
Address address = new Atomic<Address>() {
    public Address atomically() {
        return addresses.get(0);
    }
}.execute();
```

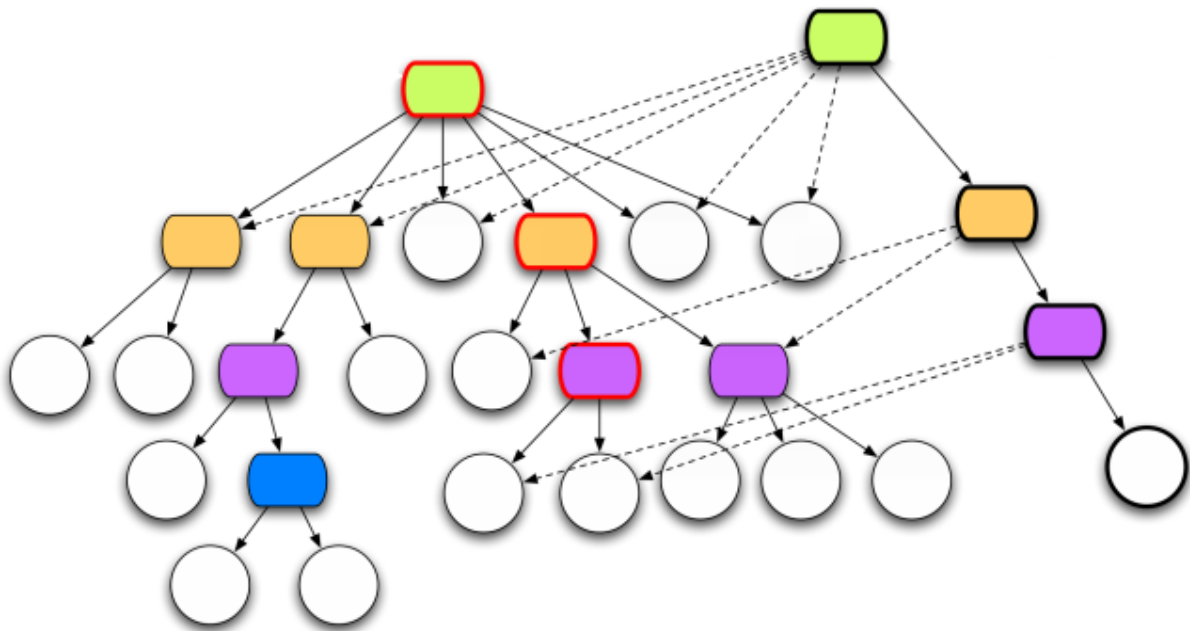
5.6.6 Persistent datastructures

Akka's STM should only be used with immutable data. This can be costly if you have large datastructures and are using a naive copy-on-write. In order to make working with immutable datastructures fast enough Scala provides what are called Persistent Datastructures. There are currently two different ones:

- `HashMap` ([scaladoc](#))
- `Vector` ([scaladoc](#))

They are immutable and each update creates a completely new version but they are using clever structural sharing in order to make them almost as fast, for both read and update, as regular mutable datastructures.

This illustration is taken from Rich Hickey's presentation. Copyright Rich Hickey 2009.



5.7 Transactors (Java)

Contents

- [Why Transactors?](#)
 - [Actors and STM](#)
- [Coordinated transactions](#)
- [UntypedTransactor](#)
- [Coordinating Typed Actors](#)

Module stability: **SOLID**

5.7.1 Why Transactors?

Actors are excellent for solving problems where you have many independent processes that can work in isolation and only interact with other Actors through message passing. This model fits many problems. But the actor model is unfortunately a terrible model for implementing truly shared state. E.g. when you need to have consensus and a stable view of state across many components. The classic example is the bank account where clients can deposit and withdraw, in which each operation needs to be atomic. For detailed discussion on the topic see [this JavaOne presentation](#).

STM on the other hand is excellent for problems where you need consensus and a stable view of the state by providing compositional transactional shared state. Some of the really nice traits of STM are that transactions compose, and it raises the abstraction level from lock-based concurrency.

Akka's Transactors combine Actors and STM to provide the best of the Actor model (concurrency and asynchronous event-based programming) and STM (compositional transactional shared state) by providing transactional, compositional, asynchronous, event-based message flows.

If you need Durability then you should not use one of the in-memory data structures but one of the persistent ones. Generally, the STM is not needed very often when working with Akka. Some use-cases (that we can think of) are:

- When you really need composable message flows across many actors updating their **internal local** state but need them to do that atomically in one big transaction. Might not often, but when you do need this then you are screwed without it.
- When you want to share a datastructure across actors.
- When you need to use the persistence modules.

Actors and STM

You can combine Actors and STM in several ways. An Actor may use STM internally so that particular changes are guaranteed to be atomic. Actors may also share transactional datastructures as the STM provides safe shared state across threads.

It's also possible to coordinate transactions across Actors or threads so that either the transactions in a set all commit successfully or they all fail. This is the focus of Transactors and the explicit support for coordinated transactions in this section.

5.7.2 Coordinated transactions

Akka provides an explicit mechanism for coordinating transactions across actors. Under the hood it uses a `CountDownCommitBarrier`, similar to a `CountDownLatch`.

Here is an example of coordinating two simple counter `UntypedActors` so that they both increment together in coordinated transactions. If one of them was to fail to increment, the other would also fail.

```
import akka.actor.ActorRef;

public class Increment {
    private final ActorRef friend;

    public Increment() {
        this.friend = null;
    }

    public Increment(ActorRef friend) {
        this.friend = friend;
    }

    public boolean hasFriend() {
        return friend != null;
    }

    public ActorRef getFriend() {
        return friend;
    }
}

import akka.actor.UntypedActor;
import akka.stm.Ref;
import akka.transactor.Atomically;
import akka.transactor.Coordinated;

public class Counter extends UntypedActor {
    private Ref<Integer> count = new Ref(0);

    private void increment() {
        count.set(count.get() + 1);
    }

    public void onReceive(Object incoming) throws Exception {
        if (incoming instanceof Coordinated) {
```

```

    Coordinated coordinated = (Coordinated) incoming;
    Object message = coordinated.getMessage();
    if (message instanceof Increment) {
        Increment increment = (Increment) message;
        if (increment.hasFriend()) {
            increment.getFriend().tell(coordinated.coordinate(new Increment()));
        }
        coordinated.atomic(new Atomically() {
            public void atomically() {
                increment();
            }
        });
    }
    else if (incoming.equals("GetCount")) {
        getContext().replyUnsafe(count.get());
    }
}
}

```

```

ActorRef counter1 = actorOf(Counter.class).start();
ActorRef counter2 = actorOf(Counter.class).start();

counter1.tell(new Coordinated(new Increment(counter2)));

```

To start a new coordinated transaction that you will also participate in, just create a `Coordinated` object:

```
Coordinated coordinated = new Coordinated();
```

To start a coordinated transaction that you won't participate in yourself you can create a `Coordinated` object with a message and send it directly to an actor. The recipient of the message will be the first member of the coordination set:

```
actor.tell(new Coordinated(new Message()));
```

To include another actor in the same coordinated transaction that you've created or received, use the `coordinate` method on that object. This will increment the number of parties involved by one and create a new `Coordinated` object to be sent.

```
actor.tell(coordinated.coordinate(new Message()));
```

To enter the coordinated transaction use the `atomic` method of the coordinated object. This accepts either an `akka.transactor.Atomically` object, or an `Atomic` object the same as used normally in the STM (just don't execute it - the coordination will do that).

```

coordinated.atomic(new Atomically() {
    public void atomically() {
        // do something in a transaction
    }
});

```

The coordinated transaction will wait for the other transactions before committing. If any of the coordinated transactions fail then they all fail.

5.7.3 UntypedTransactor

UntypedTransactors are untyped actors that provide a general pattern for coordinating transactions, using the explicit coordination described above.

Here's an example of a simple untyped transactor that will join a coordinated transaction:

```

import akka.transactor.UntypedTransactor;
import akka.stm.Ref;

```

```
public class Counter extends UntypedTransactor {
    Ref<Integer> count = new Ref<Integer>(0);

    @Override
    public void atomically(Object message) {
        if (message instanceof Increment) {
            count.set(count.get() + 1);
        }
    }
}
```

You could send this Counter transactor a `Coordinated(Increment)` message. If you were to send it just an `Increment` message it will create its own `Coordinated` (but in this particular case wouldn't be coordinating transactions with any other transactors).

To coordinate with other transactors override the `coordinate` method. The `coordinate` method maps a message to a set of `SendTo` objects, pairs of `ActorRef` and a message. You can use the `include` and `sendTo` methods to easily coordinate with other transactors.

Example of coordinating an increment, similar to the explicitly coordinated example:

```
import akka.transactor.UntypedTransactor;
import akka.transactor.SendTo;
import akka.stm.Ref;

import java.util.Set;

public class Counter extends UntypedTransactor {
    Ref<Integer> count = new Ref<Integer>(0);

    @Override
    public Set<SendTo> coordinate(Object message) {
        if (message instanceof Increment) {
            Increment increment = (Increment) message;
            if (increment.hasFriend())
                return include(increment.getFriend(), new Increment());
        }
        return nobody();
    }

    @Override
    public void atomically(Object message) {
        if (message instanceof Increment) {
            count.set(count.get() + 1);
        }
    }
}
```

To execute directly before or after the coordinated transaction, override the `before` and `after` methods. They do not execute within the transaction.

To completely bypass coordinated transactions override the `normally` method. Any message matched by `normally` will not be matched by the other methods, and will not be involved in coordinated transactions. In this method you can implement normal actor behavior, or use the normal STM atomic for local transactions.

5.7.4 Coordinating Typed Actors

It's also possible to use coordinated transactions with typed actors. You can explicitly pass around `Coordinated` objects, or use built-in support with the `@Coordinated` annotation and the `Coordination.coordinate` method.

To specify a method should use coordinated transactions add the `@Coordinated` annotation. **Note:** the `@Coordinated` annotation will only work with void (one-way) methods.

```
public interface Counter {
    @Coordinated public void increment();
    public Integer get();
}
```

To coordinate transactions use a coordinate block. This accepts either an `akka.transactor.Atomically` object, or an `Atomic` object liked used in the STM (but don't execute it). The first boolean parameter specifies whether or not to wait for the transactions to complete.

```
Coordination.coordinate(true, new Atomically() {
    public void atomically() {
        counter1.increment();
        counter2.increment();
    }
});
```

Here's an example of using `@Coordinated` with a `TypedActor` to coordinate increments:

```
import akka.transactor.annotation.Coordinated;

public interface Counter {
    @Coordinated public void increment();
    public Integer get();
}
```

```
import akka.actor.TypedActor;
import akka.stm.Ref;

public class CounterImpl extends TypedActor implements Counter {
    private Ref<Integer> count = new Ref<Integer>(0);

    public void increment() {
        count.set(count.get() + 1);
    }

    public Integer get() {
        return count.get();
    }
}
```

```
Counter counter1 = (Counter) TypedActor.newInstance(Counter.class, CounterImpl.class);
Counter counter2 = (Counter) TypedActor.newInstance(Counter.class, CounterImpl.class);

Coordination.coordinate(true, new Atomically() {
    public void atomically() {
        counter1.increment();
        counter2.increment();
    }
});

TypedActor.stop(counter1);
TypedActor.stop(counter2);
```

5.8 Remote Actors (Java)

Contents

- Managing the Remote Service
 - Starting remote service in user code as a library
 - Starting remote service as part of the stand-alone Kernel
 - Stopping the server
 - Connecting and shutting down a client connection explicitly
 - Client message frame size configuration
 - Client reconnect configuration
 - Remote Client message buffering and send retry on failure
- Running Remote Server in untrusted mode
- Using secure cookie for remote client authentication
 - Enabling secure cookie authentication
 - Generating and using the secure cookie
- Client-managed Remote UntypedActor
- Server-managed Remote UntypedActor
 - Server side setup
 - Session bound server side setup
 - Client side usage
- Automatic remote 'sender' reference management
- Identifying remote actors
- Client-managed Remote Typed Actors
- Server-managed Remote Typed Actors
 - Server side setup
 - Client side usage
- Data Compression Configuration
- Code provisioning
- Subscribe to Remote Client events
- Subscribe to Remote Server events
- Message Serialization
 - Protobuf

Module stability: **SOLID**

Akka supports starting interacting with UntypedActors and TypedActors on remote nodes using a very efficient and scalable NIO implementation built upon [JBoss Netty](#) and [Google Protocol Buffers](#) .

The usage is completely transparent with local actors, both in regards to sending messages and error handling and propagation as well as supervision, linking and restarts. You can send references to other Actors as part of the message.

WARNING: For security reasons, do not run an Akka node with a Remote Actor port reachable by untrusted connections unless you have supplied a classloader that restricts access to the JVM.

5.8.1 Managing the Remote Service

Starting remote service in user code as a library

Here is how to start up the server and specify the hostname and port programmatically:

```
import static akka.actor.Actors.*;

remote().start("localhost", 2552);

// Specify the classloader to use to load the remote class (actor)
remote().start("localhost", 2552, classLoader);
```

Here is how to start up the server and specify the hostname and port in the ‘akka.conf’ configuration file (see the section below for details):

```
import static akka.actor.Actor.*;

remote().start();

// Specify the classloader to use to load the remote class (actor)
remote().start(classLoader);
```

Starting remote service as part of the stand-alone Kernel

You simply need to make sure that the service is turned on in the external ‘akka.conf’ configuration file.

```
akka {
  remote {
    server {
      service = on
      hostname = "localhost"
      port = 2552
      connection-timeout = 1000 # in millis
    }
  }
}
```

Stopping the server

```
import static akka.actor.Actor.*;

remote().shutdown();
```

Connecting and shutting down a client connection explicitly

Normally you should not have to start and stop the client connection explicitly since that is handled by Akka on a demand basis. But if you for some reason want to do that then you can do it like this:

```
import static akka.actor.Actor.*;
import java.net.InetSocketAddress;

remote().shutdownClientConnection(new InetSocketAddress("localhost", 6666)); //Returns true if suc
remote().restartClientConnection(new InetSocketAddress("localhost", 6666)); //Returns true if suc
```

Client message frame size configuration

You can define the max message frame size for the remote messages:

```
akka {
  remote {
    client {
      message-frame-size = 1048576
    }
  }
}
```

Client reconnect configuration

The Client automatically performs reconnection upon connection failure.

You can configure it like this:

```
akka {
  remote {
    client {
      reconnect-delay = 5           # in seconds (5 sec default)
      read-timeout = 10            # in seconds (10 sec default)
      reconnection-time-window = 600 # the maximum time window that a client should try to reconnect
    }
  }
}
```

The client will automatically try to reconnect to the server if the connection is broken. By default it has a reconnection window of 10 minutes (600 seconds).

If it has not been able to reconnect during this period of time then it is shut down and further attempts to use it will yield a 'RemoteClientException'. The 'RemoteClientException' contains the message as well as a reference to the address that is not yet connect in order for you to retrieve it and do an explicit connect if needed.

You can also register a listener that will listen for example the 'RemoteClientStopped' event, retrieve the address that got disconnected and reconnect explicitly.

See the section on client listener and events below for details.

Remote Client message buffering and send retry on failure

The Remote Client implements message buffering on network failure. This feature has zero overhead (even turned on) in the successful scenario and a queue append operation in case of unsuccessful send. So it is really really fast.

The default behavior is that the remote client will maintain a transaction log of all messages that it has failed to send due to network problems (not other problems like serialization errors etc.). The client will try to resend these messages upon first successful reconnect and the message ordering is maintained. This means that the remote client will swallow all exceptions due to network failure and instead queue remote messages in the transaction log. The failures will however be reported through the remote client life-cycle events as well as the regular Akka event handler. You can turn this behavior on and off in the configuration file. It gives 'at-least-once' semantics, use a message id/counter for discarding potential duplicates (or use idempotent messages).

```
akka {
  remote {
    client {
      buffering {
        retry-message-send-on-failure = on
        capacity = -1                # If negative (or zero) then an unbounded mailbox is used
                                    # If positive then a bounded mailbox is used and the capacity is the limit
      }
    }
  }
}
```

If you choose a capacity higher than 0, then a bounded queue will be used and if the limit of the queue is reached then a 'RemoteClientMessageBufferException' will be thrown.

5.8.2 Running Remote Server in untrusted mode

You can run the remote server in untrusted mode. This means that the server will not allow any client-managed remote actors or any life-cycle messages and methods. This is useful if you want to let untrusted clients use server-managed actors in a safe way. This can optionally be combined with the secure cookie authentication mechanism described below as well as the SSL support for remote actor communication.

If the client is trying to perform one of these unsafe actions then a 'java.lang.SecurityException' is thrown on the server as well as transferred to the client and thrown there as well.

Here is how you turn it on:

```
akka {
  remote {
    server {
      untrusted-mode = on # the default is 'off'
    }
  }
}
```

The messages that it prevents are all that extends ‘LifecycleMessage’: * case class HotSwap(..) * case object RevertHotSwap * case class Restart(..) * case class Exit(..) * case class Link(..) * case class Unlink(..) * case class UnlinkAndStop(..) * case object ReceiveTimeout

It also prevents the client from invoking any life-cycle and side-effecting methods, such as: * start * stop * link * unlink * spawnLink * etc.

5.8.3 Using secure cookie for remote client authentication

Akka is using a similar scheme for remote client node authentication as Erlang; using secure cookies. In order to use this authentication mechanism you have to do two things:

- Enable secure cookie authentication in the remote server
- Use the same secure cookie on all the trusted peer nodes

Enabling secure cookie authentication

The first one is done by enabling the secure cookie authentication in the remote server section in the configuration file:

```
akka {
  remote {
    server {
      require-cookie = on
    }
  }
}
```

Now if you have try to connect to a server from a client then it will first try to authenticate the client by comparing the secure cookie for the two nodes. If they are the same then it allows the client to connect and use the server freely but if they are not the same then it will throw a ‘java.lang.SecurityException’ and not allow the client to connect.

Generating and using the secure cookie

The secure cookie can be any string value but in order to ensure that it is secure it is best to randomly generate it. This can be done by invoking the ‘generate_config_with_secure_cookie.sh’ script which resides in the ‘\$AKKA_HOME/scripts’ folder. This script will generate and print out a complete ‘akka.conf’ configuration file with the generated secure cookie defined that you can either use as-is or cut and paste the ‘secure-cookie’ snippet. Here is an example of its generated output:

```
# This config imports the Akka reference configuration.
include "akka-reference.conf"

# In this file you can override any option defined in the 'akka-reference.conf' file.
# Copy in all or parts of the 'akka-reference.conf' file and modify as you please.

akka {
  remote {
    secure-cookie = "000E02050F0300040C050C0D060A040306090B0C"
```



```
}
}
```

The simplest way to use it is to have it create your ‘akka.conf’ file like this:

```
cd $AKKA_HOME
./scripts/generate_config_with_secure_cookie.sh > ./config/akka.conf
```

Now it is good to make sure that the configuration file is only accessible by the owner of the file. On Unix-style file system this can be done like this:

```
chmod 400 ./config/akka.conf
```

Running this script requires having ‘scala’ on the path (and will take a couple of seconds to run since it is using Scala and has to boot up the JVM to run).

You can also generate the secure cookie by using the ‘Crypt’ object and its ‘generateSecureCookie’ method.

```
import akka.util.Crypt;

String secureCookie = Crypt.generateSecureCookie();
```

The secure cookie is a cryptographically secure randomly generated byte array turned into a SHA-1 hash.

5.8.4 Client-managed Remote UntypedActor

DEPRECATED AS OF 1.1

The client creates the remote actor and “moves it” to the server.

When you define an actors as being remote it is instantiated as on the remote host and your local actor becomes a proxy, it works as a handle to the remote actor. The real execution is always happening on the remote node.

Here is an example:

```
import akka.actor.UntypedActor;
import static akka.actor.Actors.*;

class MyActor extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        ...
    }
}

//How to make it client-managed:
remote().actorOf(MyActor.class, "192.68.23.769", 2552);
```

An UntypedActor can also start remote child Actors through one of the “spawn/link” methods. These will start, link and make the UntypedActor remote atomically.

```
...
getContext().spawnRemote(MyActor.class, hostname, port, timeoutInMsForFutures);
getContext().spawnLinkRemote(MyActor.class, hostname, port, timeoutInMsForFutures);
...
```

5.8.5 Server-managed Remote UntypedActor

Here it is the server that creates the remote actor and the client can ask for a handle to this actor.

Server side setup

The API for server managed remote actors is really simple. 2 methods only:

```
import akka.actor.Actors;
import akka.actor.UntypedActor;

class MyActor extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        ...
    }
}

Actors.remote().start("localhost", 2552).register("hello-service", Actors.actorOf(HelloWorldActor
```

Actors created like this are automatically started.

You can also register an actor by its UUID rather than ID or handle. This is done by prefixing the handle with the “uuid:” protocol.

```
server.register("uuid:" + actor.uuid, actor);

server.unregister("uuid:" + actor.uuid);
```

Session bound server side setup

Session bound server managed remote actors work by creating and starting a new actor for every client that connects. Actors are stopped automatically when the client disconnects. The client side is the same as regular server managed remote actors. Use the function `registerPerSession` instead of `register`.

Session bound actors are useful if you need to keep state per session, e.g. username. They are also useful if you need to perform some cleanup when a client disconnects by overriding the `postStop` method as described here

```
import static akka.actor.Actors.*;
import akka.japi.Creator;

class HelloWorldActor extends Actor {
    ...
}

remote().start("localhost", 2552);

remote().registerPerSession("hello-service", new Creator<ActorRef>() {
    public ActorRef create() {
        return actorOf(HelloWorldActor.class);
    }
});
```

Note that the second argument in `registerPerSession` is a `Creator`, it means that the `create` method will create a new `ActorRef` each invocation. It will be called to create an actor every time a session is established.

Client side usage

```
import static akka.actor.Actors.*;
ActorRef actor = remote().actorFor("hello-service", "localhost", 2552);

Object result = actor.sendRequestReply("Hello");
```

There are many variations on the ‘`remote()#actorFor`’ method. Here are some of them:

```
... = remote().actorFor(className, hostname, port);
... = remote().actorFor(className, timeout, hostname, port);
... = remote().actorFor(uuid, className, hostname, port);
... = remote().actorFor(uuid, className, timeout, hostname, port);
... // etc
```

All of these also have variations where you can pass in an explicit ‘ClassLoader’ which can be used when deserializing messages sent from the remote actor.

5.8.6 Automatic remote ‘sender’ reference management

The sender of a remote message will be reachable with a reply through the remote server on the node that the actor is residing, automatically. Please note that firewalled clients won’t work right now. [2011-01-05]

5.8.7 Identifying remote actors

The ‘id’ field in the ‘Actor’ class is of importance since it is used as identifier for the remote actor. If you want to create a brand new actor every time you instantiate a remote actor then you have to set the ‘id’ field to a unique ‘String’ for each instance. If you want to reuse the same remote actor instance for each new remote actor (of the same class) you create then you don’t have to do anything since the ‘id’ field by default is equal to the name of the actor class.

Here is an example of overriding the ‘id’ field:

```
import akka.actor.UntypedActor;
import com.eaio.uuid.UUID;

class MyActor extends UntypedActor {
  public MyActor() {
    getContext().setId(new UUID().toString());
  }

  public void onReceive(Object message) throws Exception {
    // ...
  }
}
```

5.8.8 Client-managed Remote Typed Actors

DEPRECATED AS OF 1.1

Remote Typed Actors are created through the ‘TypedActor.newRemoteInstance’ factory method.

