Akka Documentation

Release 2.0.4

Typesafe Inc

November 14, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is Akka?

Scalable real-time transaction processing

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 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" 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

Actors give 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 (approximately 2.7 million actors per GB RAM).

See *Actors (Scala)* and *Actors (Java)*

Fault Tolerance

Fault tolerance through supervisor hierarchies with "let-it-crash" semantics. Excellent for writing highly fault-tolerant systems that never stop, systems that self-heal. Supervisor hierarchies can span over multiple JVMs to provide truly fault-tolerant systems.

See *Fault Tolerance (Scala)* and *Fault Tolerance (Java)*

Location Transparency

Everything in Akka is designed to work in a distributed environment: all interactions of actors use purely message passing and everything is asynchronous.

For an overview of the remoting see *Location Transparency*

Transactors

Transactors combine actors and STM (Software Transactional Memory) into transactional actors. It allows you to compose atomic message flows with automatic retry and rollback.

See *Transactors (Scala)* and *Transactors (Java)*

1.1.2 Scala and Java APIs

Akka has both a *Scala API* and a *Java API*.

1.1.3 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 to drop your application into.

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

1.1.4 What happened to Cloudy Akka?

The commercial offering was earlier referred to as Cloudy Akka. This offering consisted of two things:

- Cluster support for Akka
- Monitoring & Management (formerly called Atmos)

Cloudy Akka have been discontinued and the Cluster support is now being moved into the Open Source version of Akka (the upcoming Akka 2.1), while the Monitoring & Management (Atmos) is now rebranded into Typesafe Console and is part of the commercial subscription for the Typesafe Stack (see below for details).

1.1.5 Typesafe Stack

Akka is now also part of the Typesafe Stack.

The Typesafe Stack is a modern software platform that makes it easy for developers to build scalable software applications. It combines the Scala programming language, Akka, the Play! web framework and robust developer tools in a simple package that integrates seamlessly with existing Java infrastructure.

The Typesafe Stack is all fully open source.

1.1.6 Typesafe Console

On top of the Typesafe Stack we also have commercial product called Typesafe Console which provides the following features:

1. Slick Web UI with real-time view into the system
2. Management through Dashboard, JMX and REST
3. Dapper-style tracing of messages across components and remote nodes
4. Real-time statistics
5. Very low overhead monitoring agents (should always be on in production)
6. Consolidation of statistics and logging information to a single node
7. Storage of statistics data for later processing
8. Provisioning and rolling upgrades

Read more here.

1.2 Why Akka?

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

Akka provides scalable real-time transaction processing.

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 users to choose the right tool for the job.

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

We see Akka being adopted by many large organizations in a big range of industries all from investment and merchant banking, retail and social media, simulation, gaming and betting, automobile and traffic systems, health care, data analytics and much more. Any system that have the need for high-throughput and low latency is a good candidate for using Akka.

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).

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

All this in the ApacheV2-licensed open source project.

1.3 Getting Started

1.3.1 Prerequisites

Akka requires that you have Java 1.6 or later installed on you machine.

1.3.2 Getting Started Guides and Template Projects

The best way to start learning Akka is to download the Typesafe Stack and either try out the Akka Getting Started Tutorials or check out one of Akka Template Projects. Both comes in several flavours depending on your development environment preferences.

- Download Typesafe Stack
- Getting Started Tutorials
- Template Projects

1.3.3 Download

There are several ways to download Akka. You can download it as part of the Typesafe Stack (as described above). You can download the full distribution with microkernel, which includes all modules. Or you can use a build tool like Maven or SBT to download dependencies from the Akka Maven repository.

1.3.4 Modules

Akka is very modular and has many JARs for containing different features.

- `akka-actor-2.0.4.jar` – Standard Actors, Typed Actors and much more
- `akka-remote-2.0.4.jar` – Remote Actors
- `akka-slf4j-2.0.4.jar` – SLF4J Event Handler Listener
- `akka-testkit-2.0.4.jar` – Toolkit for testing Actors
- `akka-kernel-2.0.4.jar` – Akka microkernel for running a bare-bones mini application server
- `akka-<storage-system>-mailbox-2.0.4.jar` – Akka durable mailboxes

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).

1.3.5 Using a release distribution

Download the release you need from http://akka.io/downloads and unzip it.

1.3.6 Using a snapshot version

The Akka nightly snapshots are published to http://akka.io/snapshots/ and are versioned with both `SNAPSHOT` and timestamps. You can choose a timestamped version to work with and can decide when to update to a newer version. The Akka snapshots repository is also proxied through http://repo.typesafe.com/typesafe/snapshots/ which includes proxies for several other repositories that Akka modules depend on.

1.3.7 Microkernel

The Akka distribution includes the microkernel. To run the microkernel put your application jar in the `deploy` directory and use the scripts in the `bin` directory.

More information is available in the documentation of the microkernel (*Microkernel (Java)*, *Microkernel (Scala)*).

1.3.8 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/realeses/ and Typesafe provides http://repo.typesafe.com/typesafe/releases/ that proxies several other repositories, including akka.io.

1.3.9 Using Akka with Maven

The simplest way to get started with Akka and Maven is to check out the Akka/Maven template project.

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 2.0.4:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor</artifactId>
  <version>2.0.4</version>
</dependency>
```

Note: for snapshot versions both `SNAPSHOT` and timestamped versions are published.

1.3.10 Using Akka with SBT

The simplest way to get started with Akka and SBT is to check out the Akka/SBT template project.

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 += "com.typesafe.akka" % "akka-actor" % "2.0.4"
```

1.3.11 Using Akka with Eclipse

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

1.3.12 Using Akka with IntelliJ IDEA

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

1.3.13 Build from sources

Akka uses Git and is hosted at Github.

- Akka: clone the Akka repository from http://github.com/akka/akka

Continue reading the page on *Building Akka*

1.3.14 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 Use-case and Deployment Scenarios

1.4.1 How can I use and deploy Akka?

Akka can be used in 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 stand alone application by instantiating ActorSystem in a main class or using the microkernel

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.

Using Akka as a stand alone microkernel

Akka can also be run as a stand-alone microkernel. For more information see *Microkernel (Java)* / *Microkernel (Scala)*.

1.5 Examples of use-cases for Akka

We see Akka being adopted by many large organizations in a big range of industries all from investment and merchant banking, retail and social media, simulation, gaming and betting, automobile and traffic systems, health care, data analytics and much more. Any system that have the need for high-throughput and low latency is a good candidate for using Akka.

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

1.5.1 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

CHAPTER

TWO

GENERAL

2.1 Actor Systems

Actors are objects which encapsulate state and behavior, they communicate exclusively by exchanging messages which are placed into the recipient's mailbox. In a sense, actors are the most stringent form of object-oriented programming, but it serves better to view them as persons: while modeling a solution with actors, envision a group of people and assign sub-tasks to them, arrange their functions into an organizational structure and think about how to escalate failure (all with the benefit of not actually dealing with people, which means that we need not concern ourselves with their emotional state or moral issues). The result can then serve as a mental scaffolding for building the software implementation.

2.1.1 Hierarchical Structure

Like in an economic organization, actors naturally form hierarchies. One actor, which is to oversee a certain function in the program might want to split up its task into smaller, more manageable pieces. For this purpose it starts child actors which it supervises. While the details of supervision are explained *here*, we shall concentrate on the underlying concepts in this section. The only prerequisite is to know that each actor has exactly one supervisor, which is the actor that created it.

The quintessential feature of actor systems is that tasks are split up and delegated until they become small enough to be handled in one piece. In doing so, not only is the task itself clearly structured, but the resulting actors can be reasoned about in terms of which messages they should process, how they should react nominally and how failure should be handled. If one actor does not have the means for dealing with a certain situation, it sends a corresponding failure message to its supervisor, asking for help. The recursive structure then allows to handle failure at the right level.

Compare this to layered software design which easily devolves into defensive programming with the aim of not leaking any failure out: if the problem is communicated to the right person, a better solution can be found than if trying to keep everything "under the carpet".

Now, the difficulty in designing such a system is how to decide who should supervise what. There is of course no single best solution, but there are a few guide lines which might be helpful:

- If one actor manages the work another actor is doing, e.g. by passing on sub-tasks, then the manager should supervise the child. The reason is that the manager knows which kind of failures are expected and how to handle them.
- If one actor carries very important data (i.e. its state shall not be lost if avoidable), this actor should source out any possibly dangerous sub-tasks to children it supervises and handle failures of these children as appropriate. Depending on the nature of the requests, it may be best to create a new child for each request, which simplifies state management for collecting the replies. This is known as the "Error Kernel Pattern" from Erlang.
- If one actor depends on another actor for carrying out its duty, it should watch that other actor's liveness and act upon receiving a termination notice. This is different from supervision, as the watching party has

no influence on the supervisor strategy, and it should be noted that a functional dependency alone is not a criterion for deciding where to place a certain child actor in the hierarchy.

There are of course always exceptions to these rules, but no matter whether you follow the rules or break them, you should always have a reason.

2.1.2 Configuration Container

The actor system as a collaborating ensemble of actors is the natural unit for managing shared facilities like scheduling services, configuration, logging, etc. Several actor systems with different configuration may co-exist within the same JVM without problems, there is no global shared state within Akka itself. Couple this with the transparent communication between actor systems—within one node or across a network connection—to see that actor systems themselves can be used as building blocks in a functional hierarchy.

2.1.3 Actor Best Practices

1. Actors should be like nice co-workers: do their job efficiently without bothering everyone else needlessly and avoid hogging resources. Translated to programming this means to process events and generate responses (or more requests) in an event-driven manner. Actors should not block (i.e. passively wait while occupying a Thread) on some external entity, which might be a lock, a network socket, etc. The blocking operations should be done in some special-cased thread which sends messages to the actors which shall act on them.
2. Do not pass mutable objects between actors. In order to ensure that, prefer immutable messages. If the encapsulation of actors is broken by exposing their mutable state to the outside, you are back in normal Java concurrency land with all the drawbacks.
3. Actors are made to be containers for behavior and state, embracing this means to not routinely send behavior within messages (which may be tempting using Scala closures). One of the risks is to accidentally share mutable state between actors, and this violation of the actor model unfortunately breaks all the properties which make programming in actors such a nice experience.
4. Top-level actors are the innermost part of your Error Kernel, so create them sparingly and prefer truly hierarchical systems. This has benefits wrt. fault-handling (both considering the granularity of configuration and the performance) and it also reduces the number of blocking calls made, since the creation of top-level actors involves synchronous messaging.

2.1.4 What you should not concern yourself with

An actor system manages the resources it is configured to use in order to run the actors which it contains. There may be millions of actors within one such system, after all the mantra is to view them as abundant and they weigh in at an overhead of only roughly 300 bytes per instance. Naturally, the exact order in which messages are processed in large systems is not controllable by the application author, but this is also not intended. Take a step back and relax while Akka does the heavy lifting under the hood.

2.2 What is an Actor?

The previous section about *Actor Systems* explained how actors form hierarchies and are the smallest unit when building an application. This section looks at one such actor in isolation, explaining the concepts you encounter while implementing it. For more an in depth reference with all the details please refer to *Actors (Scala)* and *Actors (Java)*.

An actor is a container for State, Behavior, a Mailbox, Children and a Supervisor Strategy. All of this is encapsulated behind an Actor Reference. Finally, this happens When an Actor Terminates.

2.2.1 Actor Reference

As detailed below, an actor object needs to be shielded from the outside in order to benefit from the actor model. Therefore, actors are represented to the outside using actor references, which are objects that can be passed around freely and without restriction. This split into inner and outer object enables transparency for all the desired operations: restarting an actor without needing to update references elsewhere, placing the actual actor object on remote hosts, sending messages to actors in completely different applications. But the most important aspect is that it is not possible to look inside an actor and get hold of its state from the outside, unless the actor unwisely publishes this information itself.

2.2.2 State

Actor objects will typically contain some variables which reflect possible states the actor may be in. This can be an explicit state machine (e.g. using the *FSM* module), or it could be a counter, set of listeners, pending requests, etc. These data are what make an actor valuable, and they must be protected from corruption by other actors. The good news is that Akka actors conceptually each have their own light-weight thread, which is completely shielded from the rest of the system. This means that instead of having to synchronize access using locks you can just write your actor code without worrying about concurrency at all.