```
MyPOJO remoteActor = (MyPOJO) TypedActor.newRemoteInstance(MyPOJO.class, MyPOJOImpl.class, "localhost");
```

And if you want to specify the timeout:

```
MyPOJO remoteActor = (MyPOJO) TypedActor.newRemoteInstance(MyPOJO.class, MyPOJOImpl.class, timeout);
```

You can also define the Typed Actor to be a client-managed-remote service by adding the ‘RemoteAddress’ configuration element in the declarative supervisor configuration:

```
new Component(
  Foo.class,
  FooImpl.class,
  new Lifecycle(new Permanent(), 1000),
  1000,
  new RemoteAddress("localhost", 2552))
```

5.8.9 Server-managed Remote Typed Actors

WARNING: Remote TypedActors do not work with overloaded methods on your TypedActor, refrain from using overloading.

Server side setup

The API for server managed remote typed actors is nearly the same as for untyped actor:

```
import static akka.actor.Actors.*;
remote().start("localhost", 2552);
```

```
RegistrationService typedActor = TypedActor.newInstance(RegistrationService.class, RegistrationSe
remote().registerTypedActor("user-service", typedActor);
```

Client side usage

```
import static akka.actor.Actors.*;
RegistrationService actor = remote().typedActorFor(RegistrationService.class, "user-service", 5000);
actor.registerUser(...);
```

There are variations on the ‘remote()#typedActorFor’ method. Here are some of them:

```
... = remote().typedActorFor(interfaceClazz, serviceIdOrClassName, hostname, port);
... = remote().typedActorFor(interfaceClazz, serviceIdOrClassName, timeout, hostname, port);
... = remote().typedActorFor(interfaceClazz, serviceIdOrClassName, timeout, hostname, port, classLoader);
```

5.8.10 Data Compression Configuration

Akka uses compression to minimize the size of the data sent over the wire. Currently it only supports ‘zlib’ compression but more will come later.

You can configure it like this:

```
akka {
  remote {
    compression-scheme = "zlib" # Options: "zlib" (lzf to come), leave out for no compression
    zlib-compression-level = 6 # Options: 0-9 (1 being fastest and 9 being the most compressed),
    ...
  }
}
```

5.8.11 Code provisioning

Akka does currently not support automatic code provisioning but requires you to have the remote actor class files available on both the “client” the “server” nodes. This is something that will be addressed soon. Until then, sorry for the inconvenience.

5.8.12 Subscribe to Remote Client events

Akka has a subscription API for remote client events. You can register an Actor as a listener and this actor will have to be able to process these events:

```
class RemoteClientError { Throwable cause; RemoteClientModule client; InetSocketAddress remoteAddress; }
class RemoteClientDisconnected { RemoteClientModule client; InetSocketAddress remoteAddress; }
class RemoteClientConnected { RemoteClientModule client; InetSocketAddress remoteAddress; }
class RemoteClientStarted { RemoteClientModule client; InetSocketAddress remoteAddress; }
class RemoteClientShutdown { RemoteClientModule client; InetSocketAddress remoteAddress; }
class RemoteClientWriteFailed { Object message; Throwable cause; RemoteClientModule client; InetSocketAddress remoteAddress; }
```

So a simple listener actor can look like this:

```
import akka.actor.UntypedActor;
import akka.remoteinterface.*;

class Listener extends UntypedActor {

    public void onReceive(Object message) throws Exception {
        if (message instanceof RemoteClientError) {
            RemoteClientError event = (RemoteClientError) message;
            Throwable cause = event.getCause();
            // ...
        } else if (message instanceof RemoteClientConnected) {
            RemoteClientConnected event = (RemoteClientConnected) message;
            // ...
        } else if (message instanceof RemoteClientDisconnected) {
            RemoteClientDisconnected event = (RemoteClientDisconnected) message;
            // ...
        } else if (message instanceof RemoteClientStarted) {
            RemoteClientStarted event = (RemoteClientStarted) message;
            // ...
        } else if (message instanceof RemoteClientShutdown) {
            RemoteClientShutdown event = (RemoteClientShutdown) message;
            // ...
        } else if (message instanceof RemoteClientWriteFailed) {
            RemoteClientWriteFailed event = (RemoteClientWriteFailed) message;
            // ...
        }
    }
}
```

Registration and de-registration can be done like this:

```
ActorRef listener = Actors.actorOf(Listener.class);
...
Actors.remote().addListener(listener);
...
Actors.remote().removeListener(listener);
```

5.8.13 Subscribe to Remote Server events

Akka has a subscription API for the server events. You can register an Actor as a listener and this actor will have to be able to process these events:

```
class RemoteServerStarted { RemoteServerModule server; }
class RemoteServerShutdown { RemoteServerModule server; }
class RemoteServerError { Throwable cause; RemoteServerModule server; }
class RemoteServerClientConnected { RemoteServerModule server; Option<InetSocketAddress> clientAddress; }
class RemoteServerClientDisconnected { RemoteServerModule server; Option<InetSocketAddress> clientAddress; }
class RemoteServerClientClosed { RemoteServerModule server; Option<InetSocketAddress> clientAddress; }
class RemoteServerWriteFailed { Object request; Throwable cause; RemoteServerModule server; Option<InetSocketAddress> clientAddress; }
```

So a simple listener actor can look like this:

```
import akka.actor.UntypedActor;
import akka.remoteinterface.*;

class Listener extends UntypedActor {

    public void onReceive(Object message) throws Exception {
        if (message instanceof RemoteClientError) {
            RemoteClientError event = (RemoteClientError) message;
            Throwable cause = event.getCause();
            // ...
        }
    }
}
```

```

    } else if (message instanceof RemoteClientConnected) {
        RemoteClientConnected event = (RemoteClientConnected) message;
        // ...
    } else if (message instanceof RemoteClientDisconnected) {
        RemoteClientDisconnected event = (RemoteClientDisconnected) message;
        // ...
    } else if (message instanceof RemoteClientStarted) {
        RemoteClientStarted event = (RemoteClientStarted) message;
        // ...
    } else if (message instanceof RemoteClientShutdown) {
        RemoteClientShutdown event = (RemoteClientShutdown) message;
        // ...
    } else if (message instanceof RemoteClientWriteFailed) {
        RemoteClientWriteFailed event = (RemoteClientWriteFailed) message;
        // ...
    }
}
}
}

```

Registration and de-registration can be done like this:

```

import static akka.actor.Actors.*;

ActorRef listener = actorOf(Listener.class);
...
remote().addListener(listener);
...
remote().removeListener(listener);

```

5.8.14 Message Serialization

All messages that are sent to remote actors need to be serialized to binary format to be able to travel over the wire to the remote node. This is done by letting your messages extend one of the traits in the ‘akka.serialization.Serializable’ object. If the messages don’t implement any specific serialization trait then the runtime will try to use standard Java serialization.

Here is one example, but full documentation can be found in the [Serialization \(Java\)](#).

Protobuf

Protobuf message specification needs to be compiled with ‘protoc’ compiler.

```

message ProtobufPOJO {
    required uint64 id = 1;
    required string name = 2;
    required bool status = 3;
}

```

Using the generated message builder to send the message to a remote actor:

```

actor.tell(ProtobufPOJO.newBuilder()
    .setId(11)
    .setStatus(true)
    .setName("Coltrane")
    .build());

```

5.9 Serialization (Java)

Contents

- [Serialization of a Stateless Actor](#)
- [Serialization of a Stateful Actor](#)

Akka serialization module has been documented extensively under the *Serialization (Scala)* section. In this section we will point out the different APIs that are available in Akka for Java based serialization of ActorRefs. The Scala APIs of ActorSerialization has implicit Format objects that set up the type class based serialization. In the Java API, the Format objects need to be specified explicitly.

5.9.1 Serialization of a Stateless Actor

Step 1: Define the Actor

```
import akka.actor.UntypedActor;

public class SerializationTestActor extends UntypedActor {
    public void onReceive(Object msg) {
        getContext().replySafe("got it!");
    }
}
```

Step 2: Define the typeclass instance for the actor

Note how the generated Java classes are accessed using the \$class based naming convention of the Scala compiler.

```
import akka.serialization.StatelessActorFormat;

class SerializationTestActorFormat implements StatelessActorFormat<SerializationTestActor> {
    @Override
    public SerializationTestActor fromBinary(byte[] bytes, SerializationTestActor act) {
        return (SerializationTestActor) StatelessActorFormat$class.fromBinary(this, bytes, act);
    }

    @Override
    public byte[] toBinary(SerializationTestActor ac) {
        return StatelessActorFormat$class.toBinary(this, ac);
    }
}
```

Step 3: Serialize and de-serialize

The following JUnit snippet first creates an actor using the default constructor. The actor is, as we saw above a stateless one. Then it is serialized and de-serialized to get back the original actor. Being stateless, the de-serialized version behaves in the same way on a message as the original actor.

```
import akka.actor.ActorRef;
import akka.actor.ActorTimeoutException;
import akka.actor.Actors;
import akka.actor.UntypedActor;
import akka.serialization.Format;
import akka.serialization.StatelessActorFormat;
import static akka.serialization.ActorSerialization.*;

@Test public void mustBeAbleToSerializeAfterCreateActorRefFromClass() {
    ActorRef ref = Actors.actorOf(SerializationTestActor.class);
    assertNotNull(ref);
    ref.start();
    try {
        Object result = ref.sendRequestReply("Hello");
    }
```

```

        assertEquals("got it!", result);
    } catch (ActorTimeoutException ex) {
        fail("actor should not time out");
    }

    Format<SerializationTestActor> f = new SerializationTestActorFormat();
    byte[] bytes = toBinaryJ(ref, f, false);
    ActorRef r = fromBinaryJ(bytes, f);
    assertNotNull(r);
    r.start();
    try {
        Object result = r.sendRequestReply("Hello");
        assertEquals("got it!", result);
    } catch (ActorTimeoutException ex) {
        fail("actor should not time out");
    }
    ref.stop();
    r.stop();
}

```

5.9.2 Serialization of a Stateful Actor

Let's now have a look at how to serialize an actor that carries a state with it. Here the expectation is that the serialization of the actor will also persist the state information. And after de-serialization we will get back the state with which it was serialized.

Step 1: Define the Actor

```

import akka.actor.UntypedActor;

public class MyUntypedActor extends UntypedActor {
    int count = 0;

    public void onReceive(Object msg) {
        if (msg.equals("hello")) {
            count = count + 1;
            getContext().replyUnsafe("world " + count);
        } else if (msg instanceof String) {
            count = count + 1;
            getContext().replyUnsafe("hello " + msg + " " + count);
        } else {
            throw new IllegalArgumentException("invalid message type");
        }
    }
}

```

Note the actor has a state in the form of an Integer. And every message that the actor receives, it replies with an addition to the integer member.

Step 2: Define the instance of the typeclass

```

import akka.actor.UntypedActor;
import akka.serialization.Format;
import akka.serialization.SerializerFactory;

class MyUntypedActorFormat implements Format<MyUntypedActor> {
    @Override
    public MyUntypedActor fromBinary(byte[] bytes, MyUntypedActor act) {
        ProtobufProtocol.Counter p =
            (ProtobufProtocol.Counter) new SerializerFactory().getProtobuf().fromBinary(bytes, ProtobufProtocol.Counter.class);
        act.count = p.getCount();
        return act;
    }
}

```



```

    }

    @Override
    public byte[] toBinary(MyUntypedActor ac) {
        return ProtobufProtocol.Counter.newBuilder().setCount(ac.count()).build().toByteArray();
    }
}

```

Note the usage of Protocol Buffers to serialize the state of the actor. `ProtobufProtocol.Counter` is something you need to define yourself

Step 3: Serialize and de-serialize

```

import akka.actor.ActorRef;
import akka.actor.ActorTimeoutException;
import akka.actor.Actors;
import static akka.serialization.ActorSerialization.*;

@Test public void mustBeAbleToSerializeAStatefulActor() {
    ActorRef ref = Actors.actorOf(MyUntypedActor.class);
    assertNotNull(ref);
    ref.start();
    try {
        Object result = ref.sendRequestReply("hello");
        assertEquals("world 1", result);
        result = ref.sendRequestReply("hello");
        assertEquals("world 2", result);
    } catch (ActorTimeoutException ex) {
        fail("actor should not time out");
    }

    Format<MyUntypedActor> f = new MyUntypedActorFormat();
    byte[] bytes = toBinaryJ(ref, f, false);
    ActorRef r = fromBinaryJ(bytes, f);
    assertNotNull(r);
    r.start();
    try {
        Object result = r.sendRequestReply("hello");
        assertEquals("world 3", result);
        result = r.sendRequestReply("hello");
        assertEquals("world 4", result);
    } catch (ActorTimeoutException ex) {
        fail("actor should not time out");
    }
    ref.stop();
    r.stop();
}

```

Note how the de-serialized version starts with the state value with which it was earlier serialized.

5.10 Fault Tolerance Through Supervisor Hierarchies (Java)

Contents

- Concurrency
- Distributed actors
- Supervision
 - OneForOne
 - AllForOne
 - Restart callbacks
 - Defining a supervisor's restart strategy
 - Defining actor life-cycle
- Supervising Untyped Actor
 - Declarative supervisor configuration
 - Declaratively define actors as remote services
 - Programmatic linking and supervision of Untyped Actors
 - The supervising actor's side of things
 - The supervised actor's side of things
 - Reply to initial senders
 - Handling too many actor restarts within a specific time limit
- Supervising Typed Actors
 - Declarative supervisor configuration
 - Restart callbacks
 - Programmatic linking and supervision of TypedActors

Module stability: **SOLID**

The “let it crash” approach to fault/error handling, implemented by linking actors, is very different to what Java and most non-concurrency oriented languages/frameworks have adopted. It's a way of dealing with failure that is designed for concurrent and distributed systems.

5.10.1 Concurrency

Throwing an exception in concurrent code (let's assume we are using non-linked actors), will just simply blow up the thread that currently executes the actor.

- There is no way to find out that things went wrong (apart from inspecting the stack trace).
- There is nothing you can do about it.

Here actors provide a clean way of getting notification of the error and do something about it.

Linking actors also allow you to create sets of actors where you can be sure that either: - All are dead - None are dead

This is very useful when you have thousands of concurrent actors. Some actors might have implicit dependencies and together implement a service, computation, user session etc.

It encourages non-defensive programming. Don't try to prevent things from go wrong, because they will, whether you want it or not. Instead; expect failure as a natural state in the life-cycle of your app, crash early and let someone else (that sees the whole picture), deal with it.

5.10.2 Distributed actors

You can't build a fault-tolerant system with just one single box - you need at least two. Also, you (usually) need to know if one box is down and/or the service you are talking to on the other box is down. Here actor supervision/linking is a critical tool for not only monitoring the health of remote services, but to actually manage the service, do something about the problem if the actor or node is down. Such as restarting actors on the same node or on another node.

In short, it is a very different way of thinking, but a way that is very useful (if not critical) to building fault-tolerant highly concurrent and distributed applications, which is as valid if you are writing applications for the JVM or the Erlang VM (the origin of the idea of “let-it-crash” and actor supervision).


5.10.3 Supervision

Supervisor hierarchies originate from [Erlang’s OTP framework](#).


A supervisor is responsible for starting, stopping and monitoring its child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary. This makes for a completely different view on how to write fault-tolerant servers. Instead of trying all things possible to prevent an error from happening, this approach embraces failure. It shifts the view to look at errors as something natural and something that **will** happen, instead of trying to prevent it; embraces it. Just ‘Let It Crash™’, since the components will be reset to a stable state and restarted upon failure.

Akka has two different restart strategies; All-For-One and One-For-One. Best explained using some pictures (referenced from [erlang.org](http://www.erlang.org/doc/design_principles/sup4.gif)):

OneForOne

The OneForOne fault handler will restart only the component that has crashed.  image:http://www.erlang.org/doc/design_principles/sup4.gif

AllForOne

The AllForOne fault handler will restart all the components that the supervisor is managing, including the one that have crashed. This strategy should be used when you have a certain set of components that are coupled in some way that if one is crashing they all need to be reset to a stable state before continuing.  image:http://www.erlang.org/doc/design_principles/sup5.gif

Restart callbacks

There are two different callbacks that an UntypedActor or TypedActor can hook in to:

- Pre restart
- Post restart

These are called prior to and after the restart upon failure and can be used to clean up and reset/reinitialize state upon restart. This is important in order to reset the component failure and leave the component in a fresh and stable state before consuming further messages.

Defining a supervisor’s restart strategy

Both the Typed Actor supervisor configuration and the Actor supervisor configuration take a ‘FaultHandlingStrategy’ instance which defines the fault management. The different strategies are:

- AllForOne
- OneForOne

These have the semantics outlined in the section above.

Here is an example of how to define a restart strategy:

```
new AllForOneStrategy( //Or OneForOneStrategy
    new Class[]{ Exception.class }, //List of Exceptions/Throwables to handle
    3,                             // maximum number of restart retries
    5000                           // within time in millis
)
```

Defining actor life-cycle

The other common configuration element is the ‘LifeCycle’ which defines the life-cycle. The supervised actor can define one of two different life-cycle configurations:

- Permanent: which means that the actor will always be restarted.
- Temporary: which means that the actor will **not** be restarted, but it will be shut down through the regular shutdown process so the ‘postStop’ callback function will be called.

Here is an example of how to define the life-cycle:

```
import static akka.config.Supervision.*;

getContext().setLifeCycle(permanent()); //permanent() means that the component will always be restarted
getContext().setLifeCycle(temporary()); //temporary() means that the component will not be restarted
```

5.10.4 Supervising Untyped Actor

Declarative supervisor configuration

The Actor’s supervision can be declaratively defined by creating a ‘Supervisor’ factory object. Here is an example:

```
import static akka.config.Supervision.*;
import static akka.actor.Actors.*;

Supervisor supervisor = Supervisor.apply(
    new SupervisorConfig(
        new AllForOneStrategy(new Class[]{Exception.class}, 3, 5000),
        new Supervise[] {
            new Supervise(
                actorOf(MyActor1.class),
                permanent()),
            new Supervise(
                actorOf(MyActor2.class),
                permanent())
        })
);
```

Supervisors created like this are implicitly instantiated and started.

To configure a handler function for when the actor underlying the supervisor receives a `MaximumNumberOfRestartsWithinTimeRangeReached` message, you can specify a `Procedure2<ActorRef,MaximumNumberOfRestartsWithinTimeRangeReached>` when creating the `SupervisorConfig`. This handler will be called with the `ActorRef` of the supervisor and the `MaximumNumberOfRestartsWithinTimeRangeReached` message.

```
import static akka.config.Supervision.*;
import static akka.actor.Actors.*;
import akka.event.EventHandler;
import akka.japi.Procedure2;

Procedure2<ActorRef, MaximumNumberOfRestartsWithinTimeRangeReached> handler = new Procedure2<ActorRef, MaximumNumberOfRestartsWithinTimeRangeReached>() {
    public void apply(ActorRef ref, MaximumNumberOfRestartsWithinTimeRangeReached max) {
        EventHandler.error(ref, max);
    }
};

Supervisor supervisor = Supervisor.apply(
    new SupervisorConfig(
        new AllForOneStrategy(new Class[]{Exception.class}, 3, 5000),
        new Supervise[] {
            new Supervise(
                actorOf(MyActor1.class),
```

```
permanent()),
new Supervise(
    actorOf(MyActor2.class),
    permanent())
}, handler));
```

You can link and unlink actors from a declaratively defined supervisor using the ‘link’ and ‘unlink’ methods:

```
Supervisor supervisor = Supervisor.apply(...);
supervisor.link(..);
supervisor.unlink(..);
```

You can also create declarative supervisors through the ‘SupervisorFactory’ factory object. Use this factory instead of the ‘Supervisor’ factory object if you want to control instantiation and starting of the Supervisor, if not then it is easier and better to use the ‘Supervisor’ factory object.

Example usage:

```
import static akka.config.Supervision.*;
import static akka.actor.Actors.*;

SupervisorFactory factory = new SupervisorFactory(
    new SupervisorConfig(
        new OneForOneStrategy(new Class[] {Exception.class}, 3, 5000),
        new Supervise[] {
            new Supervise(
                actorOf(MyActor1.class),
                permanent()),
            new Supervise(
                actorOf(MyActor2.class),
                temporary())
        })
);
```

Then create a new instance our Supervisor and start it up explicitly.

```
SupervisorFactory supervisor = factory.newInstance();
supervisor.start(); // start up all managed servers
```

Declaratively define actors as remote services

You can expose your actors as remote services by specifying the `registerAsRemote` to **true** in `Supervise`.

Here is an example:

```
import static akka.config.Supervision.*;
import static akka.actor.Actors.*;

Supervisor supervisor = Supervisor.apply(
    new SupervisorConfig(
        new AllForOneStrategy(new Class[] {Exception.class}, 3, 5000),
        new Supervise[] {
            new Supervise(
                actorOf(MyActor1.class),
                permanent(),
                true)
        })
);
```

Programmatic linking and supervision of Untyped Actors

Untyped Actors can at runtime create, spawn, link and supervise other actors. Linking and unlinking is done using one of the ‘link’ and ‘unlink’ methods available in the ‘ActorRef’ (therefore prefixed with `getContext()` in these examples).

Here is the API and how to use it from within an ‘Actor’:

```
// link and unlink actors
getContext().link(actorRef);
getContext().unlink(actorRef);

// starts and links Actors atomically
getContext().startLink(actorRef);
getContext().startLinkRemote(actorRef);

// spawns (creates and starts) actors
getContext().spawn(MyActor.class);
getContext().spawnRemote(MyActor.class);

// spawns and links Actors atomically
getContext().spawnLink(MyActor.class);
getContext().spawnLinkRemote(MyActor.class);
```

A child actor can tell the supervising actor to unlink him by sending him the ‘Unlink(this)’ message. When the supervisor receives the message he will unlink and shut down the child. The supervisor for an actor is available in the ‘supervisor: Option[Actor]’ method in the ‘ActorRef’ class. Here is how it can be used.