Behind the scenes Akka will run sets of actors on sets of real threads, where typically many actors share one thread, and subsequent invocations of one actor may end up being processed on different threads. Akka ensures that this implementation detail does not affect the single-threadedness of handling the actor's state.

Because the internal state is vital to an actor's operations, having inconsistent state is fatal. Thus, when the actor fails and is restarted by its supervisor, the state will be created from scratch, like upon first creating the actor. This is to enable the ability of self-healing of the system.

2.2.3 Behavior

Every time a message is processed, it is matched against the current behavior of the actor. Behavior means a function which defines the actions to be taken in reaction to the message at that point in time, say forward a request if the client is authorized, deny it otherwise. This behavior may change over time, e.g. because different clients obtain authorization over time, or because the actor may go into an "out-of-service" mode and later come back. These changes are achieved by either encoding them in state variables which are read from the behavior logic, or the function itself may be swapped out at runtime, see the `become` and `unbecome` operations. However, the initial behavior defined during construction of the actor object is special in the sense that a restart of the actor will reset its behavior to this initial one.

Note: The initial behavior of an Actor is extracted prior to constructor is run, so if you want to base your initial behavior on member state, you should use `become` in the constructor.

2.2.4 Mailbox

An actor's purpose is the processing of messages, and these messages were sent to the actor from other actors (or from outside the actor system). The piece which connects sender and receiver is the actor's mailbox: each actor has exactly one mailbox to which all senders enqueue their messages. Enqueuing happens in the time-order of send operations, which means that messages sent from different actors may not have a defined order at runtime due to the apparent randomness of distributing actors across threads. Sending multiple messages to the same target from the same actor, on the other hand, will enqueue them in the same order.

There are different mailbox implementations to choose from, the default being a FIFO: the order of the messages processed by the actor matches the order in which they were enqueued. This is usually a good default, but applications may need to prioritize some messages over others. In this case, a priority mailbox will enqueue not always at the end but at a position as given by the message priority, which might even be at the front. While using such a queue, the order of messages processed will naturally be defined by the queue's algorithm and in general not be FIFO.

An important feature in which Akka differs from some other actor model implementations is that the current behavior must always handle the next dequeued message, there is no scanning the mailbox for the next matching one. Failure to handle a message will typically be treated as a failure, unless this behavior is overridden.

2.2.5 Children

Each actor is potentially a supervisor: if it creates children for delegating sub-tasks, it will automatically supervise them. The list of children is maintained within the actor's context and the actor has access to it. Modifications to the list are done by creating (`context.actorOf(...)`) or stopping (`context.stop(child)`) children and these actions are reflected immediately. The actual creation and termination actions happen behind the scenes in an asynchronous way, so they do not "block" their supervisor.

2.2.6 Supervisor Strategy

The final piece of an actor is its strategy for handling faults of its children. Fault handling is then done transparently by Akka, applying one of the strategies described in *Supervision and Monitoring* for each incoming failure. As this strategy is fundamental to how an actor system is structured, it cannot be changed once an actor has been created.

Considering that there is only one such strategy for each actor, this means that if different strategies apply to the various children of an actor, the children should be grouped beneath intermediate supervisors with matching strategies, preferring once more the structuring of actor systems according to the splitting of tasks into sub-tasks.

2.2.7 When an Actor Terminates

Once an actor terminates, i.e. fails in a way which is not handled by a restart, stops itself or is stopped by its supervisor, it will free up its resources, draining all remaining messages from its mailbox into the system's "dead letter mailbox". The mailbox is then replaced within the actor reference with a system mailbox, redirecting all new messages "into the drain". This is done on a best effort basis, though, so do not rely on it in order to construct "guaranteed delivery".

The reason for not just silently dumping the messages was inspired by our tests: we register the TestEventListener on the event bus to which the dead letters are forwarded, and that will log a warning for every dead letter received—this has been very helpful for deciphering test failures more quickly. It is conceivable that this feature may also be of use for other purposes.

2.3 Supervision and Monitoring

This chapter outlines the concept behind supervision, the primitives offered and their semantics. For details on how that translates into real code, please refer to the corresponding chapters for Scala and Java APIs.

2.3.1 What Supervision Means

As described in *Actor Systems* supervision describes a dependency relationship between actors: the supervisor delegates tasks to subordinates and therefore must respond to their failures. When a subordinate detects a failure (i.e. throws an exception), it suspends itself and all its subordinates and sends a message to its supervisor, signaling failure. Depending on the nature of the work to be supervised and the nature of the failure, the supervisor has a choice of the following four options:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Terminate the subordinate permanently
4. Escalate the failure

It is important to always view an actor as part of a supervision hierarchy, which explains the existence of the fourth choice (as a supervisor also is subordinate to another supervisor higher up) and has implications on the first three: resuming an actor resumes all its subordinates, restarting an actor entails restarting all its subordinates (but see below for more details), similarly terminating an actor will also terminate all its subordinates. It should be noted that the default behavior of the `preRestart` hook of the `Actor` class is to terminate all its children before restarting, but this hook can be overridden; the recursive restart applies to all children left after this hook has been executed.

Each supervisor is configured with a function translating all possible failure causes (i.e. exceptions) into one of the four choices given above; notably, this function does not take the failed actor's identity as an input. It is quite easy to come up with examples of structures where this might not seem flexible enough, e.g. wishing for different strategies to be applied to different subordinates. At this point it is vital to understand that supervision is about forming a recursive fault handling structure. If you try to do too much at one level, it will become hard to reason about, hence the recommended way in this case is to add a level of supervision.

Akka implements a specific form called "parental supervision". Actors can only be created by other actors—where the top-level actor is provided by the library—and each created actor is supervised by its parent. This restriction makes the formation of actor supervision hierarchies explicit and encourages sound design decisions. It should be noted that this also guarantees that actors cannot be orphaned or attached to supervisors from the outside, which might otherwise catch them unawares. In addition, this yields a natural and clean shutdown procedure for (sub-trees of) actor applications.

2.3.2 What Restarting Means

When presented with an actor which failed while processing a certain message, causes for the failure fall into three categories:

- Systematic (i.e. programming) error for the specific message received
- (Transient) failure of some external resource used during processing the message
- Corrupt internal state of the actor

Unless the failure is specifically recognizable, the third cause cannot be ruled out, which leads to the conclusion that the internal state needs to be cleared out. If the supervisor decides that its other children or itself is not affected by the corruption—e.g. because of conscious application of the error kernel pattern—it is therefore best to restart the child. This is carried out by creating a new instance of the underlying `Actor` class and replacing the failed instance with the fresh one inside the child's `ActorRef`; the ability to do this is one of the reasons for encapsulating actors within special references. The new actor then resumes processing its mailbox, meaning that the restart is not visible outside of the actor itself with the notable exception that the message during which the failure occurred is not re-processed.

The precise sequence of events during a restart is the following:

- suspend the actor
- call the old instance's `supervisionStrategy.handleSupervisorFailing` method (defaults to suspending all children)
- call the old instance's `preRestart` hook (defaults to sending termination requests to all children and calling `postStop`)
- wait for all children stopped during `preRestart` to actually terminate
- call the old instance's `supervisionStrategy.handleSupervisorRestarted` method (defaults to sending restart request to all remaining children)
- create new actor instance by invoking the originally provided factory again
- invoke `postRestart` on the new instance
- resume the actor

2.3.3 What Lifecycle Monitoring Means

In contrast to the special relationship between parent and child described above, each actor may monitor any other actor. Since actors emerge from creation fully alive and restarts are not visible outside of the affected supervisors, the only state change available for monitoring is the transition from alive to dead. Monitoring is thus used to tie one actor to another so that it may react to the other actor's termination, in contrast to supervision which reacts to failure.

Lifecycle monitoring is implemented using a `Terminated` message to be received by the monitoring actor, where the default behavior is to throw a special `DeathPactException` if not otherwise handled. One important property is that the message will be delivered irrespective of the order in which the monitoring request and target's termination occur, i.e. you still get the message even if at the time of registration the target is already dead.

Monitoring is particularly useful if a supervisor cannot simply restart its children and has to terminate them, e.g. in case of errors during actor initialization. In that case it should monitor those children and re-create them or schedule itself to retry this at a later time.

Another common use case is that an actor needs to fail in the absence of an external resource, which may also be one of its own children. If a third party terminates a child by way of the `system.stop(child)` method or sending a `PoisonPill`, the supervisor might well be affected.

2.3.4 One-For-One Strategy vs. All-For-One Strategy

There are two classes of supervision strategies which come with Akka: `OneForOneStrategy` and `AllForOneStrategy`. Both are configured with a mapping from exception type to supervision directive (see *above*) and limits on how often a child is allowed to fail before terminating it. The difference between them is that the former applies the obtained directive only to the failed child, whereas the latter applies it to all siblings as well. Normally, you should use the `OneForOneStrategy`, which also is the default if none is specified explicitly.

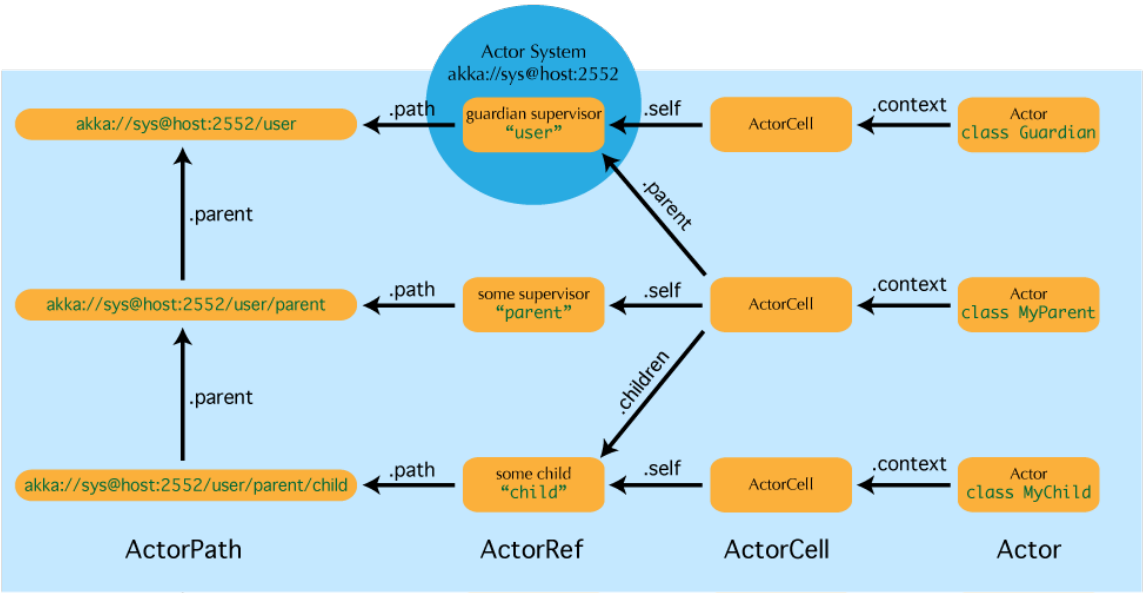
The `AllForOneStrategy` is applicable in cases where the ensemble of children has so tight dependencies among them, that a failure of one child affects the function of the others, i.e. they are intricably linked. Since a restart does not clear out the mailbox, it often is best to terminate the children upon failure and re-create them explicitly from the supervisor (by watching the children's lifecycle); otherwise you have to make sure that it is no problem for any of the actors to receive a message which was queued before the restart but processed afterwards.

Normally stopping a child (i.e. not in response to a failure) will not automatically terminate the other children in an all-for-one strategy, that can easily be done by watching their lifecycle: if the `Terminated` message is not handled by the supervisor, it will throw a `DeathPathException` which (depending on its supervisor) will restart it, and the default `preRestart` action will terminate all children. Of course this can be handled explicitly as well.

Please note that creating one-off actors from an all-for-one supervisor entails that failures escalated by the temporary actor will affect all the permanent ones. If this is not desired, install an intermediate supervisor; this can very easily be done by declaring a router of size 1 for the worker, see *Routing (Scala)* or *Routing (Java)*.