```
ActorRef supervisor = getContext().getSupervisor();
if (supervisor != null) supervisor.tell(new Unlink(getContext()))
```

The supervising actor’s side of things

If a linked Actor is failing and throws an exception then an ‘new Exit(deadActor, cause)’ message will be sent to the supervisor (however you should never try to catch this message in your own message handler, it is managed by the runtime).

The supervising Actor also needs to define a fault handler that defines the restart strategy the Actor should accommodate when it traps an ‘Exit’ message. This is done by setting the ‘setFaultHandler’ method.

The different options are:

- AllForOneStrategy(trapExit, maxNrOfRetries, withinTimeRange)
 - trapExit is an Array of classes inheriting from Throwable, they signal which types of exceptions this actor will handle
- OneForOneStrategy(trapExit, maxNrOfRetries, withinTimeRange)
 - trapExit is an Array of classes inheriting from Throwable, they signal which types of exceptions this actor will handle

Here is an example:

```
getContext().setFaultHandler(new AllForOneStrategy(new Class[] { MyException.class, IOException.class })
```

Putting all this together it can look something like this:

```
class MySupervisor extends UntypedActor {
  public MySupervisor() {
    getContext().setFaultHandler(new AllForOneStrategy(new Class[] { MyException.class, IOException.class })

  }

  public void onReceive(Object message) throws Exception {
    if (message instanceof Register) {
      Register event = (Register)message;
      UntypedActorRef actor = event.getActor();
      context.link(actor);
    } else throw new IllegalArgumentException("Unknown message: " + message);
  }
}
```

You can also link an actor from outside the supervisor like this:

```
UntypedActor supervisor = Actors.registry().actorsFor(MySupervisor.class)[0];
supervisor.link(actorRef);
```

The supervised actor's side of things

The supervised actor needs to define a life-cycle. This is done by setting the `lifeCycle` field as follows:

```
import static akka.config.Supervision.*;

getContext().setLifeCycle(permanent()); // Permanent or Temporary
```

Default is 'Permanent' so if you don't set the life-cycle then that is what you get.

In the supervised Actor you can override the 'preRestart' and 'postRestart' callback methods to add hooks into the restart process. These methods take the reason for the failure, e.g. the exception that caused termination and restart of the actor as argument. It is in these methods that **you** have to add code to do cleanup before termination and initialization after restart. Here is an example:

```
class FaultTolerantService extends UntypedActor {

    @Override
    public void preRestart(Throwable reason) {
        ... // clean up before restart
    }

    @Override
    public void postRestart(Throwable reason) {
        ... // reinit stable state after restart
    }
}
```

Reply to initial senders

Supervised actors have the option to reply to the initial sender within `preRestart`, `postRestart` and `postStop`. A reply within these methods is possible after `receive` has thrown an exception. When `receive` returns normally it is expected that any necessary reply has already been done within `receive`. Here's an example.

```
public class FaultTolerantService extends UntypedActor {
    public void onReceive(Object msg) {
        // do something that may throw an exception
        // ...

        getContext().replySafe("ok");
    }

    @Override
    public void preRestart(Throwable reason) {
        getContext().replySafe(reason.getMessage());
    }

    @Override
    public void postStop() {
        getContext().replySafe("stopped by supervisor");
    }
}
```

- A reply within `preRestart` or `postRestart` must be a safe reply via `getContext().replySafe()` because a `getContext().replyUnsafe()` will throw an exception when the actor is restarted without having failed. This can be the case in context of `AllForOne` restart strategies.

- A reply within `postStop` must be a safe reply via `getContext().replySafe()` because a `getContext().replyUnsafe()` will throw an exception when the actor has been stopped by the application (and not by a supervisor) after successful execution of `receive` (or no execution at all).

Handling too many actor restarts within a specific time limit

If you remember, when you define the ‘RestartStrategy’ you also defined maximum number of restart retries within time in millis.

```
new AllForOneStrategy( // FaultHandlingStrategy policy (AllForOneStrategy or OneForOneStrategy)
    new Class[]{MyException.class, IOException.class}, //What types of errors will be handled
    3, // maximum number of restart retries
    5000 // within time in millis
);
```

Now, what happens if this limit is reached?

What will happen is that the failing actor will send a system message to its supervisor called ‘MaximumNumberOfRestartsWithinTimeRangeReached’ with the following these properties:

- `victim`: `ActorRef`
- `maxNrOfRetries`: `int`
- `withinTimeRange`: `int`
- `lastExceptionCausingRestart`: `Throwable`

If you want to be able to take action upon this event (highly recommended) then you have to create a message handle for it in the supervisor.

Here is an example:

```
public class SampleUntypedActorSupervisor extends UntypedActor {
    ...

    public void onReceive(Object message) throws Exception {
        if (message instanceof MaximumNumberOfRestartsWithinTimeRangeReached) {
            MaximumNumberOfRestartsWithinTimeRangeReached event = (MaximumNumberOfRestartsWithinTimeRangeReached) message;
            ... = event.getVictim();
            ... = event.getMaxNrOfRetries();
            ... = event.getWithinTimeRange();
            ... = event.getLastExceptionCausingRestart();
        } else throw new IllegalArgumentException("Unknown message: " + message);
    }
}
```

You will also get this log warning similar to this:

```
WAR [20100715-14:05:25.821] actor: Maximum number of restarts [5] within time range [5000] reached
WAR [20100715-14:05:25.821] actor: Will *not* restart actor [Actor[akka.actor.SupervisorHierarchySupervisor]]
WAR [20100715-14:05:25.821] actor: Last exception causing restart was [akka.actor.SupervisorHierarchySupervisor$RestartException]
```

If you don’t define a message handler for this message then you don’t get an error but the message is simply not sent to the supervisor. Instead you will get a log warning.

5.10.5 Supervising Typed Actors

Declarative supervisor configuration

To configure Typed Actors for supervision you have to consult the ‘TypedActorConfigurator’ and its ‘configure’ method. This method takes a ‘RestartStrategy’ and an array of ‘Component’ definitions defining the Typed Actors

and their ‘LifeCycle’. Finally you call the ‘supervise’ method to start everything up. The Java configuration elements reside in the ‘akka.config.JavaConfig’ class and need to be imported statically.

Here is an example:

```
import static akka.config.Supervision.*;
import static akka.config.SupervisorConfig.*;

TypedActorConfigurator manager = new TypedActorConfigurator();

manager.configure(
    new AllForOneStrategy(new Class[]{Exception.class}, 3, 1000),
    new SuperviseTypedActor[] {
        new SuperviseTypedActor(
            Foo.class,
            FooImpl.class,
            temporary(),
            1000),
        new SuperviseTypedActor(
            Bar.class,
            BarImpl.class,
            permanent(),
            1000)
    }).supervise();
```

Then you can retrieve the Typed Actor as follows:

```
Foo foo = (Foo) manager.getInstance(Foo.class);
```

Restart callbacks

In the supervised TypedActor you can override the ‘preRestart’ and ‘postRestart’ callback methods to add hooks into the restart process. These methods take the reason for the failure, e.g. the exception that caused termination and restart of the actor as argument. It is in these methods that **you** have to add code to do cleanup before termination and initialization after restart. Here is an example:

```
class FaultTolerantService extends TypedActor {

    @Override
    public void preRestart(Throwable reason) {
        ... // clean up before restart
    }

    @Override
    public void postRestart(Throwable reason) {
        ... // reinit stable state after restart
    }
}
```

Programatic linking and supervision of TypedActors

TypedActors can be linked and unlinked just like UntypedActors:

```
TypedActor.link(supervisor, supervised);

TypedActor.unlink(supervisor, supervised);
```

If the parent TypedActor (supervisor) wants to be able to do handle failing child TypedActors, e.g. be able restart the linked TypedActor according to a given fault handling scheme then it has to set its ‘trapExit’ flag to an array of Exceptions that it wants to be able to trap:

```
TypedActor.faultHandler(supervisor, new AllForOneStrategy(new Class[]{IOException.class}, 3, 2000));
```

For convenience there is an overloaded link that takes trapExit and faultHandler for the supervisor as arguments. Here is an example:

```
import static akka.actor.TypedActor.*;
import static akka.config.Supervision.*;

foo = newInstance(Foo.class, FooImpl.class, 1000);
bar = newInstance(Bar.class, BarImpl.class, 1000);

link(foo, bar, new AllForOneStrategy(new Class[]{IOException.class}, 3, 2000));

// alternative: chaining
bar = faultHandler(foo, new AllForOneStrategy(new Class[]{IOException.class}, 3, 2000)).newInstance();

link(foo, bar);
```

5.11 Dispatchers (Java)

Contents

- Default dispatcher
- Setting the dispatcher
- Types of dispatchers
 - Thread-based
 - Event-based
 - Priority event-based
 - Work-stealing event-based
- Making the Actor mailbox bounded
 - Global configuration
 - Per-instance based configuration

Module stability: **SOLID**

The Dispatcher is an important piece that allows you to configure the right semantics and parameters for optimal performance, throughput and scalability. Different Actors have different needs.

Akka supports dispatchers for both event-driven lightweight threads, allowing creation of millions threads on a single workstation, and thread-based Actors, where each dispatcher is bound to a dedicated OS thread.

The event-based Actors currently consume ~600 bytes per Actor which means that you can create more than 6.5 million Actors on 4 GB RAM.

5.11.1 Default dispatcher

For most scenarios the default settings are the best. Here we have one single event-based dispatcher for all Actors created. The dispatcher used is globalExecutorBasedEventDrivenDispatcher in akka.dispatch.Dispatchers.

But if you feel that you are starting to contend on the single dispatcher (the ‘Executor’ and its queue) or want to group a specific set of Actors for a dedicated dispatcher for better flexibility and configurability then you can override the defaults and define your own dispatcher. See below for details on which ones are available and how they can be configured.

5.11.2 Setting the dispatcher

Normally you set the dispatcher from within the Actor itself. The dispatcher is defined by the ‘dispatcher: MessageDispatcher’ member field in ‘ActorRef’.

```
class MyActor extends UntypedActor {
  public MyActor() {
    getContext().setDispatcher(..); // set the dispatcher
  }
  ...
}
```

You can also set the dispatcher for an Actor **before** it has been started:

```
actorRef.setDispatcher(dispatcher);
```

5.11.3 Types of dispatchers

There are six different types of message dispatchers:

- Thread-based
- Event-based
- Priority event-based
- Work-stealing event-based

Factory methods for all of these, including global versions of some of them, are in the ‘akka.dispatch.Dispatchers’ object.

Let’s now walk through the different dispatchers in more detail.

Thread-based

The ‘ThreadBasedDispatcher’ binds a dedicated OS thread to each specific Actor. The messages are posted to a ‘LinkedBlockingQueue’ which feeds the messages to the dispatcher one by one. A ‘ThreadBasedDispatcher’ cannot be shared between actors. This dispatcher has worse performance and scalability than the event-based dispatcher but works great for creating “daemon” Actors that consumes a low frequency of messages and are allowed to go off and do their own thing for a longer period of time. Another advantage with this dispatcher is that Actors do not block threads for each other.

```
Dispatcher dispatcher = Dispatchers.newThreadBasedDispatcher(actorRef);
```

It would normally be used from within the actor like this:

```
class MyActor extends UntypedActor {
  public MyActor() {
    getContext().setDispatcher(Dispatchers.newThreadBasedDispatcher(getContext()));
  }
  ...
}
```

Event-based

The ‘ExecutorBasedEventDrivenDispatcher’ binds a set of Actors to a thread pool backed up by a ‘BlockingQueue’. This dispatcher is highly configurable and supports a fluent configuration API to configure the ‘BlockingQueue’ (type of queue, max items etc.) as well as the thread pool.

The event-driven dispatchers **must be shared** between multiple Typed Actors and/or Actors. One best practice is to let each top-level Actor, e.g. the Actors you define in the declarative supervisor config, to get their own dispatcher but reuse the dispatcher for each new Actor that the top-level Actor creates. But you can also share

dispatcher between multiple top-level Actors. This is very use-case specific and needs to be tried out on a case by case basis. The important thing is that Akka tries to provide you with the freedom you need to design and implement your system in the most efficient way in regards to performance, throughput and latency.

It comes with many different predefined `BlockingQueue` configurations:

- `Bounded LinkedBlockingQueue`
- `Unbounded LinkedBlockingQueue`
- `Bounded ArrayBlockingQueue`
- `Unbounded ArrayBlockingQueue`
- `SynchronousQueue`

You can also set the rejection policy that should be used, e.g. what should be done if the dispatcher (e.g. the Actor) can't keep up and the mailbox is growing up to the limit defined. You can choose between four different rejection policies:

- `java.util.concurrent.ThreadPoolExecutor.CallerRuns` - will run the message processing in the caller's thread as a way to slow him down and balance producer/consumer
- `java.util.concurrent.ThreadPoolExecutor.AbortPolicy` - rejected messages by throwing a `'RejectedExecutionException'`
- `java.util.concurrent.ThreadPoolExecutor.DiscardPolicy` - discards the message (throws it away)
- `java.util.concurrent.ThreadPoolExecutor.DiscardOldestPolicy` - discards the oldest message in the mailbox (throws it away)

You can read more about these policies [here](#).

Here is an example:

```
import akka.actor.Actor;
import akka.dispatch.Dispatchers;
import java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy;

class MyActor extends UntypedActor {
    public MyActor() {
        getContext().setDispatcher(Dispatchers.newExecutorBasedEventDrivenDispatcher(name)
            .withNewThreadPoolWithLinkedBlockingQueueWithCapacity(100)
            .setCorePoolSize(16)
            .setMaxPoolSize(128)
            .setKeepAliveTimeInMillis(60000)
            .setRejectionPolicy(new CallerRunsPolicy())
            .build());
    }
    ...
}
```

This `'ExecutorBasedEventDrivenDispatcher'` allows you to define the `'throughput'` it should have. This defines the number of messages for a specific Actor the dispatcher should process in one single sweep. Setting this to a higher number will increase throughput but lower fairness, and vice versa. If you don't specify it explicitly then it uses the default value defined in the `'akka.conf'` configuration file:

```
actor {
    throughput = 5
}
```

If you don't define a the `'throughput'` option in the configuration file then the default value of `'5'` will be used.

Browse the [Scaladoc API](#) or look at the code for all the options available.

Priority event-based

Sometimes it's useful to be able to specify priority order of messages, that is done by using `PriorityExecutorBasedEventDrivenDispatcher` and supply a `java.util.Comparator[MessageInvocation]` or use a `akka.dispatch.PriorityGenerator` (recommended):

Creating a `PriorityExecutorBasedEventDrivenDispatcher` using `PriorityGenerator`:

```
package some.pkg;

import akka.actor.*;
import akka.dispatch.*;

public class Main {
    // A simple Actor that just prints the messages it processes
    public static class MyActor extends UntypedActor {
        public void onReceive(Object message) throws Exception {
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        // Create a new PriorityGenerator, lower prio means more important
        PriorityGenerator gen = new PriorityGenerator() {
            public int gen(Object message) {
                if (message.equals("highpriority")) return 0;           // "highpriority" messages should be
                else if (message.equals("lowpriority")) return 100;    // "lowpriority" messages should be
                else return 50; // We default to 50
            }
        };

        // We create an instance of the actor that will print out the messages it processes
        ActorRef ref = Actors.actorOf(MyActor.class);
        // We create a new Priority dispatcher and seed it with the priority generator
        ref.setDispatcher(new PriorityExecutorBasedEventDrivenDispatcher("foo", gen));

        ref.start(); // Start the actor
        ref.getDispatcher().suspend(ref); // Suspending the actor so it doesn't start to treat the me
        ref.tell("lowpriority");
        ref.tell("lowpriority");
        ref.tell("highpriority");
        ref.tell("pigdog");
        ref.tell("pigdog2");
        ref.tell("pigdog3");
        ref.tell("highpriority");
        ref.getDispatcher().resume(ref); // Resuming the actor so it will start treating its messages
    }
}
```

Prints:

```
highpriority highpriority pigdog pigdog2 pigdog3 lowpriority lowpriority
```

Work-stealing event-based

The `ExecutorBasedEventDrivenWorkStealingDispatcher` is a variation of the `ExecutorBasedEventDrivenDispatcher` in which Actors of the same type can be set up to share this dispatcher and during execution time the different actors will steal messages from other actors if they have less messages to process. This can be a great way to improve throughput at the cost of a little higher latency.

Normally the way you use it is to define a static field to hold the dispatcher and then set in in the Actor explicitly.

```
class MyActor extends UntypedActor {
    public static MessageDispatcher dispatcher = Dispatchers.newExecutorBasedEventDrivenWorkStealing
```

```
public MyActor() {
    getContext().setDispatcher(dispatcher);
}
...
}
```

Here is an article with some more information: [Load Balancing Actors with Work Stealing Techniques](#) Here is another article discussing this particular dispatcher: [Flexible load balancing with Akka in Scala](#)

5.11.4 Making the Actor mailbox bounded

Global configuration

You can make the Actor mailbox bounded by a capacity in two ways. Either you define it in the configuration file under ‘default-dispatcher’. This will set it globally.

```
actor {
  default-dispatcher {
    mailbox-capacity = -1          # If negative (or zero) then an unbounded mailbox is used (default)
                                  # If positive then a bounded mailbox is used and the capacity is set
  }
}
```

Per-instance based configuration

You can also do it on a specific dispatcher instance.

For the ‘ExecutorBasedEventDrivenDispatcher’ and the ‘ExecutorBasedWorkStealingDispatcher’ you can do it through their constructor

```
class MyActor extends UntypedActor {
  public MyActor() {
    int capacity = 100;
    Duration pushTimeout = new FiniteDuration(10, TimeUnit.SECONDS);
    MailboxType mailboxCapacity = new BoundedMailbox(false, capacity, pushTimeout);
    MessageDispatcher dispatcher =
      Dispatchers.newExecutorBasedEventDrivenDispatcher(name, throughput, mailboxCapacity).build()
    getContext().setDispatcher(dispatcher);
  }
  ...
}
```

For the ‘ThreadBasedDispatcher’, it is non-shareable between actors, and associates a dedicated Thread with the actor. Making it bounded (by specifying a capacity) is optional, but if you do, you need to provide a pushTimeout (default is 10 seconds). When trying to send a message to the Actor it will throw a `MessageQueueAppendFailedException` (“BlockingMessageTransferQueue transfer timed out”) if the message cannot be added to the mailbox within the time specified by the pushTimeout.

```
class MyActor extends UntypedActor {
  public MyActor() {
    int mailboxCapacity = 100;
    Duration pushTimeout = new FiniteDuration(10, TimeUnit.SECONDS);
    getContext().setDispatcher(Dispatchers.newThreadBasedDispatcher(getContext(), mailboxCapacity, pushTimeout));
  }
  ...
}
```

5.12 Routing (Java)

5.12.1 UntypedDispatcher

An `UntypedDispatcher` is an actor that routes incoming messages to outbound actors.

```
import static akka.actor.Actors.*;
import akka.actor.*;
import akka.routing.*;

//A Pinger is an UntypedActor that prints "Pinger: <message>"
class Pinger extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        System.out.println("Pinger: " + message);
    }
}

//A Ponger is an UntypedActor that prints "Ponger: <message>"
class Ponger extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        System.out.println("Ponger: " + message);
    }
}

public class MyDispatcher extends UntypedDispatcher {
    private ActorRef pinger = actorOf(Pinger.class).start();
    private ActorRef ponger = actorOf(Ponger.class).start();

    //Route Ping-messages to the pinger, and Pong-messages to the ponger
    public ActorRef route(Object message) {
        if("Ping".equals(message)) return pinger;
        else if("Pong".equals(message)) return ponger;
        else throw new IllegalArgumentException("I do not understand " + message);
    }
}

ActorRef dispatcher = actorOf(MyDispatcher.class).start();
dispatcher.tell("Ping"); //Prints "Pinger: Ping"
dispatcher.tell("Pong"); //Prints "Ponger: Pong"
```

5.12.2 UntypedLoadBalancer

An `UntypedLoadBalancer` is an actor that forwards messages it receives to a boundless sequence of destination actors.

```
import static akka.actor.Actors.*;
import akka.actor.*;
import akka.routing.*;
import static java.util.Arrays.asList;

//A Pinger is an UntypedActor that prints "Pinger: <message>"
class Pinger extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        System.out.println("Pinger: " + message);
    }
}

//A Ponger is an UntypedActor that prints "Ponger: <message>"
class Ponger extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        System.out.println("Ponger: " + message);
    }
}
```

```

    }
}

//Our load balancer, sends messages to a pinger, then a ponger, rinse and repeat.
public class MyLoadBalancer extends UntypedLoadBalancer {
    private InfiniteIterator<ActorRef> actors = new CyclicIterator<ActorRef>(asList(
        actorOf(Pinger.class).start(),
        actorOf(Ponger.class).start()
    ));

    public InfiniteIterator<ActorRef> seq() {
        return actors;
    }
}

ActorRef dispatcher = actorOf(MyLoadBalancer.class).start();
dispatcher.tell("Pong"); //Prints "Pinger: Pong"
dispatcher.tell("Ping"); //Prints "Ponger: Ping"
dispatcher.tell("Ping"); //Prints "Pinger: Ping"
dispatcher.tell("Pong"); //Prints "Ponger: Pong"

```

You can also send a ‘new Routing.Broadcast(msg)’ message to the router to have it be broadcasted out to all the actors it represents.

```
router.tell(new Routing.Broadcast(new PoisonPill()));
```

5.13 Guice Integration

Module stability: **STABLE**

All Typed Actors support dependency injection using [Guice](#) annotations (such as ‘@Inject’ etc.). The ‘TypedActorManager’ class understands Guice and will do the wiring for you.

5.13.1 External Guice modules

You can also plug in external Guice modules and have not-actors wired up as part of the configuration. Here is an example:

```

import static akka.config.Supervision.*;
import static akka.config.SupervisorConfig.*;

TypedActorConfigurator manager = new TypedActorConfigurator();

manager.configure(
    new AllForOneStrategy(new Class[]{Exception.class}, 3, 1000),
    new SuperviseTypedActor[] {
        new SuperviseTypedActor(
            Foo.class,
            FooImpl.class,
            temporary(),
            1000),
        new SuperviseTypedActor(
            Bar.class,
            BarImpl.class,
            permanent(),
            1000)
    })
.addExternalGuiceModule(new AbstractModule() {
    protected void configure() {
        bind(Ext.class).to(ExtImpl.class).in(Scopes.SINGLETON);
    }
});

```



```
    })  
    .configure()  
    .inject()  
    .supervise();
```

5.13.2 Retrieve the external Guice dependency

The external dependency can be retrieved like this:

```
Ext ext = manager.getExternalDependency(Ext.class);
```

INFORMATION FOR DEVELOPERS

6.1 Building Akka

This page describes how to build and run Akka from the latest source code.

- [Get the source code](#)
- [SBT - Simple Build Tool](#)
- [Building Akka](#)
 - [Fetching dependencies](#)
 - [Building](#)
 - [Publish to local Ivy repository](#)
 - [Publish to local Maven repository](#)
 - [SBT interactive mode](#)
 - [SBT batch mode](#)
- [Building Akka Modules](#)
- [Dependencies](#)

6.1.1 Get the source code

Akka uses [Git](#) and is hosted at [Github](#).

You first need Git installed on your machine. You can then clone the source repositories:

- Akka repository from <http://github.com/jboner/akka>
- Akka Modules repository from <http://github.com/jboner/akka-modules>

For example:

```
git clone git://github.com/jboner/akka.git
git clone git://github.com/jboner/akka-modules.git
```

If you have already cloned the repositories previously then you can update the code with `git pull`:

```
git pull origin master
```

6.1.2 SBT - Simple Build Tool

Akka is using the excellent [SBT](#) build system. So the first thing you have to do is to download and install SBT. You can read more about how to do that in the [SBT setup](#) documentation.

The SBT commands that you'll need to build Akka are all included below. If you want to find out more about SBT and using it for your own projects do read the [SBT documentation](#).

The Akka SBT build file is `project/build/AkkaProject.scala` with some properties defined in `project/build.properties`.

6.1.3 Building Akka

First make sure that you are in the akka code directory:

```
cd akka
```

Fetching dependencies

SBT does not fetch dependencies automatically. You need to manually do this with the `update` command:

```
sbt update
```

Once finished, all the dependencies for Akka will be in the `lib_managed` directory under each module: akka-actor, akka-stm, and so on.

Note: you only need to run update the first time you are building the code, or when the dependencies have changed.

Building

To compile all the Akka core modules use the `compile` command:

```
sbt compile
```

You can run all tests with the `test` command:

```
sbt test
```

If compiling and testing are successful then you have everything working for the latest Akka development version.

Publish to local Ivy repository

If you want to deploy the artifacts to your local Ivy repository (for example, to use from an SBT project) use the `publish-local` command:

```
sbt publish-local
```

Publish to local Maven repository

If you want to deploy the artifacts to your local Maven repository use:

```
sbt publish-local publish
```

SBT interactive mode

Note that in the examples above we are calling `sbt compile` and `sbt test` and so on. SBT also has an interactive mode. If you just run `sbt` you enter the interactive SBT prompt and can enter the commands directly. This saves starting up a new JVM instance for each command and can be much faster and more convenient.

For example, building Akka as above is more commonly done like this:

```
% sbt
[info] Building project akka 1.2 against Scala 2.9.1
[info]    using AkkaParentProject with sbt 0.7.6 and Scala 2.7.7
> update
[info]
```

```
[info] == akka-actor / update ==
...
[success] Successful.
[info]
[info] Total time ...
> compile
...
> test
...
```

SBT batch mode

It's also possible to combine commands in a single call. For example, updating, testing, and publishing Akka to the local Ivy repository can be done with:

```
sbt update test publish-local
```

6.1.4 Building Akka Modules

See the Akka Modules documentation.

6.1.5 Dependencies

If you are managing dependencies by hand you can find the dependencies for each module by looking in the `lib_managed` directories. For example, this will list all compile dependencies (providing you have the source code and have run `sbt update`):

```
cd akka
ls -l */lib_managed/compile
```

You can also look at the Ivy dependency resolution information that is created on `sbt update` and found in `~/.ivy2/cache`. For example, the `.ivy2/cache/se.scalablesolutions.akka-akka-remote-compile.xml` file contains the resolution information for the akka-remote module compile dependencies. If you open this file in a web browser you will get an easy to navigate view of dependencies.

6.2 Developer Guidelines

6.2.1 Code Style

The Akka code style follows [this document](#).

Here is a code style settings file for IntelliJ IDEA: [Download](#)

Please follow the code style. Look at the code around you and mimic.

6.2.2 Testing

All code that is checked in **should** have tests. All testing is done with `ScalaTest` and `ScalaCheck`.

- Name tests as **Test.scala** if they do not depend on any external stuff. That keeps surefire happy.
- Name tests as **Spec.scala** if they have external dependencies.

There is a testing standard that should be followed: [Ticket001Spec](#)

Actor TestKit

There is a useful test kit for testing actors: `akka.util.TestKit`. It enables assertions concerning replies received and their timing, there is more documentation in the *Testing Actor Systems* module.

NetworkFailureTest

You can use the ‘NetworkFailureTest’ trait to test network failure. See the ‘RemoteErrorHandlingNetworkTest’ test. Your tests needs to end with ‘NetworkTest’. They are disabled by default. To run them you need to enable a flag.

Example:

```
project akka-remote
set akka.test.network true
test-only akka.actor.remote.RemoteErrorHandlingNetworkTest
```

It uses ‘ipfw’ for network management. Mac OSX comes with it installed but if you are on another platform you might need to install it yourself. Here is a port:

<http://info.iet.unipi.it/~luigi/dummynet>

6.3 Documentation Guidelines

Contents

- Sphinx
- reStructuredText
 - Sections
 - Cross-referencing
- Build the documentation
 - Building
 - Installing Sphinx on OS X

The Akka documentation uses `reStructuredText` as its markup language and is built using `Sphinx`.

6.3.1 Sphinx

More to come...

6.3.2 reStructuredText

More to come...

Sections

Section headings are very flexible in reST. We use the following convention in the Akka documentation:

- # (over and under) for module headings
- = for sections
- – for subsections

- ^ for subsubsections
- ~ for subsubsubsections

Cross-referencing

Sections that may be cross-referenced across the documentation should be marked with a reference. To mark a section use `.. _ref-name:` before the section heading. The section can then be linked with `:ref: 'ref-name'`. These are unique references across the entire documentation.

For example:

```
.. _akka-module:

#####
Akka Module
#####

This is the module documentation.

.. _akka-section:

Akka Section
=====

Akka Subsection
-----

Here is a reference to "akka section": :ref:'akka-section' which will have the
name "Akka Section".
```

6.3.3 Build the documentation

First install [Sphinx](#). See below.

Building

```
cd akka-docs

make html
open _build/html/index.html

make pdf
open _build/latex/Akka.pdf
```

Installing Sphinx on OS X

Install [Homebrew](#)

Install Python and pip:

```
brew install python
/usr/local/share/python/easy_install pip
```

Add the Homebrew Python path to your \$PATH:

```
/usr/local/Cellar/python/2.7.1/bin
```

More information in case of trouble: <https://github.com/mxcl/homebrew/wiki/Homebrew-and-Python>

Install sphinx:

```
pip install sphinx
```

Add sphinx_build to your \$PATH:

```
/usr/local/share/python
```

Install BasicTeX package from: <http://www.tug.org/mactex/morepackages.html>

Add texlive bin to \$PATH:

```
/usr/local/texlive/2010basic/bin/universal-darwin
```

Add missing tex packages:

```
sudo tlmgr update --self
sudo tlmgr install titlesec
sudo tlmgr install framed
sudo tlmgr install threeparttable
sudo tlmgr install wrapfig
sudo tlmgr install helvetic
sudo tlmgr install courier
```

Link the akka pygments style:

```
cd /usr/local/Cellar/python/2.7.1/lib/python2.7/site-packages/pygments/styles
ln -s /path/to/akka/akka-docs/themes/akka/pygments/akka.py akka.py
```

6.4 Team

Name	Role	Email
Jonas Bonér	Founder, Despot, Committer	jonas AT jonasboner DOT com
Viktor Klang	Bad cop, Committer	viktor DOT klang AT gmail DOT com
Debasish Ghosh	Committer	dghosh AT acm DOT org
Ross McDonald	Alumni	rossajmcd AT gmail DOT com
Eckhart Hertzler	Alumni	
Mikael Höggqvist	Alumni	
Tim Perrett	Alumni	
Jeanfrancois Arcand	Alumni	jfarcand AT apache DOT org
Martin Krasser	Committer	krasserm AT gmail DOT com
Jan Van Besien	Alumni	
Michael Kober	Alumni	
Peter Vlugter	Committer	
Peter Veentjer	Committer	
Irmo Manie	Committer	
Heiko Seeberger	Committer	
Hiram Chirino	Committer	
Scott Clasen	Committer	
Roland Kuhn	Committer	
Patrik Nordwall	Committer	patrik DOT nordwall AT gmail DOT com
Derek Williams	Committer	derek AT nebv DOT ca

PROJECT INFORMATION

7.1 Migration Guides

7.1.1 Migration Guide 1.0.x to 1.1.x

Akka has now moved to Scala 2.9.x

Akka Actor

- is now dependency free, with the exception of the dependency on the `scala-library.jar`
- does not bundle any logging anymore, but you can subscribe to events within Akka by registering an event handler on `akka.event.EventHandler` or by specifying the FQN of an Actor in the `akka.conf` under `akka.event-handlers`; there is an `akka-slf4j` module which still provides the Logging trait and a default SLF4J logger adapter.

Don't forget to add a SLF4J backend though, we recommend:

```
lazy val logback = "ch.qos.logback" % "logback-classic" % "0.9.28" % "runtime"
```

- If you used `HawtDispatcher` and want to continue using it, you need to include `akka-dispatcher-extras.jar` from Akka Modules, in your `akka.conf` you need to specify: `akka.dispatch.HawtDispatcherConfigurator` instead of `HawtDispatcher`
- FSM: the `onTransition` method changed from `Function1` to `PartialFunction`; there is an implicit conversion for the precise types in place, but it may be necessary to add an underscore if you are passing an eta-expansion (using a method as function value).

Akka Typed Actor

- All methods starting with `get*` are deprecated and will be removed in post 1.1 release.

Akka Remote

- `UnparseableException` has been renamed to `CannotInstantiateRemoteExceptionDueToRemoteProtocol` (classname, message)

Akka HTTP

- `akka.servlet.Initializer` has been moved to `akka-kernel` to be able to have `akka-http` not depend on `akka-remote`. If you don't want to use the class for kernel, just create your own version of `akka.servlet.Initializer`, it's just a couple of lines of code and there are instructions in the [HTTP docs](#).

- akka.http.ListWriter has been removed in full, if you use it and want to keep using it, here's the code: [ListWriter](#).
- Jersey-server is now a “provided” dependency for akka-http, so you'll need to add the dependency to your project, it's built against Jersey 1.3

Akka Testkit

- The TestKit moved into the akka-testkit subproject and correspondingly into the akka.testkit package.

7.1.2 Migration guide from 0.10.x to 1.0.x

Akka & Akka Modules separated into two different repositories and distributions

Akka is split up into two different parts: * Akka - Reflects all the sections under ‘Scala API’ and ‘Java API’ in the navigation bar. * Akka Modules - Reflects all the sections under ‘Add-on modules’ in the navigation bar.

Download the release you need (Akka core or Akka Modules) from <http://akka.io/downloads> and unzip it.

Changed Akka URI

<http://akkasource.org> changed to <http://akka.io>

Reflects XSDs, Maven repositories, ScalaDoc etc.

Removed ‘se.scalablesolutions’ prefix

We have removed some boilerplate by shortening the Akka package from **se.scalablesolutions.akka** to just **akka** so just do a search-replace in your project, we apologize for the inconvenience, but we did it for our users.

Akka-core is no more

Akka-core has been split into akka-actor, akka-stm, akka-typed-actor & akka-remote this means that you need to update any deps you have on akka-core.

Config

Turning on/off modules

All the ‘service = on’ elements for turning modules on and off have been replaced by a top-level list of the enabled services.

Services available for turning on/off are: * “remote” * “http” * “camel”

All services are **OFF** by default. Enable the ones you are using.

```
akka {  
  enabled-modules = [] # Comma separated list of the enabled modules. Options: ["remote", "camel"  
}
```

Renames

- ‘rest’ section - has been renamed to ‘http’ to align with the module name ‘akka-http’.
- ‘storage’ section - has been renamed to ‘persistence’ to align with the module name ‘akka-persistence’.

```
akka {
  http {
    ..
  }

  persistence {
    ..
  }
}
```

Important changes from RC2-RC3

akka.config.SupervisionSupervise

Scala

```
def apply(actorRef: ActorRef, lifeCycle: Lifecycle, registerAsRemoteService: Boolean = false)
```

- boolean instead of remoteAddress, registers that actor with it's id as service name on the local server

akka.actor.Actors now is the API for Java to interact with Actors, Remoting and ActorRegistry:

Java

```
import static akka.actor.Actors.*; // <-- The important part

actorOf();
remote().actorOf();
registry().actorsFor("foo");
```

akka.actor.Actor now is the API for Scala to interact with Actors, Remoting and ActorRegistry:

Scala

```
import akka.actor.Actor._ // <-- The important part

actorOf().method
remote.actorOf()
registry.actorsFor("foo")
```

object UntypedActor has been deleted and replaced with akka.actor.Actors/akka.actor.Actor (Java/Scala)

- UntypedActor.actorOf -> Actors.actorOf (Java) or Actor.actorOf (Scala)

object ActorRegistry has been deleted and replaced with akka.actor.Actors.registry()/akka.actor.Actor.registry (Java/Scala)

- ActorRegistry. -> Actors.registry(). (Java) or Actor.registry. (Scala)

object RemoteClient has been deleted and replaced with akka.actor.Actors.remote()/akka.actor.Actor.remote (Java/Scala)

- RemoteClient -> Actors.remote() (Java) or Actor.remote (Scala)

object RemoteServer has been deleted and replaced with akka.actor.Actors.remote()/akka.actor.Actor.remote (Java/Scala)

- RemoteServer - deleted -> Actors.remote() (Java) or Actor.remote (Scala)

classes `RemoteActor`, `RemoteUntypedActor` and `RemoteUntypedConsumerActors` has been deleted and replaced with `akka.actor.Actors.remote().actorOf(x, host port)/akka.actor.Actor.remote.actorOf(x, host, port)`

- `RemoteActor`, `RemoteUntypedActor` - deleted, use: `remote().actorOf(YourActor.class, host, port)` (Java) or `remote.actorOf[YourActor](host, port)`

Remoted spring-actors now default to spring id as service-name, use “service-name” attribute on “remote”-tag to override

Listeners for `RemoteServer` and `RemoteClient` are now registered on `Actors.remote().addListener` (Java) or `Actor.remote.addListener` (Scala), this means that all listeners get all remote events, both remote server events and remote client events, **so adjust your code accordingly.**

`ActorRef.startLinkRemote` has been removed since one specified on creation whether the actor is client-managed or not.

Important change from RC3 to RC4

The Akka-Spring namespace has changed from `akkasource.org` and `scalablesolutions.se` to `http://akka.io/schema` and `http://akka.io/akka-<version>.xsd`

Module akka-actor

The `Actor.init` callback has been renamed to “preStart” to align with the general callback naming and is more clear about when it’s called.

The `Actor.shutdown` callback has been renamed to “postStop” to align with the general callback naming and is more clear about when it’s called.

The `Actor.initTransactionalState` callback has been removed, logic should be moved to preStart and be wrapped in an atomic block

`se.scalablesolutions.akka.config.ScalaConfig` and `se.scalablesolutions.akka.config.JavaConfig` have been merged into `akka.config.Supervision`

`RemoteAddress` has moved from `se.scalablesolutions.akka.config.ScalaConfig` to `akka.config`

The `ActorRef.lifeCycle` has changed signature from `Option[LifeCycle]` to `LifeCycle`, this means you need to change code that looks like this: `self.lifeCycle = Some(LifeCycle(Permanent))` to `self.lifeCycle = Permanent`

The equivalent to `self.lifeCycle = None` is `self.lifeCycle = UndefinedLifeCycle` `LifeCycle(Permanent)` becomes `Permanent` `new LifeCycle(permanent())` becomes `permanent()` (need to do: `import static se.scalablesolutions.akka.config.Supervision.*`; first)

`JavaConfig.Component` and `ScalaConfig.Component` have been consolidated and renamed as `Supervision.SuperviseTypedActor`

`self.trapExit` has been moved into the `FaultHandlingStrategy`, and `ActorRef.faultHandler` has switched type from `Option[FaultHandlingStrategy]` to `FaultHandlingStrategy`:

Scala

```
import akka.config.Supervision._

self.faultHandler = OneForOneStrategy(List(classOf[Exception]), 3, 5000)
```

Java

```
import static akka.Supervision.*;

getContext().setFaultHandler(new OneForOneStrategy(new Class[] { Exception.class }, 50, 1000))
```

RestartStrategy, **AllForOne**, **OneForOne** have been replaced with **AllForOneStrategy** and **OneForOneStrategy** in `se.scalablesolutions.akka.config.Supervision`

Scala

```
import akka.config.Supervision._
SupervisorConfig(
  OneForOneStrategy(List(classOf[Exception]), 3, 5000),
  Supervise(pingpong1, Permanent) :: Nil
)
```

Java

```
import static akka.Supervision.*;

new SupervisorConfig(
  new OneForOneStrategy(new Class[] { Exception.class }, 50, 1000),
  new Server[] { new Supervise(pingpong1, permanent()) }
)
```

We have removed the following factory methods:

Actor.actor { case foo => bar } Actor.transactor { case foo => bar } Actor.temporaryActor { case foo => bar } Actor.init {} receive { case foo => bar }

They started the actor and no config was possible, it was inconsistent and irreparable.

replace with your own factories, or:

Scala

```
actorOf( new Actor { def receive = { case foo => bar } } ).start
actorOf( new Actor { self.lifeCycle = Temporary; def receive = { case foo => bar } } ).start
```

ReceiveTimeout is now rescheduled after every message, before there was only an initial timeout. To stop rescheduling of **ReceiveTimeout**, set **receiveTimeout = None**

HotSwap

HotSwap does no longer use behavior stacking by default, but that is an option to both “become” and HotSwap.

HotSwap now takes for Scala a Function from ActorRef to a Receive, the ActorRef passed in is the reference to self, so you can do self.reply() etc.

Module akka-stm

The STM stuff is now in its own module. This means that there is no support for transactions or transactors in akka-actor.

Local and global

The **local/global** distinction has been dropped. This means that if the following general import was being used:

Scala

```
import akka.stm.local._
```

this is now just:

Scala

```
import akka.stm._
```

Coordinated is the new global

There is a new explicit mechanism for coordinated transactions. See the Scala Transactors and Java Transactors documentation for more information. Coordinated transactions and transactors are found in the `akka.transactor` package now. The usage of transactors has changed.

Agents

Agent is now in the akka-stm module and has moved to the `akka.agent` package. The implementation has been reworked and is now closer to Clojure agents. There is not much difference in general usage, the main changes involve interaction with the STM.

While updates to Agents are asynchronous, the state of an Agent is always immediately available for reading by any thread. Agents are integrated with the STM - any dispatches made in a transaction are held until that transaction commits, and are discarded if it is retried or aborted. There is a new `sendOff` method for long-running or blocking update functions.

Module akka-camel

Access to the CamelService managed by CamelServiceManager has changed:

- Method service renamed to mandatoryService (Scala)
- Method service now returns Option[CamelService] (Scala)
- Introduced method getMandatoryService() (Java)
- Introduced method getService() (Java)

Scala

```
import se.scalablesolutions.akka.camel.CamelServiceManager._
import se.scalablesolutions.akka.camel.CamelService

val o: Option[CamelService] = service
val s: CamelService = mandatoryService
```

Java

```
import se.scalablesolutions.akka.camel.CamelService;
import se.scalablesolutions.akka.japi.Option;
import static se.scalablesolutions.akka.camel.CamelServiceManager.*;

Option<CamelService> o = getService();
CamelService s = getMandatoryService();
```

Access to the CamelContext and ProducerTemplate managed by CamelContextManager has changed:

- Method context renamed to mandatoryContext (Scala)
- Method template renamed to mandatoryTemplate (Scala)
- Method service now returns Option[CamelContext] (Scala)
- Method template now returns Option[ProducerTemplate] (Scala)
- Introduced method getMandatoryContext() (Java)
- Introduced method getContext() (Java)

- Introduced method `getMandatoryTemplate()` (Java)
- Introduced method `getTemplate()` (Java)

Scala

```
import org.apache.camel.CamelContext
import org.apache.camel.ProducerTemplate

import se.scalablesolutions.akka.camel.CamelContextManager._

val co: Option[CamelContext] = context
val to: Option[ProducerTemplate] = template

val c: CamelContext = mandatoryContext
val t: ProducerTemplate = mandatoryTemplate
```

Java

```
import org.apache.camel.CamelContext;
import org.apache.camel.ProducerTemplate;

import se.scalablesolutions.akka.japi.Option;
import static se.scalablesolutions.akka.camel.CamelContextManager.*;

Option<CamelContext> co = getContext();
Option<ProducerTemplate> to = getTemplate();

CamelContext c = getMandatoryContext();
ProducerTemplate t = getMandatoryTemplate();
```

The following methods have been renamed on class `se.scalablesolutions.akka.camel.Message`:

- `bodyAs(Class)` has been renamed to `getBodyAs(Class)`
- `headerAs(String, Class)` has been renamed to `getHeaderAs(String, Class)`

The API for waiting for consumer endpoint activation and de-activation has been changed

- `CamelService.expectEndpointActivationCount` has been removed and replaced by `CamelService.awaitEndpointActivation`
- `CamelService.expectEndpointDeactivationCount` has been removed and replaced by `CamelService.awaitEndpointDeactivation`

Scala

```
import se.scalablesolutions.akka.actor.Actor
import se.scalablesolutions.akka.camel.CamelServiceManager._

val s = startCamelService
val actor = Actor.actorOf[SampleConsumer]

// wait for 1 consumer being activated
s.awaitEndpointActivation(1) {
  actor.start
}

// wait for 1 consumer being de-activated
s.awaitEndpointDeactivation(1) {
  actor.stop
}

s.stop
```

Java

```
import java.util.concurrent.TimeUnit;
import se.scalablesolutions.akka.actor.ActorRef;
import se.scalablesolutions.akka.actor.Actors;
import se.scalablesolutions.akka.camel.CamelService;
import se.scalablesolutions.akka.japi.SideEffect;
import static se.scalablesolutions.akka.camel.CamelServiceManager.*;

CamelService s = startCamelService();
final ActorRef actor = Actors.actorOf(SampleUntypedConsumer.class);

// wait for 1 consumer being activated
s.awaitEndpointActivation(1, new SideEffect() {
    public void apply() {
        actor.start();
    }
});

// wait for 1 consumer being de-activated
s.awaitEndpointDeactivation(1, new SideEffect() {
    public void apply() {
        actor.stop();
    }
});

s.stop();
```

Module Akka-Http

Atmosphere support has been removed. If you were using akka.comet.AkkaServlet for Jersey support only, you can switch that to: akka.http.AkkaRestServlet and it should work just like before.

Atmosphere has been removed because we have a new async http support in the form of Akka Mist, a very thin bridge between Servlet3.0/JettyContinuations and Actors, enabling Http-as-messages, read more about it here: <http://doc.akka.io/http#Mist%20-%20Lightweight%20Asynchronous%20HTTP>

If you really need Atmosphere support, you can add it yourself by following the steps listed at the start of: <http://doc.akka.io/comet>

Module akka-spring

The Akka XML schema URI has changed to <http://akka.io/schema/akka>

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:akka="http://akka.io/schema/akka"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://akka.io/schema/akka
http://akka.io/akka-1.0.xsd">

<!-- ... -->

</beans>
```

7.1.3 Migration Guide 0.9.x to 0.10.x

Module akka-camel

The following list summarizes the breaking changes since Akka 0.9.1.