2.4 Actor References, Paths and Addresses

This chapter describes how actors are identified and located within a possibly distributed actor system. It ties into the central idea that *Actor Systems* form intrinsic supervision hierarchies as well as that communication between actors is transparent with respect to their placement across multiple network nodes.



The above image displays the relationship between the most important entities within an actor system, please read on for the details.

2.4.1 What is an Actor Reference?

An actor reference is a subtype of `ActorRef`, whose foremost purpose is to support sending messages to the actor it represents. Each actor has access to its canonical (local) reference through the `self` field; this reference is also included as sender reference by default for all messages sent to other actors. Conversely, during message processing the actor has access to a reference representing the sender of the current message through the `sender` field.

There are several different types of actor references that are supported depending on the configuration of the actor system:

- Purely local actor references are used by actor systems which are not configured to support networking functions. These actor references cannot ever be sent across a network connection while retaining their functionality.
- Local actor references when remoting is enabled are used by actor systems which support networking functions for those references which represent actors within the same JVM. In order to be recognizable also when sent to other network nodes, these references include protocol and remote addressing information.
- There is a subtype of local actor references which is used for routers (i.e. actors mixing in the `Router` trait). Its logical structure is the same as for the aforementioned local references, but sending a message to them dispatches to one of their children directly instead.
- Remote actor references represent actors which are reachable using remote communication, i.e. sending messages to them will serialize the messages transparently and send them to the other JVM.
- There are several special types of actor references which behave like local actor references for all practical purposes:
  - `PromiseActorRef` is the special representation of a `Promise` for the purpose of being completed by the response from an actor; it is created by the `ActorRef.ask` invocation.
  - `DeadLetterActorRef` is the default implementation of the dead letters service, where all messages are re-routed whose routees are shut down or non-existent.
  - `EmptyLocalActorRef` is what is returned when looking up a non-existing local actor path: it is equivalent to a `DeadLetterActorRef`, but it retains its path so that it can be sent over the network and compared to other existing actor refs for that path, some of which might have been obtained before the actor stopped existing.

- And then there are some one-off internal implementations which you should never really see:
  - There is an actor reference which does not represent an actor but acts only as a pseudo-supervisor for the root guardian, we call it "the one who walks the bubbles of space-time".
  - The first logging service started before actually firing up actor creation facilities is a fake actor reference which accepts log events and prints them directly to standard output; it is `Logging.StandardOutLogger`.
- **(Future Extension)** Cluster actor references represent clustered actor services which may be replicated, migrated or load-balanced across multiple cluster nodes. As such they are virtual names which the cluster service translates into local or remote actor references as appropriate.

2.4.2 What is an Actor Path?

Since actors are created in a strictly hierarchical fashion, there exists a unique sequence of actor names given by recursively following the supervision links between child and parent down towards the root of the actor system. This sequence can be seen as enclosing folders in a file system, hence we adopted the name "path" to refer to it. As in some real file-systems there also are "symbolic links", i.e. one actor may be reachable using more than one path, where all but one involve some translation which decouples part of the path from the actor's actual supervision ancestor line; these specialities are described in the sub-sections to follow.

An actor path consists of an anchor, which identifies the actor system, followed by the concatenation of the path elements, from root guardian to the designated actor; the path elements are the names of the traversed actors and are separated by slashes.

Actor Path Anchors

Each actor path has an address component, describing the protocol and location by which the corresponding actor is reachable, followed by the names of the actors in the hierarchy from the root up. Examples are:

```
"akka://my-system/user/service-a/worker1"                // purely local
"akka://my-system@serv.example.com:5678/user/service-b" // local or remote
"cluster://my-cluster/service-c"                         // clustered (Future Extension)
```

Here, `akka` is the default remote protocol for the 2.0 release, and others are pluggable. The interpretation of the host & port part (i.e. `serv.example.com:5678` in the example) depends on the transport mechanism used, but it must abide by the URI structural rules.

Logical Actor Paths

The unique path obtained by following the parental supervision links towards the root guardian is called the logical actor path. This path matches exactly the creation ancestry of an actor, so it is completely deterministic as soon as the actor system's remoting configuration (and with it the address component of the path) is set.

Physical Actor Paths

While the logical actor path describes the functional location within one actor system, configuration-based remote deployment means that an actor may be created on a different network host as its parent, i.e. within a different actor system. In this case, following the actor path from the root guardian up entails traversing the network, which is a costly operation. Therefore, each actor also has a physical path, starting at the root guardian of the actor system where the actual actor object resides. Using this path as sender reference when querying other actors will let them reply directly to this actor, minimizing delays incurred by routing.

One important aspect is that a physical actor path never spans multiple actor systems or JVMs. This means that the logical path (supervision hierarchy) and the physical path (actor deployment) of an actor may diverge if one of its ancestors is remotely supervised.

Virtual Actor Paths (Future Extension)

In order to be able to replicate and migrate actors across a cluster of Akka nodes, another level of indirection has to be introduced. The cluster component therefore provides a translation from virtual paths to physical paths which may change in reaction to node failures, cluster rebalancing, etc.

This area is still under active development, expect updates in this section for the 2.1 release.

2.4.3 How are Actor References obtained?

There are two general categories to how actor references may be obtained: by creating actors or by looking them up, where the latter functionality comes in the two flavours of creating actor references from concrete actor paths and querying the logical actor hierarchy.

While local and remote actor references and their paths work in the same way concerning the facilities mentioned below, the exact semantics of clustered actor references and their paths—while certainly as similar as possible—may differ in certain aspects, owing to the virtual nature of those paths. Expect updates for the 2.1 release.

Creating Actors

An actor system is typically started by creating actors above the guardian actor using the `ActorSystem.actorOf` method and then using `ActorContext.actorOf` from within the created actors to spawn the actor tree. These methods return a reference to the newly created actor. Each actor has direct access to references for its parent, itself and its children. These references may be sent within messages to other actors, enabling those to reply directly.

Looking up Actors by Concrete Path

In addition, actor references may be looked up using the `ActorSystem.actorFor` method, which returns an (unverified) local, remote or clustered actor reference. Sending messages to such a reference or attempting to observe its liveness will traverse the actor hierarchy of the actor system from top to bottom by passing messages from parent to child until either the target is reached or failure is certain, i.e. a name in the path does not exist (in practice this process will be optimized using caches, but it still has added cost compared to using the physical actor path, which can for example to obtained from the sender reference included in replies from that actor). The messages passed are handled automatically by Akka, so this process is not visible to client code.

Absolute vs. Relative Paths

In addition to `ActorSystem.actorFor` there is also `ActorContext.actorFor`, which is available inside any actor as `context.actorFor`. This yields an actor reference much like its twin on `ActorSystem`, but instead of looking up the path starting from the root of the actor tree it starts out on the current actor. Path elements consisting of two dots (`".."`) may be used to access the parent actor. You can for example send a message to a specific sibling:

```
context.actorFor("../brother") ! msg
```

Absolute paths may of course also be looked up on *context* in the usual way, i.e.

```
context.actorFor("/user/serviceA") ! msg
```

will work as expected.

Querying the Logical Actor Hierarchy

Since the actor system forms a file-system like hierarchy, matching on paths is possible in the same was as supported by Unix shells: you may replace (parts of) path element names with wildcards (*«*»* and *«?»*) to formulate a selection which may match zero or more actual actors. Because the result is not a single actor reference, it has a different type `ActorSelection` and does not support the full set of operations an `ActorRef` does. Selections may be formulated using the `ActorSystem.actorSelection` and `ActorContext.actorSelection` methods and do support sending messages:

```
context.actorSelection("../*") ! msg
```

will send *msg* to all siblings including the current actor. As for references obtained using *actorFor*, a traversal of the supervision hierarchy is done in order to perform the message send. As the exact set of actors which match a selection may change even while a message is making its way to the recipients, it is not possible to watch a selection for liveliness changes. In order to do that, resolve the uncertainty by sending a request and gathering all answers, extracting the sender references, and then watch all discovered concrete actors. This scheme of resolving a selection may be improved upon in a future release.

Summary: `actorOf` vs. `actorFor`

Note: What the above sections described in some detail can be summarized and memorized easily as follows:

- `actorOf` only ever creates a new actor, and it creates it as a direct child of the context on which this method is invoked (which may be any actor or actor system).
- `actorFor` only ever looks up an existing actor, i.e. does not create one.

2.4.4 Reusing Actor Paths

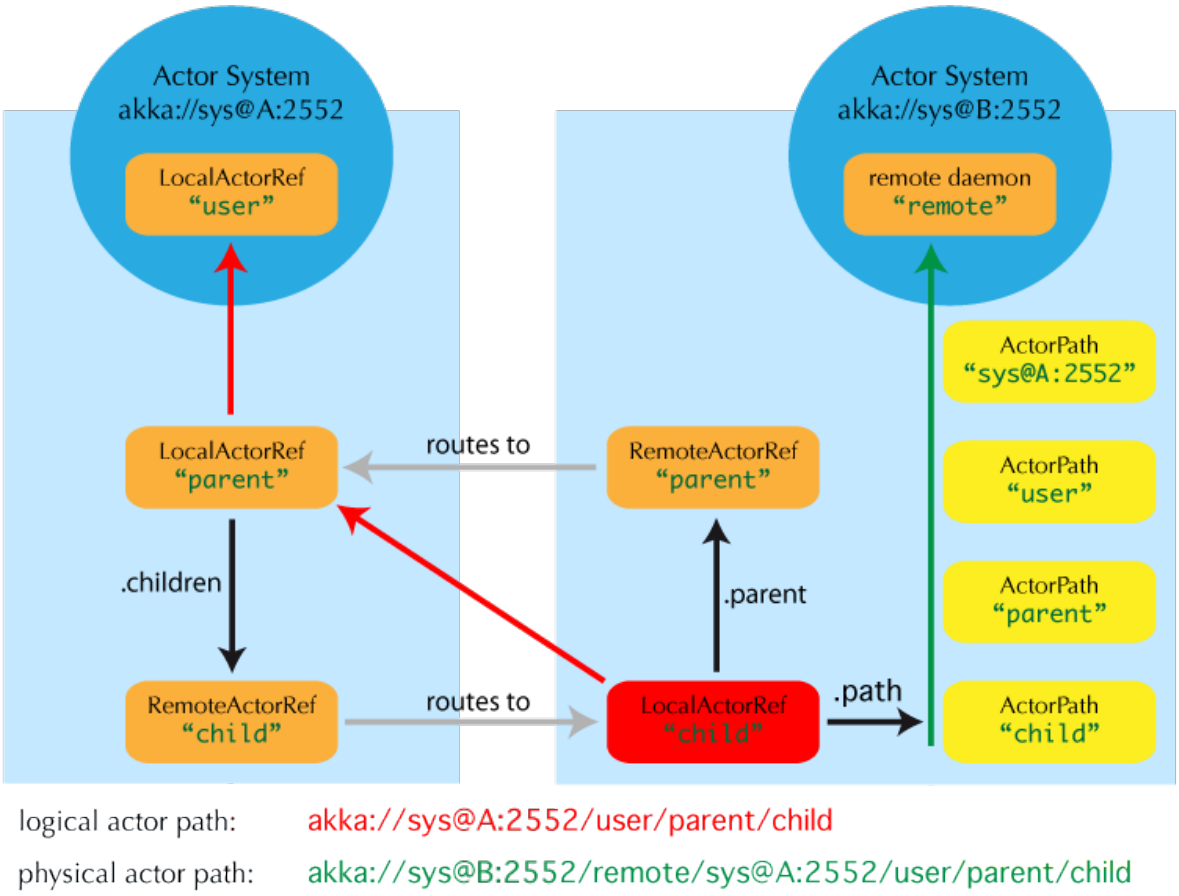
When an actor is terminated, its path will point to the dead letter mailbox, DeathWatch will publish its final transition and in general it is not expected to come back to life again (since the actor life cycle does not allow this). While it is possible to create an actor at a later time with an identical path—simply due to it being impossible to enforce the opposite without keeping the set of all actors ever created available—this is not good practice: remote actor references which "died" suddenly start to work again, but without any guarantee of ordering between this transition and any other event, hence the new inhabitant of the path may receive messages which were destined for the previous tenant.