- CamelService moved from package `se.scalablesolutions.akka.camel.service` one level up to `se.scalablesolutions.akka.camel`.
- CamelService.newInstance removed. For starting and stopping a CamelService, applications should use
 - CamelServiceManager.startCamelService and
 - CamelServiceManager.stopCamelService.
- Existing `def receive = produce` method definitions from Producer implementations must be removed (resolves compile error: method receive needs override modifier).
- The Producer.async method and the related Sync trait have been removed. This is now fully covered by Camel's [asynchronous routing engine](#).
- @consume annotation can not be placed any longer on actors (i.e. on type-level), only on typed actor methods. Consumer actors must mixin the Consumer trait.
- @consume annotation moved to package `se.scalablesolutions.akka.camel`.

Logging

We've switched to Logback (SLF4J compatible) for the logging, if you're having trouble seeing your log output you'll need to make sure that there's a `logback.xml` available on the classpath or you'll need to specify the location of the `logback.xml` file via the system property, ex: `-Dlogback.configurationFile=/path/to/logback.xml`

Configuration

- The configuration is now JSON-style (see below).
- Now you can define the time-unit to be used throughout the config file:

```
akka {
  version = "0.10"
  time-unit = "seconds"      # default timeout time unit for all timeout properties throughout the

  actor {
    timeout = 5               # default timeout for future based invocations
    throughput = 5            # default throughput for ExecutorBasedEventDrivenDispatcher
  }
  ...
}
```

RemoteClient events

All events now has a reference to the RemoteClient instance instead of 'hostname' and 'port'. This is more flexible. Enables simpler reconnecting etc.

7.1.4 Migration Guide 0.8.x to 0.9.x

This document describes between the 0.8.x and the 0.9 release.

Background for the new ActorRef

In the work towards 0.9 release we have now done a major change to how Actors are created. In short we have separated identity and value, created an ‘ActorRef’ that holds the actual Actor instance. This allows us to do many great things such as for example:

- Create serializable, immutable, network-aware Actor references that can be freely shared across the network. They “remember” their origin and will always work as expected.
- Not only kill and restart the same supervised Actor instance when it has crashed (as we do now), but dereference it, throw it away and make it eligible for garbage collection.
- etc. much more

These work very much like the ‘PID’ (process id) in Erlang.

These changes means that there is no difference in defining Actors. You still use the old Actor trait, all methods are there etc. But you can’t just new this Actor up and send messages to it since all its public API methods are gone. They now reside in a new class; ‘ActorRef’ and use need to use instances of this class to interact with the Actor (sending messages etc.).

Here is a short migration guide with the things that you have to change. It is a big conceptual change but in practice you don’t have to change much.

Creating Actors with default constructor

From:

```
val a = new MyActor
a ! msg
```

To:

```
import Actor._
val a = actorOf[MyActor]
a ! msg
```

You can also start it in the same statement:

```
val a = actorOf[MyActor].start
```

Creating Actors with non-default constructor

From:

```
val a = new MyActor(..)
a ! msg
```

To:

```
import Actor._
val a = actorOf(new MyActor(..))
a ! msg
```

Use of ‘self’ ActorRef API

Where you have used ‘this’ to refer to the Actor from within itself now use ‘self’:

```
self ! MessageToMe
```

Now the Actor trait only has the callbacks you can implement: * receive * postRestart/preRestart * init/shutdown
It has no state at all.

All API has been moved to ActorRef. The Actor is given its ActorRef through the 'self' member variable. Here you find functions like: * !, !!, !!! and forward * link, unlink, startLink, spawnLink etc * makeTransactional, makeRemote etc. * start, stop * etc.

Here you also find fields like * dispatcher = ... * id = ... * lifeCycle = ... * faultHandler = ... * trapExit = ... * etc.

This means that to use them you have to prefix them with 'self', like this:

```
self ! Message
```

However, for convenience you can import these functions and fields like below, which will allow you to drop the 'self' prefix:

```
class MyActor extends Actor {
  import self._
  id = ...
  dispatcher = ...
  spawnLink[OtherActor]
  ...
}
```

Serialization

If you want to serialize it yourself, here is how to do it:

```
val actorRef1 = actorOf[MyActor]

val bytes = actorRef1.toBinary

val actorRef2 = ActorRef.fromBinary(bytes)
```

If you are also using Protobuf then you can use the methods that work with Protobuf's Messages directly.

```
val actorRef1 = actorOf[MyActor]

val protobufMessage = actorRef1.toProtocol

val actorRef2 = ActorRef.fromProtocol(protobufMessage)
```

Camel

Some methods of the se.scalablesolutions.akka.camel.Message class have been deprecated in 0.9. These are

```
package se.scalablesolutions.akka.camel

case class Message(...) {
  // ...
  @deprecated def bodyAs[T](clazz: Class[T]): T
  @deprecated def setBodyAs[T](clazz: Class[T]): Message
  // ...
}
```

They will be removed in 1.0. Instead use

```
package se.scalablesolutions.akka.camel

case class Message(...) {
  // ...
  def bodyAs[T](implicit m: Manifest[T]): T =
```

```
def setBodyAs[T](implicit m: Manifest[T]): Message
// ...
}
```

Usage example: .. code-block:: scala

```
val m = Message(1.4) val b = m.bodyAs[String]
```

7.1.5 Migration Guide 0.7.x to 0.8.x

This is a case-by-case migration guide from Akka 0.7.x (on Scala 2.7.7) to Akka 0.8.x (on Scala 2.8.x)

Cases:

Actor.send is removed and replaced in full with Actor.!

```
myActor send "test"
```

becomes

```
myActor ! "test"
```

Actor.! now has it's implicit sender defaulted to None

```
def !(message: Any)(implicit sender: Option[Actor] = None)
```

“import Actor.Sender.Self” has been removed because it's not needed anymore

Remove

```
import Actor.Sender.Self
```

Actor.spawn now uses manifests instead of concrete class types

```
val someActor = spawn(classOf[MyActor])
```

becomes

```
val someActor = spawn[MyActor]
```

Actor.spawnRemote now uses manifests instead of concrete class types

```
val someActor = spawnRemote(classOf[MyActor], "somehost", 1337)
```

becomes

```
val someActor = spawnRemote[MyActor] ("somehost", 1337)
```

Actor.spawnLink now uses manifests instead of concrete class types

```
val someActor = spawnLink(classOf[MyActor])
```

becomes

```
val someActor = spawnLink[MyActor]
```

Actor.spawnLinkRemote now uses manifests instead of concrete class types

```
val someActor = spawnLinkRemote(classOf[MyActor], "somehost", 1337)
```

becomes

```
val someActor = spawnLinkRemote[MyActor] ("somehost", 1337)
```

Transaction.atomic and friends are moved into Transaction.Local._ and Transaction.Global._

We now make a difference between transaction management that are local within a thread and global across many threads (and actors).

7.2 Release Notes

7.2.1 Release 1.2

This release, while containing several substantial improvements, focuses on paving the way for the upcoming 2.0 release. A selection of changes is presented in the following, for the full list of tickets closed during the development cycle please refer to [the issue tracker](#).

- **Actor:**

- unified `Channel` abstraction for `Promise` & `Actor`
- reintegrate invocation tracing (to be enabled per class and globally)
- make last message available during `preRestart`
- experimental `freshInstance` life-cycle hook for priming the new instance during restart
- new textual primitives `tell(!)` and `ask(?, formerly !!!)`
- timeout for `ask` `Futures` taken from implicit argument (currently with fallback to deprecated `ActorRef.timeout`)

- **durable mailboxes:**

- beanstalk, file, mongo, redis

- **Future:**

- `onTimeout` callback
- select dispatcher for execution by implicit argument
- add safer cast methods `as[T]: T` and `mapTo[T]: Future[T]`

- **TestKit:**

- add `TestProbe` (can receive, reply and forward messages, supports all `TestKit` assertions)
- add `TestKit.awaitCond`
- support global time-factor for all timing assertions (for running on busy CI servers)

- **FSM:**
 - add `TestFSMRef`
 - add `LoggingFSM` (transition tracing, rolling event log)
- **updated dependencies:**
 - **Scala 2.9.1**
 - Jackson 1.8.0
 - Netty 3.2.5
 - Protobuf 2.4.1
 - ScalaTest 1.6.1
- various fixes, small improvements and documentation updates
- several **deprecations** in preparation for 2.0

Method	Replacement
<code>Actor.preRestart(cause)</code>	<code>Actor.preRestart(cause, lastMsg)</code>
<code>ActorRef.sendOneWay</code>	<code>ActorRef.tell</code>
<code>ActorRef.sendOneWaySafe</code>	<code>ActorRef.tryTell</code>
<code>ActorRef.sendRequestReply</code>	<code>ActorRef.ask(...).get()</code>
<code>ActorRef.sendRequestReplyFuture</code>	<code>ActorRef.ask(...).get()</code>
<code>ActorRef.replyUnsafe</code>	<code>ActorRef.reply</code>
<code>ActorRef.replySafe</code>	<code>ActorRef.tryReply</code>
<code>ActorRef.mailboxSize</code>	<code>ActorRef.dispatcher.mailboxSize(actorRef)</code>
<code>ActorRef.!!</code>	<code>ActorRef.?(...).as[T]</code>
<code>ActorRef.!!!</code>	<code>ActorRef.?</code>
<code>ActorRef.reply_?</code>	<code>ActorRef.tryReply</code>
<code>Future.receive</code>	<code>Future.onResult</code>
<code>Future.collect</code>	<code>Future.map</code>
<code>Future.failure</code>	<code>Future.recover</code>
<code>MessageDispatcher.pendingFutures</code>	<code>MessageDispatcher.tasks</code>
<code>RemoteClientModule.*Listener(s)</code>	<code>EventHandler.<X></code>
<code>TestKit.expectMsg(pf)</code>	<code>TestKit.expectMsgPF</code>
<code>TestKit.receiveWhile(pf)</code>	<code>TestKit.receiveWhile()(pf)</code>

Trivia

This release contains changes to 221 files, with 16844 insertions and 4010 deletions (diff between two points, does not count multiple changes to the same line). The authorship of the corresponding commits is distributed as shown below; the listing should not be taken too seriously, though, it has just been done using `git log --shortstat` and summing up the numbers, so it certainly misses details like who originally authored changes which were then back-ported from the master branch (do not fear, you will be correctly attributed when the stats for 2.0 are made).

Commits	Insertions	Deletions	Author
90	12315	245	Viktor Klang
78	3823	200	Roland Kuhn
41	9834	130	Patrik Nordwall
31	1819	131	Peter Vlugter
7	238	22	Derek Williams
4	86	25	Peter Veentjer
1	17	5	Debasish Ghosh
2	15	5	Jonas Bonér

Note: Release notes of previous releases consisted of ticket or change listings in no particular order

7.2.2 Release 1.1

- **ADD** - #647 Extract an akka-camel-typed module out of akka-camel for optional typed actor support (Martin Krasser)
- **ADD** - #654 Allow consumer actors to acknowledge in-only message exchanges (Martin Krasser)
- **ADD** - #669 Support self.reply in preRestart and postStop after exception in receive (Martin Krasser)
- **ADD** - #682 Support for fault-tolerant Producer actors (Martin Krasser)
- **ADD** - Move TestKit to akka-testkit and add CallingThreadDispatcher (Roland Kuhn)
- **ADD** - Remote Client message buffering transaction log for buffering messages failed to send due to network problems. Flushes the buffer on reconnect. (Jonas Bonér)
- **ADD** - Added trait simulate network problems/errors to be used for remote actor testing (Jonas Bonér)
- **ADD** - Add future and await methods to Agent (Peter Vlugter)
- **ADD** - #586 Allow explicit reconnect for RemoteClient (Viktor Klang)
- **ADD** - #587 Dead letter sink queue for messages sent through RemoteClient that didn't get sent due to connection failure (Viktor Klang)
- **ADD** - #598 actor.id when using akka-spring should be the id of the spring bean (Viktor Klang)
- **ADD** - #652 Reap expired futures from ActiveRemoteClientHandler (Viktor Klang)
- **ADD** - #656 Squeeze more out of EBEDD? (Viktor Klang)
- **ADD** - #715 EventHandler.error should be usable without Throwable (Viktor Klang)
- **ADD** - #717 Add ExecutionHandler to NettyRemoteServer for more performance and scalability (Viktor Klang)
- **ADD** - #497 Optimize remote sends done in local scope (Viktor Klang)
- **ADD** - #633 Add support for Scalaz in akka-modules (Derek Williams)
- **ADD** - #677 Add map, flatMap, foreach, and filter to Future (Derek Williams)
- **ADD** - #661 Optimized Future's internals (Derek Williams)
- **ADD** - #685 Optimize execution of Futures (Derek Williams)
- **ADD** - #711 Make Future.completeWith work with an uncompleted Future (Derek Williams)
- **UPD** - #667 Upgrade to Camel 2.7.0 (Martin Krasser)
- **UPD** - Updated HawtDispatch to 1.1 (Hiram Chirino)
- **UPD** - #688 Update Akka 1.1-SNAPSHOT to Scala 2.9.0-RC1 (Viktor Klang)
- **UPD** - #718 Add HawtDispatcher to akka-modules (Viktor Klang)
- **UPD** - #698 Deprecate client-managed actors (Viktor Klang)
- **UPD** - #730 Update Akka and Akka Modules to SBT 0.7.6-RC0 (Viktor Klang)
- **UPD** - #663 Update to latest scalatest (Derek Williams)
- **FIX** - Misc cleanup, API changes and refactorings (Jonas Bonér)
- **FIX** - #675 preStart() is called twice when creating new instance of TypedActor (Debasish Ghosh)
- **FIX** - #704 Write docs for Java Serialization (Debasish Ghosh)
- **FIX** - #645 Change Futures.awaitAll to not throw FutureTimeoutException but return a List[Option[Any]] (Viktor Klang)
- **FIX** - #681 Clean exit using server-managed remote actor via client (Viktor Klang)
- **FIX** - #720 Connection loss when sending to a dead remote actor (Viktor Klang)

- **FIX** - #593 Move Jetty specific stuff (with deps) from akka-http to akka-kernel (Viktor Klang)
- **FIX** - #638 ActiveRemoteClientHandler - Unexpected exception from downstream in remote client (Viktor Klang)
- **FIX** - #655 Remote actors with non-uuid names doesnt work for req./reply-pattern (Viktor Klang)
- **FIX** - #588 RemoteClient.shutdown does not remove client from Map with clients (Viktor Klang)
- **FIX** - #672 Remoting breaks if mutual DNS lookup isn't possible (Viktor Klang)
- **FIX** - #699 Remote typed actor per-session server won't start if called method has no result (Viktor Klang)
- **FIX** - #702 Handle ReadTimeoutException in akka-remote (Viktor Klang)
- **FIX** - #708 Fall back to Akka classloader if event-handler class cannot be found. (Viktor Klang)
- **FIX** - #716 Split akka-http and clean-up dependencies (Viktor Klang)
- **FIX** - #721 Inability to parse/load the Config should do a System.exit(-1) (Viktor Klang)
- **FIX** - #722 Race condition in Actor hotswapping (Viktor Klang)
- **FIX** - #723 MessageSerializer CNFE regression (Viktor Klang)
- **FIX** - #680 Remote TypedActor behavior differs from local one when sending to generic interfaces (Viktor Klang)
- **FIX** - #659 Calling await on a Future that is expired and uncompleted should throw an exception (Derek Williams)
- **REM** - #626 Update and clean up dependencies (Viktor Klang)
- **REM** - #623 Remove embedded-repo (Akka + Akka Modules) (Viktor Klang)
- **REM** - #686 Remove SBinary (Viktor Klang)

7.2.3 Release 1.0-RC6

- **FIX** - #628 Supervied TypedActors fails to restart (Viktor Klang)
- **FIX** - #629 Stuck upon actor invocation (Viktor Klang)

7.2.4 Release 1.0-RC5

- **FIX** - Source JARs published to 'src' instead of 'source' || Odd Moller ||
- **FIX** - #612 Conflict between Spring autostart=true for Consumer actors and <akka:camel-service> (Martin Krasser)
- **FIX** - #613 Change Akka XML schema URI to <http://akka.io/schema/akka> (Martin Krasser)
- **FIX** - Spring XSD namespace changed from 'akkasource.org' to 'akka.io' (Viktor Klang)
- **FIX** - Checking for remote secure cookie is disabled by default if no akka.conf is loaded (Viktor Klang)
- **FIX** - Changed Casbah to ScalaToolsRepo for akka-sbt-plugin (Viktor Klang)
- **FIX** - ActorRef.forward now doesn't require the sender to be set on the message (Viktor Klang)

7.2.5 Release 1.0-RC3

- **ADD** - #568 Add autostart attribute to Spring actor configuration (Viktor Klang)
- **ADD** - #586 Allow explicit reconnect for remote clients (Viktor Klang)
- **ADD** - #587 Add possibility for dead letter queues for failed remote sends (Viktor Klang)

- **ADD** - #497 Optimize remote send in local scope (Viktor Klang)
- **ADD** - Improved Java Actor API: akka.actor.actors (Viktor Klang)
- **ADD** - Improved Scala Actor API: akka.actor.Actor (Viktor Klang)
- **ADD** - #148 Create a testing framework for testing Actors (Roland Kuhn)
- **ADD** - Support Replica Set/Replica Pair connection modes with MongoDB Persistence || Brendan McAdams ||
- **ADD** - User configurable Write Concern settings for MongoDB Persistence || Brendan McAdams ||
- **ADD** - Support for configuring MongoDB Persistence with MongoDB's URI Connection String || Brendan McAdams ||
- **ADD** - Support for Authentication with MongoDB Persistence || Brendan McAdams ||
- **FIX** - Misc bug fixes || Team ||
- **FIX** - #603 Race condition in Remote send (Viktor Klang)
- **FIX** - #594 Log statement in RemoteClientHandler was wrongly formatted (Viktor Klang)
- **FIX** - #580 Message uuids must be generated (Viktor Klang)
- **FIX** - #583 Serialization classloader has a visibility issue (Viktor Klang)
- **FIX** - #598 By default the bean ID should become the actor id for Spring actor configuration (Viktor Klang)
- **FIX** - #577 RemoteClientHandler swallows certain exceptions (Viktor Klang)
- **FIX** - #581 Fix edgecase where an exception could not be deserialized (Viktor Klang)
- **FIX** - MongoDB write success wasn't being properly checked; fixed (integrated w/ new write concern features) || Brendan McAdams ||
- **UPD** - Improvements to FSM module akka.actor.FSM || Manie & Kuhn ||
- **UPD** - Changed Akka URI to <http://akka.io>. Reflects both XSDs, Maven repositories etc. (Jonas Bonér)
- **REM** - #574 Remote RemoteClient, RemoteServer and RemoteNode (Viktor Klang)
- **REM** - object UntypedActor, object ActorRegistry, class RemoteActor, class RemoteUntypedActor, class RemoteUntypedConsumerActor (Viktor Klang)

7.2.6 Release 1.0-RC1

- **ADD** - #477 Added support for Remote Agents (Viktor Klang)
- **ADD** - #460 Hotswap for Java API (UntypedActor) (Viktor Klang)
- **ADD** - #471 Added support for TypedActors to return Java Option (Viktor Klang)
- **ADD** - New design and API for more fluent and intuitive FSM module (Roland Kuhn)
- **ADD** - Added secure cookie based remote node authentication (Jonas Bonér)
- **ADD** - Untrusted safe mode for remote server (Jonas Bonér)
- **ADD** - Refactored config file format - added list of enabled modules etc. (Jonas Bonér)
- **ADD** - Docs for Dataflow Concurrency (Jonas Bonér)
- **ADD** - Made remote message frame size configurable (Jonas Bonér)
- **ADD** - #496 Detect when Remote Client disconnects (Jonas Bonér)
- **ADD** - #472 Improve API to wait for endpoint activation/deactivation (more ...) (Martin Krasser)
- **ADD** - #473 Allow consumer actors to customize their own routes (more ...) (Martin Krasser)
- **ADD** - #504 Add session bound server managed remote actors || Paul Pach ||

- **ADD** - DSL for FSM (Irmo Manie)
- **ADD** - Shared unit test for all dispatchers to enforce Actor Model (Viktor Klang)
- **ADD** - #522 Make stacking optional for become and HotSwap (Viktor Klang)
- **ADD** - #524 Make frame size configurable for client&server (Bonér & Klang)
- **ADD** - #526 Add onComplete callback to Future (Viktor Klang)
- **ADD** - #536 Document Channel-abstraction for later replies (Viktor Klang)
- **ADD** - #540 Include self-reference as parameter to HotSwap (Viktor Klang)
- **ADD** - #546 Include Garrick Evans' Akka-mist into master (Viktor Klang)
- **ADD** - #438 Support remove operation in PersistentVector (Scott Clasen)
- **ADD** - #229 Memcached protocol support for Persistence module (Scott Clasen)
- **ADD** - Amazon SimpleDb support for Persistence module (Scott Clasen)
- **FIX** - #518 refactor common storage backend to use bulk puts/gets where possible (Scott Clasen)
- **FIX** - #532 Prevent persistent datatypes with same uuid from corrupting a TX (Scott Clasen)
- **FIX** - #464 ThreadPoolBuilder should be rewritten to be an immutable builder (Viktor Klang)
- **FIX** - #449 Futures.awaitOne now uses onComplete listeners (Viktor Klang)
- **FIX** - #486 Fixed memory leak caused by Configgy that prevented full unload (Viktor Klang)
- **FIX** - #488 Fixed race condition in EBEDD restart (Viktor Klang)
- **FIX** - #492 Fixed race condition in Scheduler (Viktor Klang)
- **FIX** - #493 Switched to non-https repository for JBoss artifacts (Viktor Klang)
- **FIX** - #481 Exception when creating an actor now behaves properly when supervised (Viktor Klang)
- **FIX** - #498 Fixed no-op in supervision DSL (Viktor Klang)
- **FIX** - #491 reply and reply_? now sets a sender reference (Viktor Klang)
- **FIX** - #519 NotSerializableError when using Remote Typed Actors (Viktor Klang)
- **FIX** - #523 Message.toString is called all the time for incoming messages, expensive (Viktor Klang)
- **FIX** - #537 Make sure top folder is included in sources jar (Viktor Klang)
- **FIX** - #529 Remove Scala version number from Akka artifact ids (Viktor Klang)
- **FIX** - #533 Can't set Lifecycle from the Java API (Viktor Klang)
- **FIX** - #542 Make Future-returning Remote Typed Actor methods use onComplete (Viktor Klang)
- **FIX** - #479 Do not register listeners when CamelService is turned off by configuration (Martin Krasser)
- **FIX** - Fixed bug with finding TypedActor by type in ActorRegistry (Jonas Bonér)
- **FIX** - #515 race condition in FSM StateTimeout Handling (Irmo Manie)
- **UPD** - Akka package from "se.scalablesolutions.akka" to "akka" (Viktor Klang)
- **UPD** - Update Netty to 3.2.3.Final (Viktor Klang)
- **UPD** - #458 Camel to 2.5.0 (Martin Krasser)
- **UPD** - #458 Spring to 3.0.4.RELEASE (Martin Krasser)
- **UPD** - #458 Jetty to 7.1.6.v20100715 (Martin Krasser)
- **UPD** - Update to Scala 2.8.1 (Jonas Bonér)
- **UPD** - Changed remote server default port to 2552 (AKKA) (Jonas Bonér)
- **UPD** - Cleaned up and made remote protocol more efficient (Jonas Bonér)

- **UPD** - #528 RedisPersistentRef should not throw in case of missing key (Debasish Ghosh)
- **UPD** - #531 Fix RedisStorage add() method in Java API (Debasish Ghosh)
- **UPD** - #513 Implement snapshot based persistence control in SortedSet (Debasish Ghosh)
- **UPD** - #547 Update FSM docs (Irmo Manie)
- **UPD** - #548 Update AMQP docs (Irmo Manie)
- **REM** - Atmosphere integration, replace with Mist (Klang @ Evans)
- **REM** - JGroups integration, doesn't play with cloud services :/ (Viktor Klang)

7.2.7 Release 1.0-MILESTONE1

- **ADD** - Splitted akka-core up in akka-actor, akka-typed-actor & akka-remote (Jonas Bonér)
- **ADD** - Added meta-data to network protocol (Jonas Bonér)
- **ADD** - HotSwap and actor.become now uses a stack of PartialFunctions with API for pushing and popping the stack (Jonas Bonér)
- **ADD** - #440 Create typed actors with constructor args (Michael Kober)
- **ADD** - #322 Abstraction for unification of sender and senderFuture for later reply (Michael Kober)
- **ADD** - #364 Serialization for TypedActor proxy reference (Michael Kober)
- **ADD** - #423 Support configuration of Akka via Spring (Michael Kober)
- **FIX** - #426 UUID wrong for remote proxy for server managed actor (Michael Kober)
- **ADD** - #378 Support for server initiated remote TypedActor and UntypedActor in Spring config (Michael Kober)
- **ADD** - #194 Support for server-managed typed actor ||< Michael Kober ||
- **ADD** - #447 Allow Camel service to be turned off by configuration (Martin Krasser)
- **ADD** - #457 JavaAPI improvements for akka-camel (please read the migration guide) (Martin Krasser)
- **ADD** - #465 Dynamic message routing to actors (more ...) (Martin Krasser)
- **FIX** - #410 Use log configuration from config directory (Martin Krasser)
- **FIX** - #343 Some problems with persistent structures (Debasish Ghosh)
- **FIX** - #430 Refactor / re-implement MongoDB adapter so that it conforms to the guidelines followed in Redis and Cassandra modules (Debasish Ghosh)
- **FIX** - #436 ScalaJSON serialization does not map Int data types properly when used within a Map (Debasish Ghosh)
- **ADD** - #230 Update redisclient to be Redis 2.0 compliant (Debasish Ghosh)
- **FIX** - #435 Mailbox serialization does not retain messages (Debasish Ghosh)
- **ADD** - #445 Integrate type class based serialization of sjson into Akka (Debasish Ghosh)
- **FIX** - #480: Regression multibulk replies redis client (Debasish Ghosh)
- **FIX** - #415 Publish now generate source and doc jars (Viktor Klang)
- **FIX** - #420 REST endpoints should be able to be processed in parallel (Viktor Klang)
- **FIX** - #422 Dispatcher config should work for ThreadPoolBuilder-based dispatchers (Viktor Klang)
- **FIX** - #401 ActorRegistry should not leak memory (Viktor Klang)
- **FIX** - #250 Performance optimization for ExecutorBasedEventDrivenDispatcher (Viktor Klang)

- **FIX** - #419 Rename init and shutdown callbacks to preStart and postStop, and remove initTransactionalState (Viktor Klang)
- **FIX** - #346 Make max no of restarts (and within) are now both optional (Viktor Klang)
- **FIX** - #424 Actors self.supervisor not set by the time init() is called when started by startLink() (Viktor Klang)
- **FIX** - #427 spawnLink and startLink now has the same dispatcher semantics (Viktor Klang)
- **FIX** - #413 Actor shouldn't process more messages when waiting to be restarted (HawtDispatcher still does) (Viktor Klang)
- **FIX** - !! and !!! now do now not block the actor when used in remote actor (Viktor Klang)
- **FIX** - RemoteClient now reconnects properly (Viktor Klang)
- **FIX** - Logger.warn now properly works with varargs (Viktor Klang)
- **FIX** - #450 Removed ActorRef lifeCycle boilerplate: Some(LifeCycle(Permanent)) => Permanent (Viktor Klang)
- **FIX** - Moved ActorRef.trapExit into ActorRef.faultHandler and removed Option-boilerplate from fault-Handler (Viktor Klang)
- **FIX** - ThreadBasedDispatcher cheaper for idling actors, also benefits from all that is ExecutorBasedEvent-DrivenDispatcher (Viktor Klang)
- **FIX** - Fixing Futures.future, uses Actor.spawn under the hood, specify dispatcher to control where block is executed (Viktor Klang)
- **FIX** - #469 Akka "dist" now uses a root folder to avoid loitering if unzipped in a folder (Viktor Klang)
- **FIX** - Removed ScalaConfig, JavaConfig and rewrote Supervision configuration (Viktor Klang)
- **UPD** - Jersey to 1.3 (Viktor Klang)
- **UPD** - Atmosphere to 0.6.2 (Viktor Klang)
- **UPD** - Netty to 3.2.2.Final (Viktor Klang)
- **ADD** - Changed config file priority loading and added config modes. (Viktor Klang)
- **ADD** - #411 Bumped Jetty to v 7 and migrated to it's eclipse packages (Viktor Klang)
- **ADD** - #414 Migrate from Grizzly to Jetty for Akka Microkernel (Viktor Klang)
- **ADD** - #261 Add Java API for 'routing' module (Viktor Klang)
- **ADD** - #262 Add Java API for Agent (Viktor Klang)
- **ADD** - #264 Add Java API for Dataflow (Viktor Klang)
- **ADD** - Using JerseySimpleBroadcaster instead of JerseyBroadcaster in AkkaBroadcaster (Viktor Klang)
- **ADD** - #433 Throughput deadline added for ExecutorBasedEventDrivenDispatcher (Viktor Klang)
- **ADD** - Add possibility to set default cometSupport in akka.conf (Viktor Klang)
- **ADD** - #451 Added possibility to use akka-http as a standalone REST server (Viktor Klang)
- **ADD** - #446 Added support for Erlang-style receiveTimeout (Viktor Klang)
- **ADD** - #462 Added support for suspend/resume of processing individual actors mailbox, should give clearer restart semantics (Viktor Klang)
- **ADD** - #466 Actor.spawn now takes an implicit dispatcher to specify who should run the block (Viktor Klang)
- **ADD** - #456 Added map to Future and Futures.awaitMap (Viktor Klang)
- **REM** - #418 Remove Lift sample module and docs (Viktor Klang)
- **REM** - Removed all Reactor-based dispatchers (Viktor Klang)

- **REM** - Removed anonymous actor factories (Viktor Klang)
- **ADD** - Voldemort support for akka-persistence (Scott Clasen)
- **ADD** - HBase support for akka-persistence (David Greco)
- **ADD** - CouchDB support for akka-persistence (Yung-Luen Lan & Kahlen)
- **ADD** - #265 Java API for AMQP module (Irmo Manie)

7.2.8 Release 0.10 - Aug 21 2010

- **ADD** - Added new Actor type: UntypedActor for Java API (Jonas Bonér)
- **ADD** - #26 Deep serialization of Actor including its mailbox (Jonas Bonér)
- **ADD** - Rewritten network protocol. More efficient and cleaner. (Jonas Bonér)
- **ADD** - Rewritten Java Active Object tests into Scala to be able to run the in SBT. (Jonas Bonér)
- **ADD** - Added isDefinedAt method to Actor for checking if it can receive a certain message (Jonas Bonér)
- **ADD** - Added caching of Active Object generated class bytes, huge perf improvement (Jonas Bonér)
- **ADD** - Added RemoteClient Listener API (Jonas Bonér)
- **ADD** - Added methods to retrieve children from a Supervisor (Jonas Bonér)
- **ADD** - Rewritten Supervisor to become more clear and “correct” (Jonas Bonér)
- **ADD** - Added options to configure a blocking mailbox with custom capacity (Jonas Bonér)
- **ADD** - Added RemoteClient reconnection time window configuration option (Jonas Bonér)
- **ADD** - Added ActiveObjectContext with sender reference etc (Jonas Bonér)
- **ADD** - #293 Changed config format to JSON-style (Jonas Bonér)
- **ADD** - #302: Incorporate new ReceiveTimeout in Actor serialization (Jonas Bonér)
- **ADD** - Added Java API docs and made it comparable with Scala API docs. 1-1 mirroring (Jonas Bonér)
- **ADD** - Renamed Active Object to Typed Actor (Jonas Bonér)
- **ADD** - Enhanced Typed Actor: remoting, “real” restart upon failure etc. (Jonas Bonér)
- **ADD** - Typed Actor now inherits Actor and is a full citizen in the Actor world. (Jonas Bonér)
- **ADD** - Added support for remotely shutting down a remote actor (Jonas Bonér)
- **ADD** - #224 Add support for Camel in typed actors (more ...) (Martin Krasser)
- **ADD** - #282 Producer trait should implement Actor.receive (more...) (Martin Krasser)
- **ADD** - #271 Support for bean scope prototype in akka-spring (Johan Rask)
- **ADD** - Support for DI of values and bean references on target instance in akka-spring (Johan Rask)
- **ADD** - #287 Method annotated with @postrestart in ActiveObject is not called during restart (Johan Rask)
- **ADD** - Support for ApplicationContextAware in akka-spring (Johan Rask)
- **ADD** - #199 Support shutdown hook in TypedActor (Martin Krasser)
- **ADD** - #266 Access to typed actors from user-defined Camel routes (more ...) (Martin Krasser)
- **ADD** - #268 Revise akka-camel documentation (more ...) (Martin Krasser)
- **ADD** - #289 Support for <akka:camel-service> Spring configuration element (more ...) (Martin Krasser)
- **ADD** - #296 TypedActor lifecycle management (Martin Krasser)
- **ADD** - #297 Shutdown routes to typed actors (more ...) (Martin Krasser)
- **ADD** - #314 akka-spring to support typed actor lifecycle management (more ...) (Martin Krasser)

- **ADD** - #315 akka-spring to support configuration of shutdown callback method (more ...) (Martin Krasser)
- **ADD** - Fault-tolerant consumer actors and typed consumer actors (more ...) (Martin Krasser)
- **ADD** - #320 Leverage Camel's non-blocking routing engine (more ...) (Martin Krasser)
- **ADD** - #335 Producer trait should allow forwarding of results (Martin Krasser)
- **ADD** - #339 Redesign of Producer trait (pre/post processing hooks, async in-out) (more ...) (Martin Krasser)
- **ADD** - Non-blocking, asynchronous routing example for akka-camel (more ...) (Martin Krasser)
- **ADD** - #333 Allow applications to wait for endpoints being activated (more ...) (Martin Krasser)
- **ADD** - #356 Support @consume annotations on typed actor implementation class (Martin Krasser)
- **ADD** - #357 Support untyped Java actors as endpoint consumer (Martin Krasser)
- **ADD** - #366 CamelService should be a singleton (Martin Krasser)
- **ADD** - #392 Support untyped Java actors as endpoint producer (Martin Krasser)
- **ADD** - #393 Redesign CamelService singleton to be a CamelServiceManager (more ...) (Martin Krasser)
- **ADD** - #295 Refactoring Actor serialization to type classes (Debasish Ghosh)
- **ADD** - #317 Change documentation for Actor Serialization (Debasish Ghosh)
- **ADD** - #388 Typeclass serialization of ActorRef/UntypedActor isn't Java friendly (Debasish Ghosh)
- **ADD** - #292 Add scheduleOnce to Scheduler (Irmo Manie)
- **ADD** - #308 Initial receive timeout on actor (Irmo Manie)
- **ADD** - Redesign of AMQP module (more ...) (Irmo Manie)
- **ADD** - Added "become(behavior: Option[Receive])" to Actor (Viktor Klang)
- **ADD** - Added "find[T](f: PartialFunction[ActorRef,T]) : Option[T]" to ActorRegistry (Viktor Klang)
- **ADD** - #369 Possibility to configure dispatchers in akka.conf (Viktor Klang)
- **ADD** - #395 Create ability to add listeners to RemoteServer (Viktor Klang)
- **ADD** - #225 Add possibility to use Scheduler from TypedActor (Viktor Klang)
- **ADD** - #61 Integrate new persistent datastructures in Scala 2.8 (Peter Vlugter)
- **ADD** - Expose more of what Multiverse can do (Peter Vlugter)
- **ADD** - #205 STM transaction settings (Peter Vlugter)
- **ADD** - #206 STM transaction deferred and compensating (Peter Vlugter)
- **ADD** - #232 Expose blocking transactions (Peter Vlugter)
- **ADD** - #249 Expose Multiverse Refs for primitives (Peter Vlugter)
- **ADD** - #390 Expose transaction propagation level in multiverse (Peter Vlugter)
- **ADD** - Package objects for importing local/global STM (Peter Vlugter)
- **ADD** - Java API for the STM (Peter Vlugter)
- **ADD** - #379 Create STM Atomic templates for Java API (Peter Vlugter)
- **ADD** - #270 SBT plugin for Akka (Peter Vlugter)
- **ADD** - #198 support for ThreadBasedDispatcher in Spring config (Michael Kober)
- **ADD** - #377 support HawtDispatcher in Spring config (Michael Kober)
- **ADD** - #376 support Spring config for untyped actors (Michael Kober)
- **ADD** - #200 support WorkStealingDispatcher in Spring config (Michael Kober)
- **UPD** - #336 RabbitMQ 1.8.1 (Irmo Manie)

- **UPD** - #288 Netty to 3.2.1.Final (Viktor Klang)
- **UPD** - Atmosphere to 0.6.1 (Viktor Klang)
- **UPD** - Lift to 2.8.0-2.1-M1 (Viktor Klang)
- **UPD** - Camel to 2.4.0 (Martin Krasser)
- **UPD** - Spring to 3.0.3.RELEASE (Martin Krasser)
- **UPD** - Multiverse to 0.6 (Peter Vlugter)
- **FIX** - Fixed bug with stm not being enabled by default when no AKKA_HOME is set (Jonas Bonér)
- **FIX** - Fixed bug in network manifest serialization (Jonas Bonér)
- **FIX** - Fixed bug Remote Actors (Jonas Bonér)
- **FIX** - Fixed memory leak in Active Objects (Jonas Bonér)
- **FIX** - Fixed indeterministic deadlock in Transactor restart (Jonas Bonér)
- **FIX** - #325 Fixed bug in STM with dead hanging CountdownCommitBarrier (Jonas Bonér)
- **FIX** - #316: NoSuchElementException during ActiveObject restart (Jonas Bonér)
- **FIX** - #256: Tests for ActiveObjectContext (Jonas Bonér)
- **FIX** - Fixed bug in restart of Actors with 'Temporary' life-cycle (Jonas Bonér)
- **FIX** - #280 Tests fail if there is no akka.conf set (Jonas Bonér)
- **FIX** - #286 unwanted transitive dependencies from Geronimo project (Viktor Klang)
- **FIX** - Atmosphere comet comment to use stream instead of writer (Viktor Klang)
- **FIX** - #285 akka.conf is now used as defaults for Akka REST servlet init parameters (Viktor Klang)
- **FIX** - #321 fixed performance regression in ActorRegistry (Viktor Klang)
- **FIX** - #286 geronimo servlet 2.4 dep is no longer transitively loaded (Viktor Klang)
- **FIX** - #334 partial lift sample rewrite to fix breakage (Viktor Klang)
- **FIX** - Fixed a memory leak in ActorRegistry (Viktor Klang)
- **FIX** - Fixed a race-condition in Cluster (Viktor Klang)
- **FIX** - #355 Switched to Array instead of List on ActorRegistry return types (Viktor Klang)
- **FIX** - #352 ActorRegistry.actorsFor(class) now checks isAssignableFrom (Viktor Klang)
- **FIX** - Fixed a race condition in ActorRegistry.register (Viktor Klang)
- **FIX** - #337 Switched from Configgy logging to SLF4J, better for OSGi (Viktor Klang)
- **FIX** - #372 Scheduler now returns Futures to cancel tasks (Viktor Klang)
- **FIX** - #306 JSON serialization between remote actors is not transparent (Debasish Ghosh)
- **FIX** - #204 Reduce object creation in STM (Peter Vlugter)
- **FIX** - #253 Extend Multiverse BasicRef rather than wrap ProgrammaticRef (Peter Vlugter)
- **REM** - Removed pure POJO-style Typed Actor (old Active Object) (Jonas Bonér)
- **REM** - Removed Lift as a dependency for Akka-http (Viktor Klang)
- **REM** - #294 Remove `reply` and `reply_?` from Actor (Viktor Klang)
- **REM** - Removed one field in Actor, should be a minor memory reduction for high actor quantities (Viktor Klang)
- **FIX** - #301 DI does not work in akka-spring when specifying an interface (Johan Rask)
- **FIX** - #328 trapExit should pass through self with Exit to supervisor (Irmo Manie)

- **FIX** - Fixed warning when deregistering listeners (Martin Krasser)
- **FIX** - Added camel-jetty-2.4.0.1 to Akka's embedded-repo. (fixes a concurrency bug in camel-jetty-2.4.0, to be officially released in Camel 2.5.0) (Martin Krasser)
- **FIX** - #338 RedisStorageBackend fails when redis closes connection to idle client (Debasish Ghosh)
- **FIX** - #340 RedisStorage Map.get does not throw exception when disconnected from redis but returns None (Debasish Ghosh)

7.2.9 Release 0.9 - June 2th 2010

- **ADD** - Serializable, immutable, network-aware ActorRefs (Jonas Bonér)
- **ADD** - Optionally JTA-aware STM transactions (Jonas Bonér)
- **ADD** - Rewritten supervisor management, making use of ActorRef, now really kills the Actor instance and replaces it (Jonas Bonér)
- **ADD** - Allow linking and unlinking a declaratively configured Supervisor (Jonas Bonér)
- **ADD** - Remote protocol rewritten to allow passing along sender reference in all situations (Jonas Bonér)
- **ADD** - #37 API for JTA usage (Jonas Bonér)
- **ADD** - Added user accessible 'sender' and 'senderFuture' references (Jonas Bonér)
- **ADD** - Sender actor is now passed along for all message send functions (!, !!, forward) (Jonas Bonér)
- **ADD** - Subscription API for listening to RemoteClient failures (Jonas Bonér)
- **ADD** - Implemented link/unlink for ActiveObjects || Jan Kronquist / Michael Kober ||
- **ADD** - Added alter method to TransactionalRef + added appl(initValue) to Transactional Map/Vector/Ref (Peter Vlugter)
- **ADD** - Load dependency JARs in JAR deployed in kernel's ./deploy dir (Jonas Bonér)
- **ADD** - Allowing using Akka without specifying AKKA_HOME or path to akka.conf config file (Jonas Bonér)
- **ADD** - Redisclient now supports PubSub (Debasish Ghosh)
- **ADD** - Added a sample project under akka-samples for Redis PubSub using Akka actors (Debasish Ghosh)
- **ADD** - Richer API for Actor.reply (Viktor Klang)
- **ADD** - Added Listeners to Akka patterns (Viktor Klang)
- **ADD** - #183 Deactivate endpoints of stopped consumer actors (Martin Krasser)
- **ADD** - Camel Message API improvements (Martin Krasser)
- **ADD** - #83 Send notification to parent supervisor if all actors supervised by supervisor has been permanently killed (Jonas Bonér)
- **ADD** - #121 Make it possible to dynamically create supervisor hierarchies for Active Objects (Michael Kober)
- **ADD** - #131 Subscription API for node joining & leaving cluster (Jonas Bonér)
- **ADD** - #145 Register listener for errors in RemoteClient/RemoteServer (Jonas Bonér)
- **ADD** - #146 Create an additional distribution with sources (Jonas Bonér)
- **ADD** - #149 Support loading JARs from META-INF/lib in JARs put into the ./deploy directory (Jonas Bonér)
- **ADD** - #166 Implement insertVectorStorageEntriesFor in CassandraStorageBackend (Jonas Bonér)
- **ADD** - #168 Separate ID from Value in Actor; introduce ActorRef (Jonas Bonér)

- **ADD** - #174 Create sample module for remote actors (Jonas Bonér)
- **ADD** - #175 Add new sample module with Peter Vlugter's Ant demo (Jonas Bonér)
- **ADD** - #177 Rewrite remote protocol to make use of new ActorRef (Jonas Bonér)
- **ADD** - #180 Make use of ActorRef indirection for fault-tolerance management (Jonas Bonér)
- **ADD** - #184 Upgrade to Netty 3.2.0.CR1 (Jonas Bonér)
- **ADD** - #185 Rewrite Agent and Supervisor to work with new ActorRef (Jonas Bonér)
- **ADD** - #188 Change the order of how the akka.conf is detected (Jonas Bonér)
- **ADD** - #189 Reintroduce 'sender: Option[Actor]' ref in Actor (Jonas Bonér)
- **ADD** - #203 Upgrade to Scala 2.8 RC2 (Jonas Bonér)
- **ADD** - #222 Using Akka without AKKA_HOME or akka.conf (Jonas Bonér)
- **ADD** - #234 Add support for injection and management of ActiveObjectContext with RTTI such as 'sender' and 'senderFuture' references etc. (Jonas Bonér)
- **ADD** - #236 Upgrade SBinary to Scala 2.8 RC2 (Jonas Bonér)
- **ADD** - #235 Problem with RedisStorage.getVector(..) data structure storage management (Jonas Bonér)
- **ADD** - #239 Upgrade to Camel 2.3.0 (Martin Krasser)
- **ADD** - #242 Upgraded to Scala 2.8 RC3 (Jonas Bonér)
- **ADD** - #243 Upgraded to Protobuf 2.3.0 (Jonas Bonér)
- **ADD** - Added option to specify class loader when de-serializing messages and RemoteActorRef in Remote-Client (Jonas Bonér)
- **ADD** - #238 Upgrading to Cassandra 0.6.1 (Jonas Bonér)
- **ADD** - Upgraded to Jersey 1.2 (Viktor Klang)
- **ADD** - Upgraded Atmosphere to 0.6-SNAPSHOT, adding WebSocket support (Viktor Klang)
- **FIX** - Simplified ActiveObject configuration (Michael Kober)
- **FIX** - #237 Upgrade Mongo Java driver to 1.4 (the latest stable release) (Debasish Ghosh)
- **FIX** - #165 Implemented updateVectorStorageEntryFor in Mongo persistence module (Debasish Ghosh)
- **FIX** - #154: Allow ActiveObjects to use the default timeout in config file (Michael Kober)
- **FIX** - Active Object methods with @inittransactionalstate should be invoked automatically (Michael Kober)
- **FIX** - Nested supervisor hierarchy failure propagation bug fixed (Jonas Bonér)
- **FIX** - Fixed bug on CommitBarrier transaction registration (Jonas Bonér)
- **FIX** - Merged many modules to reduce total number of modules (Viktor Klang)
- **FIX** - Future parameterized (Viktor Klang)
- **FIX** - #191: Workstealing dispatcher didn't work with !! (Viktor Klang)
- **FIX** - #202: Allow applications to disable stream-caching (Martin Krasser)
- **FIX** - #119 Problem with Cassandra-backed Vector (Jonas Bonér)
- **FIX** - #147 Problem replying to remote sender when message sent with ! (Jonas Bonér)
- **FIX** - #171 initial value of Ref can become null if first transaction rolled back (Jonas Bonér)
- **FIX** - #172 Fix "broken" Protobuf serialization API (Jonas Bonér)
- **FIX** - #173 Problem with Vector::slice in CassandraStorage (Jonas Bonér)
- **FIX** - #190 RemoteClient shutdown ends up in endless loop (Jonas Bonér)