It may be the right thing to do in very specific circumstances, but make sure to confine the handling of this precisely to the actor's supervisor, because that is the only actor which can reliably detect proper deregistration of the name, before which creation of the new child will fail.

It may also be required during testing, when the test subject depends on being instantiated at a specific path. In that case it is best to mock its supervisor so that it will forward the Terminated message to the appropriate point in the test procedure, enabling the latter to await proper deregistration of the name.

2.4.5 The Interplay with Remote Deployment

When an actor creates a child, the actor system's deployer will decide whether the new actor resides in the same JVM or on another node. In the second case, creation of the actor will be triggered via a network connection to happen in a different JVM and consequently within a different actor system. The remote system will place the new actor below a special path reserved for this purpose and the supervisor of the new actor will be a remote actor reference (representing that actor which triggered its creation). In this case, `context.parent` (the supervisor reference) and `context.path.parent` (the parent node in the actor's path) do not represent the same actor. However, looking up the child's name within the supervisor will find it on the remote node, preserving logical structure e.g. when sending to an unresolved actor reference.



2.4.6 The Interplay with Clustering (Future Extension)

This section is subject to change!

When creating a scaled-out actor subtree, a cluster name is created for a routed actor reference, where sending to this reference will send to one (or more) of the actual actors created in the cluster. In order for those actors to be able to query other actors while processing their messages, their sender reference must be unique for each of the replicas, which means that physical paths will be used as `self` references for these instances. In the case of replication for achieving fault-tolerance the opposite is required: the `self` reference will be a virtual (cluster) path so that in case of migration or fail-over communication is resumed with the fresh instance.

2.4.7 What is the Address part used for?

When sending an actor reference across the network, it is represented by its path. Hence, the path must fully encode all information necessary to send messages to the underlying actor. This is achieved by encoding protocol, host and port in the address part of the path string. When an actor system receives an actor path from a remote node, it checks whether that path's address matches the address of this actor system, in which case it will be resolved to the actor's local reference. Otherwise, it will be represented by a remote actor reference.

2.4.8 Special Paths used by Akka

At the root of the path hierarchy resides the root guardian above which all other actors are found. The next level consists of the following:

- `"/user"` is the guardian actor for all user-created top-level actors; actors created using `ActorSystem.actorOf` are found at the next level.

- `"/system"` is the guardian actor for all system-created top-level actors, e.g. logging listeners or actors automatically deployed by configuration at the start of the actor system.
- `"/deadLetters"` is the dead letter actor, which is where all messages sent to stopped or non-existing actors are re-routed.
- `"/temp"` is the guardian for all short-lived system-created actors, e.g. those which are used in the implementation of `ActorRef.ask`.
- `"/remote"` is an artificial path below which all actors reside whose supervisors are remote actor references

2.5 Location Transparency

The previous section describes how actor paths are used to enable location transparency. This special feature deserves some extra explanation, because the related term "transparent remoting" was used quite differently in the context of programming languages, platforms and technologies.

2.5.1 Distributed by Default

Everything in Akka is designed to work in a distributed setting: all interactions of actors use purely message passing and everything is asynchronous. This effort has been undertaken to ensure that all functions are available equally when running within a single JVM or on a cluster of hundreds of machines. The key for enabling this is to go from remote to local by way of optimization instead of trying to go from local to remote by way of generalization. See this classic paper for a detailed discussion on why the second approach is bound to fail.

2.5.2 Ways in which Transparency is Broken

What is true of Akka need not be true of the application which uses it, since designing for distributed execution poses some restrictions on what is possible. The most obvious one is that all messages sent over the wire must be serializable. While being a little less obvious this includes closures which are used as actor factories (i.e. within `Props`) if the actor is to be created on a remote node.

Another consequence is that everything needs to be aware of all interactions being fully asynchronous, which in a computer network might mean that it may take several minutes for a message to reach its recipient (depending on configuration). It also means that the probability for a message to be lost is much higher than within one JVM, where it is close to zero (still: no hard guarantee!).

2.5.3 How is Remoting Used?

We took the idea of transparency to the limit in that there is nearly no API for the remoting layer of Akka: it is purely driven by configuration. Just write your application according to the principles outlined in the previous sections, then specify remote deployment of actor sub-trees in the configuration file. This way, your application can be scaled out without having to touch the code. The only piece of the API which allows programmatic influence on remote deployment is that `Props` contain a field which may be set to a specific `Deploy` instance; this has the same effect as putting an equivalent deployment into the configuration file (if both are given, configuration file wins).

2.5.4 Marking Points for Scaling Up with Routers

In addition to being able to run different parts of an actor system on different nodes of a cluster, it is also possible to scale up onto more cores by multiplying actor sub-trees which support parallelization (think for example a search engine processing different queries in parallel). The clones can then be routed to in different fashions, e.g. round-robin. The only thing necessary to achieve this is that the developer needs to declare a certain actor as "withRouter", then—in its stead—a router actor will be created which will spawn up a configurable number of children of the desired type and route to them in the configured fashion. Once such a router has been declared, its

configuration can be freely overridden from the configuration file, including mixing it with the remote deployment of (some of) the children. Read more about this in *Routing (Scala)* and *Routing (Java)*.

2.6 Akka and the Java Memory Model

A major benefit of using the Typesafe Stack, including Scala and Akka, is that it simplifies the process of writing concurrent software. This article discusses how the Typesafe Stack, and Akka in particular, approaches shared memory in concurrent applications.

2.6.1 The Java Memory Model

Prior to Java 5, the Java Memory Model (JMM) was ill defined. It was possible to get all kinds of strange results when shared memory was accessed by multiple threads, such as:

- a thread not seeing values written by other threads: a visibility problem
- a thread observing 'impossible' behavior of other threads, caused by instructions not being executed in the order expected: an instruction reordering problem.

With the implementation of JSR 133 in Java 5, a lot of these issues have been resolved. The JMM is a set of rules based on the "happens-before" relation, which constrain when one memory access must happen before another, and conversely, when they are allowed to happen out of order. Two examples of these rules are:

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

Although the JMM can seem complicated, the specification tries to find a balance between ease of use and the ability to write performant and scalable concurrent data structures.

2.6.2 Actors and the Java Memory Model

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

- if a message is sent 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 a "happens before" rule, it would be possible for the receiver to see partially initialized data structures and possibly even values out of thin air (longs/doubles).
- if an actor makes changes to its internal state while processing a message, and accesses that state while processing another message moments later. It is important to realize that with the actor model you don't get any guarantee that the same thread will be executing the same actor for different messages.

To prevent visibility and reordering problems on actors, Akka guarantees the following two "happens before" rules:

- **The actor send rule:** the send of the message to an actor happens before the receive of that message by the same actor.
- **The actor subsequent processing rule:** 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.

2.6.3 Futures and the Java Memory Model

The completion of a Future "happens before" the invocation of any callbacks registered to it are executed.

We recommend not to close over non-final fields (final in Java and val in Scala), and if you *do* choose to close over non-final fields, they must be marked *volatile* in order for the current value of the field to be visible to the callback.

If you close over a reference, you must also ensure that the instance that is referred to is thread safe. We highly recommend staying away from objects that use locking, since it can introduce performance problems and in the worst case, deadlocks. Such are the perils of synchronized.

2.6.4 STM and the Java Memory Model

Akka's Software Transactional Memory (STM) also provides a "happens before" rule:

- **The transactional reference rule:** a successful write during commit, on an transactional reference, happens before every subsequent read of the same transactional reference.

This rule looks a lot like the 'volatile variable' rule from the JMM. Currently the Akka STM only supports deferred writes, so the actual writing to shared memory is deferred until the transaction commits. Writes during the transaction are placed in a local buffer (the writeset of the transaction) and are not visible to other transactions. That is why dirty reads are not possible.

How these rules are realized in Akka is an implementation detail and can change over time, and the exact details could even depend on the used configuration. But they will build on the other JMM rules like the monitor lock rule or the volatile variable rule. 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 business logic, and the Akka framework makes sure that those rules are guaranteed on your behalf.

2.6.5 Actors and shared mutable state

Since Akka runs on the JVM there are still some rules to be followed.

- Closing over internal Actor state and exposing it to other threads

```
class MyActor extends Actor {
 var state = ...
 def receive = {
    case _ =>
      //Wrongs

    // Very bad, shared mutable state,
    // will break your application in weird ways
      Future { state = NewState }
      anotherActor ? message onSuccess { r => state = r }

    // Very bad, "sender" changes for every message,
    // shared mutable state bug
      Future { expensiveCalculation(sender) }

      //Rights

    // Completely safe, "self" is OK to close over
    // and it's an ActorRef, which is thread-safe
      Future { expensiveCalculation() } onComplete { f => self ! f.value.get }

    // Completely safe, we close over a fixed value
    // and it's an ActorRef, which is thread-safe
      val currentSender = sender
      Future { expensiveCalculation(currentSender) }
```

```
  }
}
```

- Messages **should** be immutable, this is to avoid the shared mutable state trap.

2.7 Message send semantics

2.7.1 Guaranteed Delivery?

Akka does *not* support guaranteed delivery.

First it is close to impossible to actually give guarantees like that, second it is extremely costly trying to do so. The network is inherently unreliable and there is no such thing as 100% guarantee delivery, so it can never be guaranteed.

The question is what to guarantee. That:

1. The message is sent out on the network?
2. The message is received by the other host?
3. The message is put on the target actor's mailbox?
4. The message is applied to the target actor?
5. The message is starting to be executed by the target actor?
6. The message is finished executing by the target actor?

Each one of this have different challenges and costs.

Akka embraces distributed computing and the network and makes it explicit through message passing, therefore it does not try to lie and emulate a leaky abstraction. This is a model that have been used with great success in Erlang and requires the user to model his application around. You can read more about this approach in the Erlang documentation (section 10.9 and 10.10), Akka follows it closely.

Bottom line: you as a developer know what guarantees you need in your application and can solve it fastest and most reliable by explicit `ACK` and `RETRY` (if you really need it, most often you don't). Using Akka's Durable Mailboxes could help with this.

2.7.2 Delivery semantics

At-most-once

Actual transports may provide stronger semantics, but at-most-once is the semantics you should expect. The alternatives would be once-and-only-once, which is extremely costly, or at-least-once which essentially requires idempotency of message processing, which is a user-level concern.

Ordering is preserved on a per-sender basis

Actor `A1` sends messages `M1`, `M2`, `M3` to `A2` Actor `A3` sends messages `M4`, `M5`, `M6` to `A2`

This means that:

1. If `M1` is delivered it must be delivered before `M2` and `M3`
2. If `M2` is delivered it must be delivered before `M3`
3. If `M4` is delivered it must be delivered before `M5` and `M6`
4. If `M5` is delivered it must be delivered before `M6`
5. `A2` can see messages from `A1` interleaved with messages from `A3`

6. Since there is no guaranteed delivery, none, some or all of the messages may arrive to `A2`

2.8 Configuration

Akka uses the Typesafe Config Library, which might also be a good choice for the configuration of your own application or library built with or without Akka. This library is implemented in Java with no external dependencies; you should have a look at its documentation (in particular about ConfigFactory), which is only summarized in the following.

Warning: If you use Akka from the Scala REPL from the 2.9.x series, and you do not provide your own ClassLoader to the ActorSystem, start the REPL with "-Yrepl-sync" to work around a deficiency in the REPLs provided Context ClassLoader.

2.8.1 Where configuration is read from

All configuration for Akka is held within instances of `ActorSystem`, or put differently, as viewed from the outside, `ActorSystem` is the only consumer of configuration information. While constructing an actor system, you can either pass in a `Config` object or not, where the second case is equivalent to passing `ConfigFactory.load()` (with the right class loader). This means roughly that the default is to parse all `application.conf`, `application.json` and `application.properties` found at the root of the class path—please refer to the aforementioned documentation for details. The actor system then merges in all `reference.conf` resources found at the root of the class path to form the fallback configuration, i.e. it internally uses

```
appConfig.withFallback(ConfigFactory.defaultReference(classLoader))
```

The philosophy is that code never contains default values, but instead relies upon their presence in the `reference.conf` supplied with the library in question.