- **FIX** - #211 Problem with getting CommitBarrierOpenException when using Transaction.Global (Jonas Bonér)
- **FIX** - #240 Supervised actors not started when starting supervisor (Jonas Bonér)
- **FIX** - Fixed problem with Transaction.Local not committing to persistent storage (Jonas Bonér)
- **FIX** - #215: Re-engineered the JAX-RS support (Viktor Klang)
- **FIX** - Many many bug fixes || Team ||
- **REM** - Shoal cluster module (Viktor Klang)

7.2.10 Release 0.8.1 - April 6th 2010

- **ADD** - Redis cluster support (Debasish Ghosh)
- **ADD** - Reply to remote sender from message set with ! (Jonas Bonér)
- **ADD** - Load-balancer which prefers actors with few messages in mailbox || Jan Van Besien ||
- **ADD** - Added developer mailing list: [akka-dev AT googlegroups DOT com] (Jonas Bonér)
- **FIX** - Separated thread-local from thread-global transaction API (Jonas Bonér)
- **FIX** - Fixed bug in using STM outside Actors (Jonas Bonér)
- **FIX** - Fixed bug in anonymous actors (Jonas Bonér)
- **FIX** - Moved web initializer to new akka-servlet module (Viktor Klang)

7.2.11 Release 0.8 - March 31st 2010

- **ADD** - Scala 2.8 based (Viktor Klang)
- **ADD** - Monadic API for Agents (Jonas Bonér)
- **ADD** - Agents are transactional (Jonas Bonér)
- **ADD** - Work-stealing dispatcher || Jan Van Besien ||
- **ADD** - Improved Spring integration (Michael Kober)
- **FIX** - Various bugfixes || Team ||
- **FIX** - Improved distribution packaging (Jonas Bonér)
- **REMOVE** - Actor.send function (Jonas Bonér)

7.2.12 Release 0.7 - March 21st 2010

- **ADD** - Rewritten STM now works generically with fire-forget message flows (Jonas Bonér)
- **ADD** - Apache Camel integration (Martin Krasser)
- **ADD** - Spring integration (Michael Kober)
- **ADD** - Server-managed Remote Actors (Jonas Bonér)
- **ADD** - Clojure-style Agents (Viktor Klang)
- **ADD** - Shoal cluster backend (Viktor Klang)
- **ADD** - Redis-based transactional queue storage backend (Debasish Ghosh)
- **ADD** - Redis-based transactional sorted set storage backend (Debasish Ghosh)
- **ADD** - Redis-based atomic INC (index) operation (Debasish Ghosh)

- **ADD** - Distributed Comet (Viktor Klang)
- **ADD** - Project moved to SBT (simple-build-tool) || Peter Hausel ||
- **ADD** - Futures object with utility methods for Future's (Jonas Bonér)
- **ADD** - !!! function that returns a Future (Jonas Bonér)
- **ADD** - Richer ActorRegistry API (Jonas Bonér)
- **FIX** - Improved event-based dispatcher performance with 40% || Jan Van Besien ||
- **FIX** - Improved remote client pipeline performance (Viktor Klang)
- **FIX** - Support several Clusters on the same network (Viktor Klang)
- **FIX** - Structural package refactoring (Jonas Bonér)
- **FIX** - Various bugs fixed || Team ||

7.2.13 Release 0.6 - January 5th 2010

- **ADD** - Clustered Comet using Akka remote actors and clustered membership API (Viktor Klang)
- **ADD** - Cluster membership API and implementation based on JGroups (Viktor Klang)
- **ADD** - Security module for HTTP-based authentication and authorization (Viktor Klang)
- **ADD** - Support for using Scala XML tags in RESTful Actors (scala-jersey) (Viktor Klang)
- **ADD** - Support for Comet Actors using Atmosphere (Viktor Klang)
- **ADD** - MongoDB as Akka storage backend (Debasish Ghosh)
- **ADD** - Redis as Akka storage backend (Debasish Ghosh)
- **ADD** - Transparent JSON serialization of Scala objects based on SJSON (Debasish Ghosh)
- **ADD** - Kerberos/SPNEGO support for Security module || Eckhart Hertzler ||
- **ADD** - Implicit sender for remote actors: Remote actors are able to use reply to answer a request || Mikael Höggqvist ||
- **ADD** - Support for using the Lift Web framework with Actors || Tim Perrett ||
- **ADD** - Added CassandraSession API (with socket pooling) wrapping Cassandra's Thrift API in Scala and Java APIs (Jonas Bonér)
- **ADD** - Rewritten STM, now integrated with Multiverse STM (Jonas Bonér)
- **ADD** - Added STM API for atomic {..} and run {..} orElse {..} (Jonas Bonér)
- **ADD** - Added STM retry (Jonas Bonér)
- **ADD** - AMQP integration; abstracted as actors in a supervisor hierarchy. Impl AMQP 0.9.1 (Jonas Bonér)
- **ADD** - Complete rewrite of the persistence transaction management, now based on Unit of Work and Multiverse STM (Jonas Bonér)
- **ADD** - Monadic API to TransactionalRef (use it in for-comprehension) (Jonas Bonér)
- **ADD** - Lightweight actor syntax using one of the Actor.actor(..) methods. F.e: 'val a = actor { case _ => .. }' (Jonas Bonér)
- **ADD** - Rewritten event-based dispatcher which improved performance by 10x, now substantially faster than event-driven Scala Actors (Jonas Bonér)
- **ADD** - New Scala JSON parser based on sjson (Jonas Bonér)
- **ADD** - Added zlib compression to remote actors (Jonas Bonér)
- **ADD** - Added implicit sender reference for fire-forget ('!') message sends (Jonas Bonér)

- **ADD** - Monadic API to TransactionalRef (use it in for-comprehension) (Jonas Bonér)
- **ADD** - Smoother web app integration; just add akka.conf to the classpath (WEB-INF/classes), no need for AKKA_HOME or -Dakka.conf=.. (Jonas Bonér)
- **ADD** - Modularization of distribution into a thin core (actors, remoting and STM) and the rest in submodules (Jonas Bonér)
- **ADD** - Added 'forward' to Actor, forwards message but keeps original sender address (Jonas Bonér)
- **ADD** - JSON serialization for Java objects (using Jackson) (Jonas Bonér)
- **ADD** - JSON serialization for Scala objects (using SJSON) (Jonas Bonér)
- **ADD** - Added implementation for remote actor reconnect upon failure (Jonas Bonér)
- **ADD** - Protobuf serialization for Java and Scala objects (Jonas Bonér)
- **ADD** - SBinary serialization for Scala objects (Jonas Bonér)
- **ADD** - Protobuf as remote protocol (Jonas Bonér)
- **ADD** - Updated Cassandra integration and CassandraSession API to v0.4 (Jonas Bonér)
- **ADD** - CassandraStorage is now works with external Cassandra cluster (Jonas Bonér)
- **ADD** - ActorRegistry for retrieving Actor instances by class name and by id (Jonas Bonér)
- **ADD** - SchedulerActor for scheduling periodic tasks (Jonas Bonér)
- **ADD** - Now start up kernel with 'java -jar dist/akka-0.6.jar' (Jonas Bonér)
- **ADD** - Added Akka user mailing list: akka-user AT googlegroups DOT com]] (Jonas Bonér)
- **ADD** - Improved and restructured documentation (Jonas Bonér)
- **ADD** - New URL: <http://akkasource.org> (Jonas Bonér)
- **ADD** - New and much improved docs (Jonas Bonér)
- **ADD** - Enhanced trapping of failures: 'trapExit = List(classOf[..], classOf[..])' (Jonas Bonér)
- **ADD** - Upgraded to Netty 3.2, Protobuf 2.2, ScalaTest 1.0, Jersey 1.1.3, Atmosphere 0.4.1, Cassandra 0.4.1, Configgy 1.4 (Jonas Bonér)
- **FIX** - Lowered actor memory footprint; now an actor consumes ~600 bytes, which mean that you can create 6.5 million on 4 GB RAM (Jonas Bonér)
- **FIX** - Remote actors are now defined by their UUID (not class name) (Jonas Bonér)
- **FIX** - Fixed dispatcher bugs (Jonas Bonér)
- **FIX** - Cleaned up Maven scripts and distribution in general (Jonas Bonér)
- **FIX** - Fixed many many bugs and minor issues (Jonas Bonér)
- **FIX** - Fixed inconsistencies and ugliness in Actors API (Jonas Bonér)
- **REMOVE** - Removed concurrent mode (Jonas Bonér)
- **REMOVE** - Removed embedded Cassandra mode (Jonas Bonér)
- **REMOVE** - Removed the !? method in Actor (synchronous message send, since it's evil. Use !! with time-out instead. (Jonas Bonér)
- **REMOVE** - Removed startup scripts and lib dir (Jonas Bonér)
- **REMOVE** - Removed the 'Transient' life-cycle scope since to close to 'Temporary' in semantics. (Jonas Bonér)
- **REMOVE** - Removed 'Transient' Actors and restart timeout (Jonas Bonér)

7.3 Scaladoc API

7.3.1 Akka Snapshot

Automatically published Scaladoc API for the latest SNAPSHOT version of Akka can be found here:

- Akka - <http://akka.io/api/akka/snapshot>
- Akka Modules - <http://akka.io/api/akka-modules/snapshot>

7.3.2 Release Versions

1.2

- Akka 1.2 - <http://akka.io/api/akka/1.2/>
- Akka Modules 1.2 - <http://akka.io/api/akka-modules/1.2/>

1.1

- Akka 1.1 - <http://akka.io/api/akka/1.1/>
- Akka Modules 1.1 - <http://akka.io/api/akka-modules/1.1/>

1.0

- Akka 1.0 - <http://akka.io/api/1.0/>

7.4 Documentation for Other Versions

7.4.1 Akka Snapshot

Automatically published documentation for the latest SNAPSHOT version of Akka can be found here:

- Akka - <http://akka.io/docs/akka/snapshot/> (or in PDF format)
- Akka Modules - <http://akka.io/docs/akka-modules/snapshot/> (or in PDF format)

7.4.2 Release Versions

1.2

- Akka 1.2 - <http://akka.io/docs/akka/1.2/> (or in PDF format)
- Akka Modules 1.2 - <http://akka.io/docs/akka-modules/1.2/> (or in PDF format)

1.1

- Akka 1.1 - <http://akka.io/docs/akka/1.1/> (or in PDF format)
- Akka Modules 1.1 - <http://akka.io/docs/akka-modules/1.1/> (or in PDF format)

1.0

- Akka 1.0 - <http://akka.io/docs/akka-1.0/Home.html> (or in PDF format)

7.5 Issue Tracking

Akka is using Assembla as issue tracking system.

7.5.1 Browsing

Tickets

You can find the Akka tickets [here](#)

You can find the Akka Modules tickets [here](#)

Roadmaps

The roadmap for each Akka milestone is [here](#)

The roadmap for each Akka Modules milestone is [here](#)

7.5.2 Creating tickets

In order to create tickets you need to do the following:

[Register here](#) then log in

For Akka tickets: [Link to create new ticket](#)

For Akka Modules tickets: [Link to create new ticket](#)

Thanks a lot for reporting bugs and suggesting features.

7.5.3 Failing test

Please submit a failing test on the following format:

```
import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers

class Ticket001Spec extends WordSpec with MustMatchers {

  "An XXX" should {
    "do YYY" in {
      1 must be (1)
    }
  }
}
```

7.6 Licenses

7.6.1 Akka License

This software is licensed under the Apache 2 license, quoted below.

Copyright 2009–2011 Scalable Solutions AB <<http://scalablesolutions.se>>

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

7.6.2 Akka Committer License Agreement

All committers have signed this CLA

Based on: <http://www.apache.org/licenses/icla.txt>

Scalable Solutions AB
Individual Contributor License Agreement ("Agreement") V2.0
<http://www.scalablesolutions.se/licenses/>

Thank you for your interest in Akka, a Scalable Solutions AB (the "Company") Open Source project. In order to clarify the intellectual property license granted with Contributions from any person or entity, the Company must have a Contributor License Agreement ("CLA") on file that has been signed by each Contributor, indicating agreement to the license terms below. This license is for your protection as a Contributor as well as the protection of the Company and its users; it does not change your rights to use your own Contributions for any other purpose.

Full name: _____

Mailing Address: _____

Country: _____

Telephone: _____

Facsimile: _____

E-Mail: _____

You accept and agree to the following terms and conditions for Your present and future Contributions submitted to the Company. In return, the Company shall not use Your Contributions in a way that is contrary to the public benefit or inconsistent with its nonprofit

status and bylaws in effect at the time of the Contribution. Except for the license granted herein to the Company and recipients of software distributed by the Company, You reserve all right, title, and interest in and to Your Contributions.

1. Definitions.

"You" (or "Your") shall mean the copyright owner or legal entity authorized by the copyright owner that is making this Agreement with the Company. For legal entities, the entity making a Contribution and all other entities that control, are controlled by, or are under common control with that entity are considered to be a single Contributor. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"Contribution" shall mean any original work of authorship, including any modifications or additions to an existing work, that is intentionally submitted by You to the Company for inclusion in, or documentation of, any of the products owned or managed by the Company (the "Work"). For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Company or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Company for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by You as "Not a Contribution."

2. Grant of Copyright License. Subject to the terms and conditions of this Agreement, You hereby grant to the Company and to recipients of software distributed by the Company a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, sublicense, and distribute Your Contributions and such derivative works.
3. Grant of Patent License. Subject to the terms and conditions of this Agreement, You hereby grant to the Company and to recipients of software distributed by the Company a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by You that are necessarily infringed by Your Contribution(s) alone or by combination of Your Contribution(s) with the Work to which such Contribution(s) was submitted. If any entity institutes patent litigation against You or any other entity (including a cross-claim or counterclaim in a lawsuit) alleging that your Contribution, or the Work to which you have contributed, constitutes direct or contributory patent infringement, then any patent licenses granted to that entity under this Agreement for that Contribution or Work shall terminate as of the date such litigation is filed.
4. You agree that all Contributions are and will be given entirely voluntarily. Company will not be required to use, or to refrain from using, any Contributions that You, will not, absent a separate written agreement signed by Company, create any confidentiality obligation of Company, and Company has not

undertaken any obligation to treat any Contributions or other information You have given Company or will give Company in the future as confidential or proprietary information. Furthermore, except as otherwise provided in a separate subsequence written agreement between You and Company, Company will be free to use, disclose, reproduce, license or otherwise distribute, and exploit the Contributions as it sees fit, entirely without obligation or restriction of any kind on account of any proprietary or intellectual property rights or otherwise.

5. You represent that you are legally entitled to grant the above license. If your employer(s) has rights to intellectual property that you create that includes your Contributions, you represent that you have received permission to make Contributions on behalf of that employer, that your employer has waived such rights for your Contributions to the Company, or that your employer has executed a separate Corporate CLA with the Company.
6. You represent that each of Your Contributions is Your original creation (see section 7 for submissions on behalf of others). You represent that Your Contribution submissions include complete details of any third-party license or other restriction (including, but not limited to, related patents and trademarks) of which you are personally aware and which are associated with any part of Your Contributions.
7. You are not expected to provide support for Your Contributions, except to the extent You desire to provide support. You may provide support for free, for a fee, or not at all. Unless required by applicable law or agreed to in writing, You provide Your Contributions on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE.
8. Should You wish to submit work that is not Your original creation, You may submit it to the Company separately from any Contribution, identifying the complete details of its source and of any license or other restriction (including, but not limited to, related patents, trademarks, and license agreements) of which you are personally aware, and conspicuously marking the work as "Submitted on behalf of a third-party: [named here]".
9. You agree to notify the Company of any facts or circumstances of which you become aware that would make these representations inaccurate in any respect.
9. The validity of the interpretation of this Agreements shall be governed by, and constructed and enforced in accordance with, the laws of Sweden, applicable to the agreements made there (excluding the conflict of law rules). This Agreement embodies the entire agreement and understanding of the parties hereto and supersedes any and all prior agreements, arrangements and understandings relating to the matters provided for herein. No alteration, waiver, amendment changed or supplement hereto shall be binding more effective unless the same as set forth in writing signed by both parties.

Please sign: _____ Date: _____

7.6.3 Licenses for Dependency Libraries

Each dependency and its license can be seen in the project build file (the comment on the side of each dependency):
<https://github.com/jboner/akka/blob/master/project/build/AkkaProject.scala#L127>

7.7 Sponsors

7.7.1 YourKit

YourKit is kindly supporting open source projects with its full-featured Java Profiler. YourKit, LLC is the creator of innovative and intelligent tools for profiling Java and .NET applications. Take a look at YourKit's leading software products: [YourKit Java Profiler](#) and [YourKit .NET Profiler](#)

7.8 Support

Typesafe

7.9 Mailing List

[Akka User Google Group](#)

[Akka Developer Google Group](#)

7.10 Downloads

<http://akka.io/downloads/>

7.11 Source Code

Akka uses Git and is hosted at [Github](#).

- Akka: clone the Akka repository from <http://github.com/jboner/akka>
- Akka Modules: clone the Akka Modules repository from <http://github.com/jboner/akka-modules>

7.12 Maven Repository

The Akka Maven repository can be found at <http://akka.io/repository>.

Typesafe provides <http://repo.typesafe.com/typesafe/releases/> that proxies several other repositories, including akka.io. It is convenient to use the Typesafe repository, since it includes all external dependencies of Akka. It is a “best-effort” service, and if it is unavailable you may need to use the underlying repositories directly.

- <http://akka.io/repository>
- <http://repository.codehaus.org>
- <http://guiceyfruit.googlecode.com/svn/repo/releases/>
- <http://repository.jboss.org/nexus/content/groups/public/>
- <http://download.java.net/maven/2>

- <http://oss.sonatype.org/content/repositories/releases>
- <http://download.java.net/maven/glassfish>
- <http://databinder.net/repo>

ADDITIONAL INFORMATION

8.1 Add-on Modules

Akka Modules consist of add-on modules outside the core of Akka:

- akka-kernel-1.2.jar – Akka microkernel for running a bare-bones mini application server (embeds Jetty etc.)
- akka-amqp-1.2.jar – AMQP integration
- akka-camel-1.2.jar – Apache Camel Actors integration (it's the best way to have your Akka application communicate with the rest of the world)
- akka-camel-typed-1.2.jar – Apache Camel Typed Actors integration
- akka-scalaz-1.2.jar – Support for the Scalaz library
- akka-spring-1.2.jar – Spring framework integration
- akka-osgi-dependencies-bundle-1.2.jar – OSGi support

Documentation for Akka Modules is located [here](#).

8.2 Articles & Presentations

8.2.1 Videos

Functional Programming eXchange - March 2011

NE Scala - Feb 2011

JFokus - Feb 2011.

London Scala User Group - Oct 2010

Akka LinkedIn Tech Talk - Sept 2010

Akka talk at Scala Days - March 2010

Devoxx 2010 talk “Akka: Simpler Scalability, Fault-Tolerance, Concurrency” by Viktor Klang

8.2.2 Articles

Scatter-Gather with Akka Dataflow

Actor-Based Continuations with Akka and Swarm

Mimicking Twitter Using an Akka-Based Event-Driven Architecture

Remote Actor Class Loading with Akka

[Akka Producer Actors: New Features and Best Practices](#)
[Akka Consumer Actors: New Features and Best Practices](#)
[Compute Grid with Cloudy Akka](#)
[Clustered Actors with Cloudy Akka](#)
[Unit testing Akka Actors with the TestKit](#)
[Starting with Akka 1.0](#)
[Akka Does Async](#)
[CQRS with Akka actors and functional domain models](#)
[High Level Concurrency with JRuby and Akka Actors](#)
[Container-managed actor dispatchers](#)
[Even simpler scalability with Akka through RegistryActor](#)
[FSM in Akka \(in Vietnamese\)](#)
[Repeater and Idempotent Receiver implementation in Akka](#)
[EDA Akka as EventBus](#)
[Upgrading examples to Akka master \(0.10\) and Scala 2.8.0 Final](#)
[Testing Akka Remote Actor using Serializable.ProtoBuf](#)
[Flexible load balancing with Akka in Scala](#)
[Eventually everything, and actors](#)
[Join messages with Akka](#)
[Starting with Akka part 2, IntelliJ IDEA, Test Driven Development](#)
[Starting with Akka and Scala](#)
[PubSub using Redis and Akka Actors](#)
[Akka's grown-up hump](#)
[Akka features for application integration](#)
[Load Balancing Actors with Work Stealing Techniques](#)
[Domain Services and Bounded Context using Akka - Part 2](#)
[Thinking Asynchronous - Domain Modeling using Akka Transactors - Part 1](#)
[Introducing Akka – Simpler Scalability, Fault-Tolerance, Concurrency & Remoting through Actors](#)
[Using Cassandra with Scala and Akka](#)
[No Comet, Hacking with WebSocket and Akka](#)
[MongoDB for Akka Persistence](#)
[Pluggable Persistent Transactors with Akka](#)
[Enterprise scala actors: introducing the Akka framework](#)

8.2.3 Books

[Akka and Camel \(appendix E of Camel in Action\) Ett första steg i Scala \(Kapitel “Aktörer och Akka”\) \(en. “A first step in Scala”, chapter “Actors and Akka”, book in Swedish\)](#)

8.2.4 Presentations

Slides from Akka talk at Scala Days 2010, good short intro to Akka

Akka: Simpler Scalability, Fault-Tolerance, Concurrency & Remoting through Actors

http://ccombs.net/storage/presentations/Akka_High_Level_Abstractions.pdf

<https://github.com/deanwampler/Presentations/tree/master/akka-intro/>

8.2.5 Podcasts

Episode 16 – Scala and Akka an Interview with Jonas Bonér

Jonas Bonér on the Akka framework, Scala, and highly scalable applications

8.2.6 Interviews

JetBrains/DZone interview: Talking about Akka, Scala and life with Jonas Bonér

Artima interview of Jonas on Akka 1.0

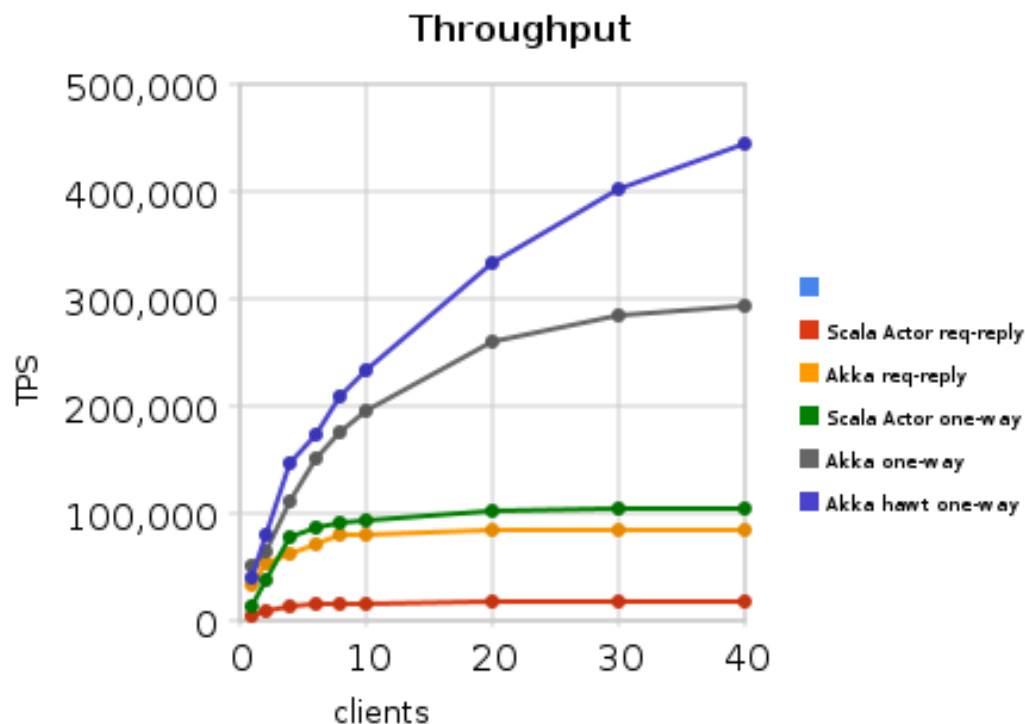
InfoQ interview of Jonas on Akka 1.0

InfoQ interview of Jonas on Akka 0.7

<http://jaxenter.com/we-ve-added-tons-of-new-features-since-0-10-33360.html>

8.3 Benchmarks

8.3.1 Scalability, Throughput and Latency benchmark



Simple Trading system.