Highest precedence is given to overrides given as system properties, see the HOCON specification (near the bottom). Also noteworthy is that the application configuration—which defaults to `application`—may be overridden using the `config.resource` property (there are more, please refer to the Config docs).

Note: If you are writing an Akka application, keep you configuration in `application.conf` at the root of the class path. If you are writing an Akka-based library, keep its configuration in `reference.conf` at the root of the JAR file.

2.8.2 When using JarJar, OneJar, Assembly or any jar-bundler

Warning: Akka's configuration approach relies heavily on the notion of every module/jar having its own reference.conf file, all of these will be discovered by the configuration and loaded. Unfortunately this also means that if you put merge multiple jars into the same jar, you need to merge all the reference.confs as well. Otherwise all defaults will be lost and Akka will not function.

2.8.3 How to structure your configuration

Given that `ConfigFactory.load()` merges all resources with matching name from the whole class path, it is easiest to utilize that functionality and differenciate actor systems within the hierarchy of the configuration:

```
myapp1 {
  akka.loglevel = WARNING
  my.own.setting = 43
```

```
}
myapp2 {
  akka.loglevel = ERROR
  app2.setting = "appname"
}
my.own.setting = 42
my.other.setting = "hello"
```

```
val config = ConfigFactory.load()
val app1 = ActorSystem("MyApp1", config.getConfig("myapp1").withFallback(config))
val app2 = ActorSystem("MyApp2", config.getConfig("myapp2").withOnlyPath("akka").withFallback(con
```

These two samples demonstrate different variations of the "lift-a-subtree" trick: in the first case, the configuration accessible from within the actor system is this

```
akka.loglevel = WARNING
my.own.setting = 43
my.other.setting = "hello"
// plus myapp1 and myapp2 subtrees
```

while in the second one, only the "akka" subtree is lifted, with the following result:

```
akka.loglevel = ERROR
my.own.setting = 42
my.other.setting = "hello"
// plus myapp1 and myapp2 subtrees
```

Note: The configuration library is really powerful, explaining all features exceeds the scope affordable here. In particular not covered are how to include other configuration files within other files (see a small example at Including files) and copying parts of the configuration tree by way of path substitutions.

You may also specify and parse the configuration programmatically in other ways when instantiating the `ActorSystem`.

```
import akka.actor.ActorSystem
import com.typesafe.config.ConfigFactory
    val customConf = ConfigFactory.parseString("""
      akka.actor.deployment {
        /my-service {
          router = round-robin
          nr-of-instances = 3
        }
      }
      """)
    // ConfigFactory.load sandwiches customConfig between default reference
    // config and default overrides, and then resolves it.
    val system = ActorSystem("MySystem", ConfigFactory.load(customConf))
```

2.8.4 Listing of the Reference Configuration

Each Akka module has a reference configuration file with the default values.

akka-actor

```
####################################
# Akka Actor Reference Config File #
####################################

# This is the reference config file that contains all the default settings.
```

```
# Make your edits/overrides in your application.conf.

akka {
  # Akka version, checked against the runtime version of Akka.
  version = "2.0.4"

  # Home directory of Akka, modules in the deploy directory will be loaded
  home = ""

  # Event handlers to register at boot time (Logging$DefaultLogger logs to STDOUT)
  event-handlers = ["akka.event.Logging$DefaultLogger"]

  # Event handlers are created and registered synchronously during ActorSystem
  # start-up, and since they are actors, this timeout is used to bound the
  # waiting time
  event-handler-startup-timeout = 5s

  # Log level used by the configured loggers (see "event-handlers") as soon
  # as they have been started; before that, see "stdout-loglevel"
  # Options: ERROR, WARNING, INFO, DEBUG
  loglevel = "INFO"

  # Log level for the very basic logger activated during AkkaApplication startup
  # Options: ERROR, WARNING, INFO, DEBUG
  stdout-loglevel = "WARNING"

  # Log the complete configuration at INFO level when the actor system is started.
  # This is useful when you are uncertain of what configuration is used.
  log-config-on-start = off

  # List FQCN of extensions which shall be loaded at actor system startup.
  # Should be on the format: 'extensions = ["foo", "bar"]' etc.
  # See the Akka Documentation for more info about Extensions
  extensions = []

  # Toggles whether the threads created by this ActorSystem should be daemons or not
  daemonic = off

  # JVM shutdown, System.exit(-1), in case of a fatal error, such as OutOfMemoryError
  jvm-exit-on-fatal-error = on

  actor {

    provider = "akka.actor.LocalActorRefProvider"

    # Timeout for ActorSystem.actorOf
    creation-timeout = 20s

    # frequency with which stopping actors are prodded in case they had to be
    # removed from their parents
    reaper-interval = 5s

    # Serializes and deserializes (non-primitive) messages to ensure immutability,
    # this is only intended for testing.
    serialize-messages = off

    # Serializes and deserializes creators (in Props) to ensure that they can be sent over the net
    # this is only intended for testing.
    serialize-creators = off

    typed {
      # Default timeout for typed actor methods with non-void return type
      timeout = 5s
```

```
  }

  deployment {

    # deployment id pattern - on the format: /parent/child etc.
    default {

      # routing (load-balance) scheme to use
      #     available: "from-code", "round-robin", "random", "smallest-mailbox", "scatter-gathe
      #     or:        Fully qualified class name of the router class.
      #                The router class must extend akka.routing.CustomRouterConfig and and have
      #                with com.typesafe.config.Config parameter.
      #     default is "from-code";
      # Whether or not an actor is transformed to a Router is decided in code only (Props.withR
      # The type of router can be overridden in the configuration; specifying "from-code" means
      # that the values specified in the code shall be used.
      # In case of routing, the actors to be routed to can be specified
      # in several ways:
      # - nr-of-instances: will create that many children
      # - routees.paths: will look the paths up using actorFor and route to
      #   them, i.e. will not create children
      # - resizer: dynamically resizable number of routees as specified in resizer below
      router = "from-code"

      # number of children to create in case of a non-direct router; this setting
      # is ignored if routees.paths is given
      nr-of-instances = 1

      # within is the timeout used for routers containing future calls
      within = 5 seconds

      routees {
        # Alternatively to giving nr-of-instances you can specify the full
        # paths of those actors which should be routed to. This setting takes
        # precedence over nr-of-instances
        paths = []
      }

      # Routers with dynamically resizable number of routees; this feature is enabled
      # by including (parts of) this section in the deployment
      resizer {

        # The fewest number of routees the router should ever have.
        lower-bound = 1

        # The most number of routees the router should ever have.
        # Must be greater than or equal to lower-bound.
        upper-bound = 10

        # Threshold to evaluate if routee is considered to be busy (under pressure).
        # Implementation depends on this value (default is 1).
        # 0:   number of routees currently processing a message.
        # 1:   number of routees currently processing a message has
        #      some messages in mailbox.
        # > 1: number of routees with at least the configured pressure-threshold
        #      messages in their mailbox. Note that estimating mailbox size of
        #      default UnboundedMailbox is O(N) operation.
        pressure-threshold = 1

        # Percentage to increase capacity whenever all routees are busy.
        # For example, 0.2 would increase 20% (rounded up), i.e. if current
        # capacity is 6 it will request an increase of 2 more routees.
        rampup-rate = 0.2
```

```
        # Minimum fraction of busy routees before backing off.
        # For example, if this is 0.3, then we'll remove some routees only when
        # less than 30% of routees are busy, i.e. if current capacity is 10 and
        # 3 are busy then the capacity is unchanged, but if 2 or less are busy
        # the capacity is decreased.
        # Use 0.0 or negative to avoid removal of routees.
        backoff-threshold = 0.3

        # Fraction of routees to be removed when the resizer reaches the
        # backoffThreshold.
        # For example, 0.1 would decrease 10% (rounded up), i.e. if current
        # capacity is 9 it will request an decrease of 1 routee.
        backoff-rate = 0.1

        # When the resizer reduce the capacity the abandoned routee actors are stopped
        # with PoisonPill after this delay. The reason for the delay is to give concurrent
        # messages a chance to be placed in mailbox before sending PoisonPill.
        # Use 0s to skip delay.
        stop-delay = 1s

        # Number of messages between resize operation.
        # Use 1 to resize before each message.
        messages-per-resize = 10
      }
    }
  }

  default-dispatcher {
    # Must be one of the following
    # Dispatcher, (BalancingDispatcher, only valid when all actors using it are of
    # the same type), PinnedDispatcher, or a FQCN to a class inheriting
    # MessageDispatcherConfigurator with a constructor with
    # com.typesafe.config.Config parameter and akka.dispatch.DispatcherPrerequisites
    # parameters.
    # PinnedDispatcher must be used toghether with executor=thread-pool-executor.
    type = "Dispatcher"

    # Which kind of ExecutorService to use for this dispatcher
    # Valid options:
    #               "fork-join-executor" requires a "fork-join-executor" section
    #               "thread-pool-executor" requires a "thread-pool-executor" section
    #               or
    #               A FQCN of a class extending ExecutorServiceConfigurator
    executor = "fork-join-executor"

    # This will be used if you have set "executor = "fork-join-executor""
    fork-join-executor {
      # Min number of threads to cap factor-based parallelism number to
      parallelism-min = 8

      # Parallelism (threads) ... ceil(available processors * factor)
      parallelism-factor = 3.0

      # Max number of threads to cap factor-based parallelism number to
      parallelism-max = 64
    }

    # This will be used if you have set "executor = "thread-pool-executor""
    thread-pool-executor {
      # Keep alive time for threads
      keep-alive-time = 60s
```

```
    # Min number of threads to cap factor-based core number to
    core-pool-size-min = 8

    # No of core threads ... ceil(available processors * factor)
    core-pool-size-factor = 3.0

    # Max number of threads to cap factor-based number to
    core-pool-size-max = 64

    # Hint: max-pool-size is only used for bounded task queues
    # minimum number of threads to cap factor-based max number to
    max-pool-size-min = 8

    # Max no of threads ... ceil(available processors * factor)
    max-pool-size-factor  = 3.0

    # Max number of threads to cap factor-based max number to
    max-pool-size-max = 64

    # Specifies the bounded capacity of the task queue (< 1 == unbounded)
    task-queue-size = -1

    # Specifies which type of task queue will be used, can be "array" or
    # "linked" (default)
    task-queue-type = "linked"

    # Allow core threads to time out
    allow-core-timeout = on
  }

  # How long time the dispatcher will wait for new actors until it shuts down
  shutdown-timeout = 1s

  # Throughput defines the number of messages that are processed in a batch
  # before the thread is returned to the pool. Set to 1 for as fair as possible.
  throughput = 5

  # Throughput deadline for Dispatcher, set to 0 or negative for no deadline
  throughput-deadline-time = 0ms

  # If negative (or zero) then an unbounded mailbox is used (default)
  # If positive then a bounded mailbox is used and the capacity is set using the
  # property
  # NOTE: setting a mailbox to 'blocking' can be a bit dangerous, could lead to
  # deadlock, use with care
  # The following mailbox-push-timeout-time is only used for type=Dispatcher and
  # only if mailbox-capacity > 0
  mailbox-capacity = -1

  # Specifies the timeout to add a new message to a mailbox that is full -
  # negative number means infinite timeout. It is only used for type=Dispatcher
  # and only if mailbox-capacity > 0
  mailbox-push-timeout-time = 10s

  # FQCN of the MailboxType, if not specified the default bounded or unbounded
  # mailbox is used. The Class of the FQCN must have a constructor with
  # (akka.actor.ActorSystem.Settings, com.typesafe.config.Config) parameters.
  mailbox-type = ""

  # For BalancingDispatcher: If the balancing dispatcher should attempt to
  # schedule idle actors using the same dispatcher when a message comes in,
  # and the dispatchers ExecutorService is not fully busy already.
  attempt-teamwork = on
```

```
      # For Actor with Stash: The default capacity of the stash.
      # If negative (or zero) then an unbounded stash is used (default)
      # If positive then a bounded stash is used and the capacity is set using the
      # property
      stash-capacity = -1
    }

    debug {
      # enable function of Actor.loggable(), which is to log any received message at
      # DEBUG level, see the "Testing Actor Systems" section of the Akka Documentation
      # at http://akka.io/docs
      receive = off

      # enable DEBUG logging of all AutoReceiveMessages (Kill, PoisonPill and the like)
      autoreceive = off

      # enable DEBUG logging of actor lifecycle changes
      lifecycle = off

      # enable DEBUG logging of all LoggingFSMs for events, transitions and timers
      fsm = off

      # enable DEBUG logging of subscription changes on the eventStream
      event-stream = off
    }

    # Entries for pluggable serializers and their bindings.
    serializers {
      java = "akka.serialization.JavaSerializer"
    }

    # Class to Serializer binding. You only need to specify the name of an interface
    # or abstract base class of the messages. In case of ambiguity it is using the
    # most specific configured class, or giving a warning and choosing the "first" one.
    #
    # To disable one of the default serializers, assign its class to "none", like
    # "java.io.Serializable" = none
    serialization-bindings {
      "java.io.Serializable" = java
    }
  }

  # Used to set the behavior of the scheduler.
  # Changing the default values may change the system behavior drastically so make sure
  # you know what you're doing! See the Scheduler section of the Akka documentation for more deta
  scheduler {
    # The HashedWheelTimer (HWT) implementation from Netty is used as the default scheduler
    # in the system.
    # HWT does not execute the scheduled tasks on exact time.
    # It will, on every tick, check if there are any tasks behind the schedule and execute them.
    # You can increase or decrease the accuracy of the execution timing by specifying smaller
    # or larger tick duration.
    # If you are scheduling a lot of tasks you should consider increasing the ticks per wheel.
    # For more information see: http://www.jboss.org/netty/
    tick-duration = 100ms
    ticks-per-wheel = 512
  }
}
```

akka-remote

```
####################################
# Akka Remote Reference Config File #
####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

# comments about akka.actor settings left out where they are already in akka-
# actor.jar, because otherwise they would be repeated in config rendering.

akka {

  actor {

    serializers {
      proto = "akka.serialization.ProtobufSerializer"
    }


    serialization-bindings {
      # Since com.google.protobuf.Message does not extend Serializable but GeneratedMessage
      # does, need to use the more specific one here in order to avoid ambiguity
      "com.google.protobuf.GeneratedMessage" = proto
    }

    deployment {

      default {

        # if this is set to a valid remote address, the named actor will be deployed
        # at that node e.g. "akka://sys@host:port"
        remote = ""

        target {

          # A list of hostnames and ports for instantiating the children of a
          # non-direct router
          #   The format should be on "akka://sys@host:port", where:
          #    - sys is the remote actor system name
          #    - hostname can be either hostname or IP address the remote actor
          #      should connect to
          #    - port should be the port for the remote server on the other node
          # The number of actor instances to be spawned is still taken from the
          # nr-of-instances setting as for local routers; the instances will be
          # distributed round-robin among the given nodes.
          nodes = []

        }
      }
    }
  }

  remote {

    # Which implementation of akka.remote.RemoteTransport to use
    # default is a TCP-based remote transport based on Netty
    transport = "akka.remote.netty.NettyRemoteTransport"

    # Enable untrusted mode for full security of server managed actors, allows
    # untrusted clients to connect.
    untrusted-mode = off
```

```
# Timeout for ACK of cluster operations, like checking actor out etc.
remote-daemon-ack-timeout = 30s

# If this is "on", Akka will log all inbound messages at DEBUG level, if off then they are not
log-received-messages = off

# If this is "on", Akka will log all outbound messages at DEBUG level, if off then they are no
log-sent-messages = off

# If this is "on", Akka will log all RemoteLifeCycleEvents at the level defined for each, if o
log-remote-lifecycle-events = off

# Each property is annotated with (I) or (O) or (I&O), where I stands for “inbound” and O for
# The NettyRemoteTransport always starts the server role to allow inbound connections, and it
# active client connections whenever sending to a destination which is not yet connected; if o
# it reuses inbound connections for replies, which is called a passive client connection (i.e
# to client).
netty {

  # (O) In case of increased latency / overflow how long should we wait (blocking the sender)
  # until we deem the send to be cancelled?
  # 0 means "never backoff", any positive number will indicate time to block at most.
  backoff-timeout = 0ms

  # (I&O) Generate your own with ’$AKKA_HOME/scripts/generate_config_with_secure_cookie.sh’
  # or using ’akka.util.Crypt.generateSecureCookie’
  secure-cookie = ""

  # (I) Should the remote server require that its peers share the same secure-cookie
  # (defined in the ’remote’ section)?
  require-cookie = off

  # (I) Reuse inbound connections for outbound messages
  use-passive-connections = on

  # (I) The hostname or ip to bind the remoting to,
  # InetAddress.getLocalHost.getHostAddress is used if empty
  hostname = ""

  # (I) The default remote server port clients should connect to.
  # Default is 2552 (AKKA), use 0 if you want a random available port
  # This port needs to be unique for each actor system on the same machine.
  port = 2552

  # (O) The address of a local network interface (IP Address) to bind to when creating
  # outbound connections. Set to "" or "auto" for automatic selection of local address.
  outbound-local-address = "auto"

  # (I&O) Increase this if you want to be able to send messages with large payloads
  message-frame-size = 1 MiB

  # (O) Timeout duration
  connection-timeout = 120s

  # (I) Sets the size of the connection backlog
  backlog = 4096

  # (I) Length in akka.time-unit how long core threads will be kept alive if idling
  execution-pool-keepalive = 60s

  # (I) Size of the core pool of the remote execution unit
  execution-pool-size = 4
```

```
    # (I) Maximum channel size, 0 for off
    max-channel-memory-size = 0b

    # (I) Maximum total size of all channels, 0 for off
    max-total-memory-size = 0b

    # (O) Time between reconnect attempts for active clients
    reconnect-delay = 5s

    # (O) Read inactivity period (lowest resolution is seconds)
    # after which active client connection is shutdown;
    # will be re-established in case of new communication requests.
    # A value of 0 will turn this feature off
    read-timeout = 0s

    # (O) Write inactivity period (lowest resolution is seconds)
    # after which a heartbeat is sent across the wire.
    # A value of 0 will turn this feature off
    write-timeout = 10s

    # (O) Inactivity period of both reads and writes (lowest resolution is seconds)
    # after which active client connection is shutdown;
    # will be re-established in case of new communication requests
    # A value of 0 will turn this feature off
    all-timeout = 0s

    # (O) Maximum time window that a client should try to reconnect for
    reconnection-time-window = 600s

    # (I&O) Used to configure the number of I/O worker threads on server sockets
    server-socket-worker-pool {
      # Min number of threads to cap factor-based number to
      pool-size-min = 2

      # The pool size factor is used to determine thread pool size
      # using the following formula: ceil(available processors * factor).
      # Resulting size is then bounded by the pool-size-min and
      # pool-size-max values.
      pool-size-factor = 2.0

      # Max number of threads to cap factor-based number to
      pool-size-max = 128
    }

    # (I&O) Used to configure the number of I/O worker threads on client sockets
    client-socket-worker-pool {
      # Min number of threads to cap factor-based number to
      pool-size-min = 2

      # The pool size factor is used to determine thread pool size
      # using the following formula: ceil(available processors * factor).
      # Resulting size is then bounded by the pool-size-min and
      # pool-size-max values.
      pool-size-factor = 2.0

      # Max number of threads to cap factor-based number to
      pool-size-max = 128
    }
  }

  # The dispatcher used for the system actor "network-event-sender"
  network-event-sender-dispatcher {
```

```
      executor = thread-pool-executor
      type = PinnedDispatcher
    }
  }
}
```

akka-testkit

```
#####################################
# Akka Testkit Reference Config File #
#####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  test {
    # factor by which to scale timeouts during tests, e.g. to account for shared
    # build system load
    timefactor =  1.0

    # duration of EventFilter.intercept waits after the block is finished until
    # all required messages are received
    filter-leeway = 3s

    # duration to wait in expectMsg and friends outside of within() block by default
    single-expect-default = 3s

    # The timeout that is added as an implicit by DefaultTimeout trait
    default-timeout = 5s

    calling-thread-dispatcher {
      type = akka.testkit.CallingThreadDispatcherConfigurator
    }
  }
}
```

akka-transactor

```
########################################
# Akka Transactor Reference Config File #
########################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  transactor {
    # The timeout used for coordinated transactions across actors
    coordinated-timeout = 5s
  }
}
```

akka-agent

```
###################################
# Akka Agent Reference Config File #
###################################
```

```
# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  agent {

    # The dispatcher used for agent-send-off actor
    send-off-dispatcher {
      executor = thread-pool-executor
      type = PinnedDispatcher
    }

    # The dispatcher used for agent-alter-off actor
    alter-off-dispatcher {
      executor = thread-pool-executor
      type = PinnedDispatcher
    }
  }
}
```

akka-zeromq

```
#####################################
# Akka ZeroMQ Reference Config File #
#####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {

  zeromq {

    # The default timeout for a poll on the actual zeromq socket.
    poll-timeout = 100ms

    # Timeout for creating a new socket
    new-socket-timeout = 5s

    socket-dispatcher {
      # A zeromq socket needs to be pinned to the thread that created it.
      # Changing this value results in weird errors and race conditions within zeromq
      executor = thread-pool-executor
      type = "PinnedDispatcher"
    }
  }
}
```

akka-beanstalk-mailbox

```
################################################
# Akka Beanstalk Mailboxes Reference Config File #
################################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.
#
# for more information see <https://github.com/kr/beanstalkd/blob/v1.3/doc/protocol.txt>
```

```
akka {
  actor {
    mailbox {
      beanstalk {
        # hostname to connect to
        hostname = "127.0.0.1"

        # port to connect to
        port = 11300

        # wait period in case of a connection failure before reconnect
        reconnect-window = 5s

        # integer number of seconds to wait before putting the job in
        # the ready queue. The job will be in the "delayed" state during this time.
        message-submit-delay = 0s

        # time to run -- is an integer number of seconds to allow a worker
        # to run this job. This time is counted from the moment a worker reserves
        # this job. If the worker does not delete, release, or bury the job within
        # <ttr> seconds, the job will time out and the server will release the job.
        message-time-to-live = 120s
      }
    }
  }

}
```

akka-file-mailbox

```
##############################################
# Akka File Mailboxes Reference Config File #
##############################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.
#
# For more information see <https://github.com/robey/kestrel/>

akka {
  actor {
    mailbox {
      file-based {
        # directory below which this queue resides
        directory-path = "./_mb"

        # attempting to add an item after the queue reaches this size (in items) will fail.
        max-items = 2147483647

        # attempting to add an item after the queue reaches this size (in bytes) will fail.
        max-size = 2147483647 bytes

        # attempting to add an item larger than this size (in bytes) will fail.
        max-item-size = 2147483647 bytes

        # maximum expiration time for this queue (seconds).
        max-age = 0s

        # maximum journal size before the journal should be rotated.
        max-journal-size = 16 MiB
```

```
        # maximum size of a queue before it drops into read-behind mode.
        max-memory-size = 128 MiB

        # maximum overflow (multiplier) of a journal file before we re-create it.
        max-journal-overflow = 10

        # absolute maximum size of a journal file until we rebuild it, no matter what.
        max-journal-size-absolute = 9223372036854775807 bytes

        # whether to drop older items (instead of newer) when the queue is full
        discard-old-when-full = on

        # whether to keep a journal file at all
        keep-journal = on

        # whether to sync the journal after each transaction
        sync-journal = off
      }
    }
  }
}
```

akka-mongo-mailbox

```
################################################
# Akka MongoDB Mailboxes Reference Config File #
################################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  actor {
    mailbox {
      mongodb {

        # Any specified collection name will be used as a prefix for
        # collections that use durable mongo mailboxes.
        # Follow Mongo URI Spec - http://www.mongodb.org/display/DOCS/Connections
        uri = "mongodb://localhost/akka.mailbox"

        # Configurable timeouts for certain ops
        timeout {
          # time to wait for a read to succeed before timing out the future
          read = 3000ms
          # time to wait for a write to succeed before timing out the future
          write = 3000ms
        }
      }
    }
  }
}
```

akka-redis-mailbox

```
############################################
# Akka Redis Mailboxes Reference Config File #
############################################

# This is the reference config file that contains all the default settings.
```

```
# Make your edits/overrides in your application.conf.
#
# for more information see <http://redis.io/>

akka {
  actor {
    mailbox {
      redis {
        # hostname of where the redis queue resides
        hostname = "127.0.0.1"

        # port at which the redis queue resides
        port = 6379
      }
    }
  }
}
```

akka-zookeeper-mailbox

```
##################################################
# Akka ZooKepper Mailboxes Reference Config File #
##################################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.
#
# For more information see <http://wiki.apache.org/hadoop/ZooKeeper>

akka {
  actor {
    mailbox {
      zookeeper {
        # host and port to connect to ZooKeeper
        server-addresses = "127.0.0.1:2181"

        # timeout after which an unreachable client is considered dead and its session is closed
        session-timeout = 60s

        # maximum wait period while connecting to ZooKeeper service
        connection-timeout = 60s
      }
    }
  }
}
```