- [Here is the result with some graphs](#)
- [Here is the article](#)
- [Here is the code](#)

Compares:

- Synchronous Scala solution
- Scala library Actors
 - Fire-forget
 - Request-reply
- Akka - Request-reply - Fire-forget with default dispatcher - Fire-forget with Hawt dispatcher

8.3.2 Performance benchmark

Benchmarking Akka against:

- Scala Library Actors
- Raw Java concurrency
- Jetlang (Java actors lib) <http://github.com/jboner/akka-bench>

8.4 Here is a list of recipes for all things Akka

- [PostStart => Link to Klangism](#)
- [Consumer actors best practices](#)
- [Producer actors best practices](#)

8.5 External Sample Projects

Here are some external sample projects created by Akka's users.

8.5.1 Camel in Action - Akka samples

Akka samples for the upcoming Camel in Action book by Martin Krasser. <http://code.google.com/p/camelinaction/source/browse/trunk/appendixE/>

8.5.2 CQRS impl using Scalaz and Akka

An implementation of CQRS using scalaz for functional domain models and Akka for event sourcing. <https://github.com/debasishg/cqrs-akka>

8.5.3 Example of using Comet with Akka Mist

<https://github.com/wrwills/AkkaMistComet>

8.5.4 Movie store

Code for a book on Scala/Akka. Showcasing Remote Actors. http://github.com/obcode/moviestore_akka

8.5.5 Estimating Pi with Akka

<http://www.earldouglas.com/estimating-pi-with-akka>

8.5.6 Running Akka on Android

Sample showing Dining Philosophers running in UI on Android. <https://github.com/gseitz/DiningAkkaDroids>
<http://www.vimeo.com/20303656>

8.5.7 Remote chat application using Java API

<https://github.com/mariofusco/akkachat>

8.5.8 Remote chat application using Java API

A sample chat application using the Java API for Akka. Port of the Scala API chat sample application in the Akka repository. https://github.com/abramsm/akka_chat_java

8.5.9 Sample parallel computing with Akka and Scala API

<https://github.com/yannart/ParallelPolynomialIntegral>

8.5.10 Akka, Facebook Graph API, WebGL sample

Showcasing Akka Mist HTTP module <https://github.com/buka/fbgl1>

8.5.11 Akka Mist Sample

<https://github.com/buka/akka-mist-sample>

8.5.12 Another Akka Mist Sample

<https://github.com/nppssk/akka-http-sbt>

8.5.13 Bank application

Showcasing Transactors and STM. <http://github.com/weiglewilczek/demo-akka>

8.5.14 Ant simulation 1

Traveling salesman problem. Inspired by Clojure's Ant demo. Uses SPDE for GUI. Idiomatic Scala/Akka code. Good example on how to use Actors and STM <http://github.com/pvlugter/ants>

8.5.15 Ant simulation 2

Traveling salesman problem. Close to straight port by Clojure's Ant demo. Uses Swing for GUI. Another nice example on how to use Actors and STM <http://github.com/azzoti/ScalaAkkaAnts>

8.5.16 The santa clause STM example by SPJ using Akka

<http://github.com/arjanblokzijl/akka-santa>

8.5.17 Akka trading system

<http://github.com/patriknw/akka-sample-trading>

8.5.18 Snowing version of Game of Life in Akka

<https://github.com/mariogleichmann/AkkaSamples/tree/master/src/main/scala/com/mgi/akka/gameoflife>

8.5.19 Akka Web (REST/Comet) template project

A sbt-based, scala Akka project that sets up a web project with REST and comet support
<http://github.com/mattbowen/akka-web-template>

8.5.20 Various samples on how to use Akka

From the May Chciago-Area Scala Enthusiasts Meeting <http://github.com/deanwampler/AkkaWebSampleExercise>

8.5.21 Absurd concept for a ticket sales & inventory system, using Akka framework

<http://github.com/bmjames/ticketfaster>

8.5.22 Akka sports book sample: Java API

<http://github.com/jrask/akka-activeobjects-application>

8.5.23 Sample of using the Finite State Machine (FSM) DSL

<http://github.com/ngocdaothanh/lock-fsm-akka>

8.5.24 Akka REST, Jetty, SBT template project

Great starting point for building an Akka application. <http://github.com/efleming969/akka-template-rest>

8.5.25 Samples of various Akka features (in Scala)

<http://github.com/efleming969/akka-samples> Fork at <http://github.com/flintobrien/akka-samples>

8.5.26 A sample sbt setup for running the akka-sample-chat

<http://github.com/dwhitney/sbt-akka-sample-chat>

8.5.27 Akka Benchmark project

Benches Akka against various other actors and concurrency tools <http://github.com/jboner/akka-bench>

8.5.28 Typed Actor (Java API) sample project

http://github.com/bobo/akka_sample_java

8.5.29 Akka PI calculation sample project

<http://github.com/bonnefoa/akkaPi/>

8.5.30 Akka Vaadin Ice sample

<https://github.com/tomhowe/vaadin-akka-ice-test>

8.5.31 Port of Jersey (JAX-RS) samples to Akka

<http://github.com/akollegger/akka-jersey-samples>

8.5.32 Akka Expect Testing

<https://github.com/joda/akka-expect>

8.5.33 Akka Java API playground

<https://github.com/koevet/akka-java-playground>

8.5.34 Family web page build with Scala, Lift, Akka, Redis, and Facebook Connect

<http://github.com/derekjw/williamsfamily>

8.5.35 An example of queued computation tasks using Akka

<http://github.com/derekjw/computation-queue-example>

8.5.36 The samples for the New York Scala Enthusiasts Meetup discussing Akka

<http://www.meetup.com/New-York-Scala-Enthusiasts/calendar/12315985/> http://github.com/dwhitney/akka_meetup

8.5.37 Container managed thread pools for Akka Dispatchers

<https://github.com/remeniuk/akka-cm-dispatcher>

8.5.38 “Lock” Finite State Machine demo with Akka

<http://github.com/ngocdaothanh/lock-fsm-akka>

8.5.39 Template w/ IntelliJ stuff for random akka playing around (with Bivvy)

<http://github.com/b3n00/akka10-template>

8.5.40 Akka chat using Akka Java API by Mario Fusco

<https://github.com/mariofusco/akkachat>

8.6 Projects using the removed Akka Persistence modules

8.6.1 Akka Terrastore sample

<https://github.com/dgreco/akka-terrastore-example>

8.6.2 Akka Persistence for Force.com

<https://github.com/sclasen/akka-persistence-force>

8.6.3 Template for Akka and Redis

<http://github.com/andrewmilkowski/template-akka-persistence-redis>

8.7 Companies and Open Source projects using Akka

8.7.1 Production Users

These are some of the production Akka users that are able to talk about their use publicly.

CSC

CSC is a global provider of information technology services. The Traffic Management business unit in the Netherlands is a systems integrator for the implementation of Traffic Information and Traffic Enforcement Systems, such as section control, weigh in motion, travel time and traffic jam detection and national data warehouse for traffic information. CSC Traffic Management is using Akka for their latest Traffic Information and Traffic Enforcement Systems.

http://www.csc.com/nl/ds/42449-traffic_management

“Akka has been in use for almost a year now (since 0.7) and has been used successfully for two projects so far. Akka has enabled us to deliver very flexible, scalable and high performing systems with as little friction as possible. The Actor model has simplified a lot of concerns in the type of systems that we build and is now part of our reference architecture. With Akka we deliver systems that meet the most strict performance requirements of our clients in a near-realtime environment. We have found the Akka framework and it’s support team invaluable.”

Thatcham Motor Insurance Repair Research Centre

Thatcham is a EuroNCAP member. They research efficient, safe, cost effective repair of vehicles, and work with manufacturers to influence the design of new vehicles. Thatcham are using Akka as the implementation for their distributed modules. All Scala based research software now talks to an Akka based publishing platform. Using Akka enables Thatcham to ‘free their domain’, and ensures that the platform is cloud enabled and scalable, and that the team is confident that they are flexible. Akka has been in use, tested under load at Thatcham for almost a year, with no problems migrating up through the different versions. An old website currently under redesign on a new Scala powered platform: www.thatcham.org

“We have been in production with Akka for over 18 months with zero downtime. The core is rock solid, never a problem, performance is great, integration capabilities are diverse and ever growing, and the toolkit is just a

pleasure to work with. Combine that with the excellent response you get from the devs and users on this list and you have a winner. Absolutely no regrets on our part for choosing to work with Akka.”

“Scala and Akka are now enabling improvements in the standard of vehicle damage assessment, and in the safety of vehicle repair across the UK, with Europe, USA, Asia and Australasia to follow. Thatcham (Motor Insurance Repair Research Centre) are delivering crash specific information with linked detailed repair information for over 7000 methods.

For Thatcham, the technologies enable scalability and elegance when dealing with complicated design constraints. Because of the complexity of interlinked methods, caching is virtually impossible in most cases, so in steps the ‘actors’ paradigm. Where previously something like JMS would have provided a stable but heavyweight, rigid solution, Thatcham are now more flexible, and can expand into the cloud in a far simpler, more rewarding way.

Thatcham’s customers, body shop repairers and insurers receive up to date repair information in the form of crash repair documents of the quality necessary to ensure that every vehicle is repaired back to the original safety standard. In a market as important as this, availability is key, as is performance. Scala and Akka have delivered consistently so far.

While recently introduced, growing numbers of UK repairers are receiving up to date repair information from this service, with the rest to follow shortly. Plans are already in motion to build new clusters to roll the service out across Europe, USA, Asia and Australasia.

The sheer opportunities opened up to teams by Scala and Akka, in terms of integration, concise expression of intent and scalability are of huge benefit.”

SVT (Swedish Television)

<http://svt.se>

“I’m currently working in a project at the Swedish Television where we’re developing a subtitling system with collaboration capabilities similar to Google Wave. It’s a mission critical system and the design and server implementation is all based on Akka and actors etc. We’ve been running in production for about 6 months and have been upgrading Akka whenever a new release comes out. We’ve never had a single bug due to Akka, and it’s been a pure pleasure to work with. I would choose Akka any day of the week!

Our system is highly asynchronous so the actor style of doing things is a perfect fit. I don’t know about how you feel about concurrency in a big system, but rolling your own abstractions is not a very easy thing to do. When using Akka you can almost forget about all that. Synchronizing between threads, locking and protecting access to state etc. Akka is not just about actors, but that’s one of the most pleasurable things to work with. It’s easy to add new ones and it’s easy to design with actors. You can fire up work actors tied to a specific dispatcher etc. I could make the list of benefits much longer, but I’m at work right now. I suggest you try it out and see how it fits your requirements.

We saw a perfect business reason for using Akka. It lets you concentrate on the business logic instead of the low level things. It’s easy to teach others and the business intent is clear just by reading the code. We didn’t chose Akka just for fun. It’s a business critical application that’s used in broadcasting. Even live broadcasting. We wouldn’t have been where we are today in such a short time without using Akka. We’re two developers that have done great things in such a short amount of time and part of this is due to Akka. As I said, it lets us focus on the business logic instead of low level things such as concurrency, locking, performance etc.”

Tapad

<http://tapad.com>

“Tapad is building a real-time ad exchange platform for advertising on mobile and connected devices. Real-time ad exchanges allows for advertisers (among other things) to target audiences instead of buying fixed set of ad slots that will be displayed “randomly” to users. To developers without experience in the ad space, this might seem boring, but real-time ad exchanges present some really interesting technical challenges.

Take for instance the process backing a page view with ads served by a real-time ad exchange auction (somewhat simplified):

1. A user opens a site (or app) which has ads in it.
2. As the page / app loads, the ad serving components fires off a request to the ad exchange (this might just be due to an image tag on the page).
3. The ad exchange enriches the request with any information about the current user (tracking cookies are often employed for this) and display context information (“news article about parenting”, “blog about food” etc).
4. The ad exchange forwards the enriched request to all bidders registered with the ad exchange.
5. The bidders consider the provided user information and responds with what price they are willing to pay for this particular ad slot.
6. The ad exchange picks the highest bidder and ensures that the winning bidder’s ad is shown to the user.

Any latency in this process directly influences user experience latency, so this has to happen really fast. All-in-all, the total time should not exceed about 100ms and most ad exchanges allow bidders to spend about 60ms (including network time) to return their bids. That leaves the ad exchange with less than 40ms to facilitate the auction. At Tapad, this happens billions of times per month / tens of thousands of times per second.

Tapad is building bidders which will participate in auctions facilitated by other ad exchanges, but we’re also building our own. We are using Akka in several ways in several parts of the system. Here are some examples:

Plain old parallelization During an auction in the real-time exchange, it’s obvious that all bidders must receive the bid requests in parallel. An auctioneer actor sends the bid requests to bidder actors which in turn handles throttling and eventually IO. We use futures in these requests and the auctioneer discards any responses which arrive too late.

Inside our bidders, we also rely heavily on parallel execution. In order to determine how much to pay for an ad slot, several data stores are queried for information pertinent to the current user. In a “traditional” system, we’d be doing this sequentially, but again, due to the extreme latency constraints, we’re doing this concurrently. Again, this is done with futures and data that is not available in time, get cut from the decision making (and logged :)).

Maintaining state under concurrent load This is probably the *de facto* standard use case for the actors model. Bidders internal to our system are actors backed by a advertiser campaign. A campaign includes, among other things, budget and “pacing” information. The budget determines how much money to spend for the duration of the campaign, whereas pacing information might set constraints on how quickly or slowly the money should be spent. Ad traffic changes from day to day and from hour to hour and our spending algorithms considers past performance in order to spend the right amount of money at the right time. Needless to say, these algorithms use a lot of state and this state is in constant flux. A bidder with a high budget may see tens of thousands of bid requests per second. Luckily, due to round-robin load-balancing and the predictability of randomness under heavy traffic, the bidder actors do not share state across cluster nodes, they just share their instance count so they know which fraction of the campaign budget to try to spend.

Pacing is also done for external bidders. Each 3rd party bidder end-point has an actor coordinating requests and measuring latency and throughput. The actor never blocks itself, but when an incoming bid request is received, it considers the current performance of the 3rd party system and decides whether to pass on the request and respond negatively immediately, or forward the request to the 3rd party request executor component (which handles the IO).

Batch processing We store a lot of data about every single ad request we serve and this is stored in a key-value data store. Due to the performance characteristics of the data store, it is not feasible to store every single data point one at a time - it must be batched up and performed in parallel. We don’t need a durable messaging system for this (losing a couple of hundred data points is no biggie). All our data logging happens asynchronously and we have a basic load-balanced actors which batches incoming messages and writes on regular intervals (using Scheduler) or whenever the specified batch size has been reached.

Analytics Needless to say, it’s not feasible / useful to store our traffic information in a relational database. A lot of analytics and data analysis is done “offline” with map / reduce on top the data store, but this doesn’t work well for real-time analytics which our customers love. We therefore have metrics actors that receives campaign bidding and click / impression information in real-time, aggregates this information over configurable periods of time and flushes it to the database used for customer dashboards for “semi-real-time” display. Five minute history is considered real-time in this business, but in theory, we could have queried the actors directly for really real-time data. :)

Our Akka journey started as a prototyping project, but Akka has now become a crucial part of our system. All of the above mentioned components, except the 3rd party bidder integration, have been running in production for a couple of weeks (on Akka 1.0RC3) and we have not seen any issues at all so far.”

Flowdock

Flowdock delivers Google Wave for the corporate world.

“Flowdock makes working together a breeze. Organize the flow of information, task things over and work together towards common goals seamlessly on the web - in real time.”

<http://flowdock.com/>

Travel Budget

<http://labs.inevo.pt/travel-budget>

Says.US

“says.us is a gathering place for people to connect in real time - whether an informal meeting of people who love Scala or a chance for people anywhere to speak out about the latest headlines.”

<http://says.us/>

LShift

- *“Diffa is an open source data analysis tool that automatically establishes data differences between two or more real-time systems.*
- Diffa will help you compare local or distributed systems for data consistency, without having to stop them running or implement manual cross-system comparisons. The interface provides you with simple visual summary of any consistency breaks and tools to investigate the issues.*
- Diffa is the ideal tool to use to investigate where or when inconsistencies are occurring, or simply to provide confidence that your systems are running in perfect sync. It can be used operationally as an early warning system, in deployment for release verification, or in development with other enterprise diagnosis tools to help troubleshoot faults.”*

<http://diffa.lshift.net/>

Twimpact

“Real-time twitter trends and user impact”

<http://twimpact.com>

Rocket Pack Platform

“Rocket Pack Platform is the only fully integrated solution for plugin-free browser game development.”

<http://rocketpack.fi/platform/>

8.7.2 Open Source Projects using Akka

Redis client

A Redis client written Scala, using Akka actors, HawtDispath and non-blocking IO. Supports Redis 2.0+

<http://github.com/derekjw/fyrie-redis>

Narrator

“Narrator is a library which can be used to create story driven clustered load-testing packages through a very readable and understandable api.”

<http://github.com/shorrockin/narrator>

Kandash

“Kandash is a lightweight kanban web-based board and set of analytics tools.”

<http://vasilrem.com/blog/software-development/kandash-project-v-0-3-is-now-available/>

<http://code.google.com/p/kandash/>

Wicket Cassandra Datastore

This project provides an `org.apache.wicket.pageStore.IDataStore` implementation that writes pages to an Apache Cassandra cluster using Akka.

<http://github.com/gseitz/wicket-cassandra-datastore/>

Spray

“spray is a lightweight Scala framework for building RESTful web services on top of Akka actors and Akka Mist. It sports the following main features:

- *Completely asynchronous, non-blocking, actor-based request processing for efficiently handling very high numbers of concurrent connections*
- *Powerful, flexible and extensible internal Scala DSL for declaratively defining your web service behavior*
- *Immutable model of the HTTP protocol, decoupled from the underlying servlet container*
- *Full testability of your REST services, without the need to fire up containers or actors”*

<https://github.com/spray/spray/wiki>

8.8 Third-party Integrations

8.8.1 The Play! Framework

Dustin Whitney has done an Akka integration module for the Play! framework.

Detailed instructions here: <http://github.com/dwhitney/akka/blob/master/README.textile>.

There are three screencasts:

- Using Play! with Akka STM: <http://vimeo.com/10764693>
- Using Play! with Akka Actors: <http://vimeo.com/10792173>
- Using Play! with Akka Remote Actors: <http://vimeo.com/10793443>

8.8.2 The Pinky REST/MVC Framework

Peter Hausel has done an Akka integration module for the [Pinky](#) framework.

Read more here: <http://wiki.github.com/pk11/pinky/release-13>

8.9 Other Language Bindings

8.9.1 JRuby

High level concurrency using Akka actors and JRuby.

<https://github.com/danielribeiro/RubyOnAkka>

If you are using STM with JRuby then you need to unwrap the Multiverse control flow exception as follows:

```
begin
  ... atomic stuff
rescue NativeException => e
  raise e.cause if e.cause.java_class.package.name.include? "org.multiverse"
end
```

8.9.2 Groovy/Groovy++

<https://gist.github.com/620439>

8.10 Feature Stability Matrix

Akka is comprised of a number of modules, with different levels of maturity and in different parts of their lifecycle, the matrix below gives you the current stability level of the modules.

8.10.1 Explanation of the different levels of stability

- **Solid** - Proven solid in heavy production usage
- **Stable** - Ready for use in production environment
- **In progress** - Not enough feedback/use to claim it's ready for production use

Feature	Solid	Stable	In progress
Actors (Scala)	Solid		
Actors (Java)	Solid		
Typed Actors (Scala)	Solid		
Typed Actors (Java)	Solid		
STM (Scala)	Solid		
STM (Java)	Solid		
Transactors (Scala)	Solid		
Transactors (Java)	Solid		
Remote Actors (Scala)	Solid		
Remote Actors (Java)	Solid		
Camel	Solid		
AMQP	Solid		
HTTP	Solid		
Integration Guice		Stable	
Integration Spring		Stable	
Scheduler	Solid		
Redis Pub Sub			In progress

LINKS

- *Migration Guides*
- Downloads
- Source Code
- *Scaladoc API*
- *Documentation for Other Versions*
- Akka Modules Documentation
- *Issue Tracking*
- *Support*

PYTHON MODULE INDEX

a

`akka-testkit`, 162

f

FSM (*Scala*), 139

INDEX

A

akka-testkit (module), [162](#)

F

FSM (module), [139](#)