2.8.5 Custom application.conf

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

```
# In this file you can override any option defined in the reference files.
# Copy in parts of the reference files and modify as you please.

akka {

  # Event handlers to register at boot time (Logging$DefaultLogger logs to STDOUT)
  event-handlers = ["akka.event.slf4j.Slf4jEventHandler"]

  # Log level used by the configured loggers (see "event-handlers") as soon
  # as they have been started; before that, see "stdout-loglevel"
```

```
  # Options: ERROR, WARNING, INFO, DEBUG
  loglevel = DEBUG

  # Log level for the very basic logger activated during AkkaApplication startup
  # Options: ERROR, WARNING, INFO, DEBUG
  stdout-loglevel = DEBUG

  actor {
    default-dispatcher {
      # Throughput for default Dispatcher, set to 1 for as fair as possible
      throughput = 10
    }
  }

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

2.8.6 Including files

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

Specifying system property with `-Dconfig.resource=/dev.conf` will load the `dev.conf` file, which includes the `application.conf`

dev.conf:

```
include "application"

akka {
  loglevel = "DEBUG"
}
```

More advanced include and substitution mechanisms are explained in the HOCON specification.

2.8.7 Logging of Configuration

If the system or config property `akka.log-config-on-start` is set to `on`, then the complete configuration at INFO level when the actor system is started. This is useful when you are uncertain of what configuration is used.

If in doubt, you can also easily and nicely inspect configuration objects before or after using them to construct an actor system:

```
Welcome to Scala version 2.9.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_27).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import com.typesafe.config._
import com.typesafe.config._

scala> ConfigFactory.parseString("a.b=12")
res0: com.typesafe.config.Config = Config(SimpleConfigObject({"a" : {"b" : 12}}))

scala> res0.root.render
```

```
res1: java.lang.String =
{
    # String: 1
    "a" : {
        # String: 1
        "b" : 12
    }
}
```

The comments preceding every item give detailed information about the origin of the setting (file & line number) plus possible comments which were present, e.g. in the reference configuration. The settings as merged with the reference and parsed by the actor system can be displayed like this:

```
final ActorSystem system = ActorSystem.create();
println(system.settings());
// this is a shortcut for system.settings().config().root().render()
```

2.8.8 A Word About ClassLoaders

In several places of the configuration file it is possible to specify the fully-qualified class name of something to be instantiated by Akka. This is done using Java reflection, which in turn uses a `ClassLoader`. Getting the right one in challenging environments like application containers or OSGi bundles is not always trivial, the current approach of Akka is that each `ActorSystem` implementation stores the current thread's context class loader (if available, otherwise just its own loader as in `this.getClass.getClassLoader`) and uses that for all reflective accesses. This implies that putting Akka on the boot class path will yield `NullPointerException` from strange places: this is simply not supported.

2.8.9 Application specific settings

The configuration can also be used for application specific settings. A good practice is to place those settings in an Extension, as described in:

- Scala API: *Application specific settings*
- Java API: *Application specific settings*

CHAPTER

THREE

COMMON UTILITIES

3.1 Duration

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.

3.1.1 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.

3.1.2 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()));
```

3.1.3 Deadline

Durations have a brother named `Deadline`, which is a class holding a representation of an absolute point in time, and support deriving a duration from this by calculating the difference between now and the deadline. This is useful when you want to keep one overall deadline without having to take care of the book-keeping wrt. the passing of time yourself:

```
val deadline = 10 seconds fromNow
// do something which takes time
awaitCond(..., deadline.timeLeft)
```

In Java you create these from durations:

```
final Deadline d = Duration.create(5, "seconds").fromNow();
```

CHAPTER

FOUR

SCALA API

4.1 Actors (Scala)

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.

4.1.1 Creating Actors

Since Akka enforces parental supervision every actor is supervised and (potentially) the supervisor of its children; it is advisable that you familiarize yourself with *Actor Systems* and *Supervision and Monitoring* and it may also help to read *Summary: actorOf vs. actorFor*.

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.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" ⇒ log.info("received test")
    case _      ⇒ log.info("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. Otherwise an `akka.actor.UnhandledMessage(message, sender, recipient)` will be published to the `ActorSystem`'s `EventStream`.

Creating Actors with default constructor

```
object Main extends App {
  val system = ActorSystem("MySystem")
  val myActor = system.actorOf(Props[MyActor], name = "myactor")
```

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.

In the above example the actor was created from the system. It is also possible to create actors from other actors with the actor `context`. The difference is how the supervisor hierarchy is arranged. When using the context the current actor will be supervisor of the created child actor. When using the system it will be a top level actor, that is supervised by the system (internal guardian actor).

```
class FirstActor extends Actor {
  val myActor = context.actorOf(Props[MyActor], name = "myactor")
```

The name parameter is optional, but you should preferably name your actors, since that is used in log messages and for identifying actors. The name must not be empty or start with `$`. If the given name is already in use by another child to the same parent actor an *InvalidActorNameException* is thrown.

> **Warning:** Creating top-level actors with `system.actorOf` is a blocking operation, hence it may deadlock due to starvation if the default dispatcher is overloaded. To avoid problems, do not call this method from within actors or futures which run on the default dispatcher.

Actors are automatically started asynchronously when created. When you create the `Actor` then it will automatically call the `preStart` callback method on the `Actor` trait. This is an excellent place to add initialization code for the actor.

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

Creating Actors with non-default constructor

If your Actor has a constructor that takes parameters then you can't create it using `actorOf(Props[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:

```
// allows passing in arguments to the MyActor constructor
val myActor = system.actorOf(Props(new MyActor("...")), name = "myactor")
```

> **Warning:** You might be tempted at times to offer an `Actor` factory which always returns the same instance, e.g. by using a `lazy val` or an `object ... extends Actor`. This is not supported, as it goes against the meaning of an actor restart, which is described here: *What Restarting Means*.

Props

`Props` is a configuration class to specify options for the creation of actors. Here are some examples on how to create a `Props` instance.

```
import akka.actor.Props
val props1 = Props()
val props2 = Props[MyActor]
```

```
val props3 = Props(new MyActor)
val props4 = Props(
  creator = { () ⇒ new MyActor },
  dispatcher = "my-dispatcher")
val props5 = props1.withCreator(new MyActor)
val props6 = props5.withDispatcher("my-dispatcher")
```

Creating Actors with Props

Actors are created by passing in a `Props` instance into the `actorOf` factory method.

```
import akka.actor.Props
val myActor = system.actorOf(Props[MyActor].withDispatcher("my-dispatcher"), name = "myactor2")
```

Creating Actors using anonymous classes

When spawning actors for specific sub-tasks from within an actor, it may be convenient to include the code to be executed directly in place, using an anonymous class.

```
def receive = {
  case m: DoIt ⇒
    context.actorOf(Props(new Actor {
      def receive = {
        case DoIt(msg) ⇒
          val replyMsg = doSomeDangerousWork(msg)
          sender ! replyMsg
          context.stop(self)
      }
      def doSomeDangerousWork(msg: ImmutableMessage): String = { "done" }
    })) forward m
}
```

Warning: In this case you need to carefully avoid closing over the containing actor's reference, i.e. do not call methods on the enclosing actor from within the anonymous Actor class. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the other actor's code will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time. See also: *Actors and shared mutable state*

4.1.2 Actor API

The `Actor` trait defines only one abstract method, the above mentioned `receive`, which implements the behavior of the actor.

If the current actor behavior does not match a received message, `unhandled` is called, which by default publishes an `akka.actor.UnhandledMessage(message, sender, recipient)` on the actor system's event stream.

In addition, it offers:

- `self` reference to the `ActorRef` of the actor
- `sender` reference sender Actor of the last received message, typically used as described in *Reply to messages*
- `supervisorStrategy` user overridable definition the strategy to use for supervising child actors
- `context` exposes contextual information for the actor and the current message, such as:
  - factory methods to create child actors (`actorOf`)

- system that the actor belongs to
- parent supervisor
- supervised children
- lifecycle monitoring
- hotswap behavior stack as described in *Become/Unbecome*

You can import the members in the `context` to avoid prefixing access with `context.`

```
class FirstActor extends Actor {
  import context._
  val myActor = actorOf(Props[MyActor], name = "myactor")
  def receive = {
    case x ⇒ myActor ! x
  }
}
```

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

```
def preStart() {}
def preRestart(reason: Throwable, message: Option[Any]) {
  context.children foreach (context.stop(_))
  postStop()
}
def postRestart(reason: Throwable) { preStart() }
def postStop() {}
```

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

Lifecycle Monitoring aka DeathWatch

In order to be notified when another actor terminates (i.e. stops permanently, not temporary failure and restart), an actor may register itself for reception of the `Terminated` message dispatched by the other actor upon termination (see Stopping Actors). This service is provided by the `DeathWatch` component of the actor system.

Registering a monitor is easy:

```
import akka.actor.{ Actor, Props, Terminated }

class WatchActor extends Actor {
  val child = context.actorOf(Props.empty, "child")
  context.watch(child) // <-- this is the only call needed for registration
  var lastSender = system.deadLetters

  def receive = {
    case "kill"                ⇒ context.stop(child); lastSender = sender
    case Terminated(`child`) ⇒ lastSender ! "finished"
  }
}
```

It should be noted that the `Terminated` message is generated independent of the order in which registration and termination occur. Registering multiple times does not necessarily lead to multiple messages being generated, but there is no guarantee that only exactly one such message is received: if termination of the watched actor has generated and queued the message, and another registration is done before this message has been processed, then a second message will be queued, because registering for monitoring of an already terminated actor leads to the immediate generation of the `Terminated` message.

It is also possible to deregister from watching another actor's liveliness using `context.unwatch(target)`, but obviously this cannot guarantee non-reception of the `Terminated` message because that may already have been queued.

Start Hook

Right after starting the actor, its `preStart` method is invoked.

```
override def preStart() {
  // registering with other actors
  someService ! Register(self)
}
```

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors will be restarted in case an exception is thrown while processing a message. This restart involves 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. By default it stops all children and calls `postStop`.
2. The initial factory from the `actorOf` call is used to produce the fresh instance.
3. The new actor's `postRestart` method is invoked with the exception which caused the restart. By default the `preStart` is called, just as in the normal start-up case.

An actor restart replaces only the actual actor object; the contents of the mailbox is unaffected by the restart, so processing of messages will resume after the `postRestart` hook returns. The message that triggered the exception will not be received again. 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. messages sent to a stopped actor will be redirected to the `deadLetters` of the `ActorSystem`.

4.1.3 Identifying Actors

As described in *Actor References, Paths and Addresses*, each actor has a unique logical path, which is obtained by following the chain of actors from child to parent until reaching the root of the actor system, and it has a physical path, which may differ if the supervision chain includes any remote supervisors. These paths are used by the system to look up actors, e.g. when a remote message is received and the recipient is searched, but they are also useful more directly: actors may look up other actors by specifying absolute or relative paths—logical or physical—and receive back an `ActorRef` with the result:

```
context.actorFor("/user/serviceA/aggregator") // will look up this absolute path
context.actorFor("../joe")                    // will look up sibling beneath same supervisor
```

The supplied path is parsed as a `java.net.URI`, which basically means that it is split on `/` into path elements. If the path starts with `/`, it is absolute and the look-up starts at the root guardian (which is the parent of `"/user"`); otherwise it starts at the current actor. If a path element equals `..`, the look-up will take a step "up" towards the supervisor of the currently traversed actor, otherwise it will step "down" to the named child. It should be noted that the `..` in actor paths here always means the logical structure, i.e. the supervisor.

If the path being looked up does not exist, a special actor reference is returned which behaves like the actor system's dead letter queue but retains its identity (i.e. the path which was looked up).

Remote actor addresses may also be looked up, if remoting is enabled:

```
context.actorFor("akka://app@otherhost:1234/user/serviceB")
```

These look-ups return a (possibly remote) actor reference immediately, so you will have to send to it and await a reply in order to verify that `serviceB` is actually reachable and running. An example demonstrating actor look-up is given in *Remote Lookup*.

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. Also known as `tell`.
- `?` sends a message asynchronously and returns a `Future` representing a possible reply. Also known as `ask`.

Message ordering is guaranteed on a per-sender basis.

Note: There are performance implications of using `ask` since something needs to keep track of when it times out, there needs to be something that bridges a `Promise` into an `ActorRef` and it also needs to be reachable through remoting. So always prefer `tell` for performance, and only `ask` if you must.

Tell: 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 `sender: ActorRef` member field. The target actor can use this to reply to the original sender, by using `sender ! replyMsg`.

If invoked from an instance that is **not** an Actor the sender will be `deadLetters` actor reference by default.

Ask: Send-And-Receive-Future

The `ask` pattern involves actors as well as futures, hence it is offered as a use pattern rather than a method on `ActorRef`:

```
import akka.pattern.{ ask, pipe }

case class Result(x: Int, s: String, d: Double)
case object Request

implicit val timeout = Timeout(5 seconds) // needed for `?` below

val f: Future[Result] =
  for {
    x ← ask(actorA, Request).mapTo[Int] // call pattern directly
    s ← actorB ask Request mapTo manifest[String] // call by implicit conversion
    d ← actorC ? Request mapTo manifest[Double] // call by symbolic name
  } yield Result(x, s, d)

f pipeTo actorD // .. or ..
pipe(f) to actorD
```

This example demonstrates `ask` together with the `pipeTo` pattern on futures, because this is likely to be a common combination. Please note that all of the above is completely non-blocking and asynchronous: `ask` produces a `Future`, three of which are composed into a new future using the for-comprehension and then `pipeTo` installs an `onComplete`-handler on the future to effect the submission of the aggregated `Result` to another actor.

Using `ask` will send a message to the receiving Actor as with `tell`, and the receiving actor must reply with `sender ! reply` in order to complete the returned `Future` with a value. The `ask` operation involves creating an internal actor for handling this reply, which needs to have a timeout after which it is destroyed in order not to leak resources; see more below.

To complete the future with an exception you need send a Failure message to the sender. This is *not done automatically* when an actor throws an exception while processing a message.

```
try {
  val result = operation()
  sender ! result
} catch {
  case e: Exception ⇒
    sender ! akka.actor.Status.Failure(e)
    throw e
}
```

If the actor does not complete the future, it will expire after the timeout period, completing it with an `AskTimeoutException`. The timeout is taken from one of the following locations in order of precedence:

1. explicitly given timeout as in:

```
import akka.util.duration._
import akka.pattern.ask
val future = myActor.ask("hello")(5 seconds)
```

2. implicit argument of type `akka.util.Timeout`, e.g.

```
import akka.util.duration._
import akka.util.Timeout
import akka.pattern.ask
implicit val timeout = Timeout(5 seconds)
val future = myActor ? "hello"
```

See *Futures (Scala)* for more information on how to await or query a future.

The `onComplete`, `onResult` methods of the `Future` can be used to register a callback to get a notification when the Future completes. Gives you a way to avoid blocking.

Warning: When using future callbacks, such as `onComplete`, `onSuccess`, and `onFailure`, inside actors you need to carefully avoid closing over the containing actor's reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time. See also: *Actors and shared mutable state*

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.

```
myActor.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:

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" ⇒ log.info("received test")
    case _      ⇒ log.info("received unknown message")
  }
}
```

4.1.7 Reply to messages

If you want to have a handle for replying to a message, you can use `sender`, which gives you an ActorRef. You can reply by sending to that ActorRef with `sender ! replyMsg`. You can also store the ActorRef for replying later, or passing on to other actors. If there is no sender (a message was sent without an actor or future context) then the sender defaults to a 'dead-letter' actor ref.

```
case request =>
  val result = process(request)
  sender ! result       // will have dead-letter actor as default
```

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 handing the ReceiveTimeout object.

```
import akka.actor.ReceiveTimeout
import akka.util.duration._
class MyActor extends Actor {
  context.setReceiveTimeout(30 milliseconds)
  def receive = {
    case "Hello"          ⇒ //...
    case ReceiveTimeout ⇒ throw new RuntimeException("received timeout")
  }
}
```

4.1.9 Stopping actors

Actors are stopped by invoking the `stop` method of a `ActorRefFactory`, i.e. `ActorContext` or `ActorSystem`. Typically the context is used for stopping child actors and the system for stopping top level actors. The actual termination of the actor is performed asynchronously, i.e. `stop` may return before the actor is stopped.

Processing of the current message, if any, will continue before the actor is stopped, but additional messages in the mailbox will not be processed. By default these messages are sent to the `deadLetters` of the `ActorSystem`, but that depends on the mailbox implementation.

Termination of an actor proceeds in two steps: first the actor suspends its mailbox processing and sends a stop command to all its children, then it keeps processing the termination messages from its children until the last one is gone, finally terminating itself (invoking `postStop`, dumping mailbox, publishing `Terminated` on the *DeathWatch*, telling its supervisor). This procedure ensures that actor system sub-trees terminate in an orderly fashion, propagating the stop command to the leaves and collecting their confirmation back to the stopped supervisor. If one of the actors does not respond (i.e. processing a message for extended periods of time and therefore not receiving the stop command), this whole process will be stuck.

Upon `ActorSystem.shutdown`, the system guardian actors will be stopped, and the aforementioned process will ensure proper termination of the whole system.

The `postStop` hook is invoked after an actor is fully stopped. This enables cleaning up of resources:

```
override def postStop() = {
  // close some file or database connection
}
```

Note: Since stopping an actor is asynchronous, you cannot immediately reuse the name of the child you just stopped; this will result in an `InvalidActorNameException`. Instead, `watch` the terminating actor and create its replacement in response to the `Terminated` message which will eventually arrive.

PoisonPill

You can also send an actor the `akka.actor.PoisonPill` message, which will stop the actor when the message is processed. `PoisonPill` is enqueued as ordinary messages and will be handled after messages that were already queued in the mailbox.

Graceful Stop

`gracefulStop` is useful if you need to wait for termination or compose ordered termination of several actors:

```
import akka.pattern.gracefulStop
import akka.dispatch.Await
import akka.actor.ActorTimeoutException

try {
  val stopped: Future[Boolean] = gracefulStop(actorRef, 5 seconds)(system)
```

```
  Await.result(stopped, 6 seconds)
  // the actor has been stopped
} catch {
  case e: ActorTimeoutException ⇒ // the actor wasn't stopped within 5 seconds
}
```

When `gracefulStop()` returns successfully, the actor's `postStop()` hook will have been executed: there exists a happens-before edge between the end of `postStop()` and the return of `gracefulStop()`.

Warning: Keep in mind that an actor stopping and its name being deregistered are separate events which happen asynchronously from each other. Therefore it may be that you will find the name still in use after `gracefulStop()` returned. In order to guarantee proper deregistration, only reuse names from within a supervisor you control and only in response to a `Terminated` message, i.e. not for top-level actors.

4.1.10 Become/Unbecome

Upgrade

Akka supports hotswapping the Actor's message loop (e.g. its implementation) at runtime: Invoke the `context.become` method from within the Actor.

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

Warning: Please note that the actor will revert to its original behavior when restarted by its Supervisor.

To hotswap the Actor behavior using `become`:

```
class HotSwapActor extends Actor {
  import context._
  def angry: Receive = {
    case "foo" ⇒ sender ! "I am already angry?"
    case "bar" ⇒ become(happy)
  }

  def happy: Receive = {
    case "bar" ⇒ sender ! "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 Hakkers.

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

```
case object Swap
class Swapper extends Actor {
  import context._
  val log = Logging(system, this)

  def receive = {
    case Swap ⇒
      log.info("Hi")
      become {
```

```
        case Swap ⇒
          log.info("Ho")
          unbecome() // resets the latest 'become' (just for fun)
      }
  }
}

object SwapperApp extends App {
  val system = ActorSystem("SwapperSystem")
  val swap = system.actorOf(Props[Swapper], name = "swapper")
  swap ! Swap // logs Hi
  swap ! Swap // logs Ho
  swap ! Swap // logs Hi
  swap ! Swap // logs Ho
  swap ! Swap // logs Hi
  swap ! Swap // logs Ho
}
```

Encoding Scala Actors nested receives without accidentally leaking memory

See this Unnested receive example.

Downgrade

Since the hotswapped code is pushed to a Stack you can downgrade the code as well, all you need to do is to: Invoke the `context.unbecome` method from within the Actor.

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

Here's how you use the `unbecome` method:

```
def receive = {
  case "revert" => context.unbecome()
}
```

4.1.11 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
```

4.1.12 Actors and exceptions

It can happen that while a message is being processed by an actor, that some kind of exception is thrown, e.g. a database exception.

What happens to the Message

If an exception is thrown while a message is being processed (so taken of his mailbox and handed over the the receive), then this message will be lost. It is important to understand that it is not put back on the mailbox. So if you want to retry processing of a message, you need to deal with it yourself by catching the exception and retry your flow. Make sure that you put a bound on the number of retries since you don't want a system to livelock (so consuming a lot of cpu cycles without making progress).

What happens to the mailbox

If an exception is thrown while a message is being processed, nothing happens to the mailbox. If the actor is restarted, the same mailbox will be there. So all messages on that mailbox, will be there as well.

What happens to the actor

If an exception is thrown, the actor instance is discarded and a new instance is created. This new instance will now be used in the actor references to this actor (so this is done invisible to the developer). Note that this means that current state of the failing actor instance is lost if you don't store and restore it in `preRestart` and `postRestart` callbacks.

4.1.13 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.

```
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
}

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

case class MyMsg(subject: String)
```

Or:

```
trait ComposableActor extends Actor {
  private var receives: List[Receive] = List()
  protected def registerReceive(receive: Receive) {
    receives = receive :: receives
  }

  def receive = receives reduce { _ orElse _ }
}

class MyComposableActor extends ComposableActor {
  override def preStart() {
    registerReceive({
      case "foo" ⇒ /* Do something */
    })

    registerReceive({
      case "bar" ⇒ /* Do something */
    })
  }
}
```

4.2 Typed Actors (Scala)

Akka Typed Actors is an implementation of the Active Objects pattern. Essentially turning method invocations into asynchronous dispatch instead of synchronous that has been the default way since Smalltalk came out.

Typed Actors consist of 2 "parts", a public interface and an implementation, and if you've done any work in "enterprise" Java, this will be very familiar to you. As with normal Actors you have an external API (the public interface instance) that will delegate methodcalls asynchronously to a private instance of the implementation.

The advantage of Typed Actors vs. Actors is that with TypedActors you have a static contract, and don't need to define your own messages, the downside is that it places some limitations on what you can do and what you can't, i.e. you can't use become/unbecome.

Typed Actors are implemented using JDK Proxies which provide a pretty easy-worked API to intercept method calls.

Note: Just as with regular Akka Actors, Typed Actors process one call at a time.

4.2.1 When to use Typed Actors

Typed actors are nice for bridging between actor systems (the "inside") and non-actor code (the "outside"), because they allow you to write normal OO-looking code on the outside. Think of them like doors: their practicality lies in interfacing between private sphere and the public, but you don't want that many doors inside your house, do you? For a longer discussion see this blog post.

A bit more background: TypedActors can very easily be abused as RPC, and that is an abstraction which is well-known to be leaky. Hence TypedActors are not what we think of first when we talk about making highly scalable concurrent software easier to write correctly. They have their niche, use them sparingly.

4.2.2 The tools of the trade

Before we create our first Typed Actor we should first go through the tools that we have at our disposal, it's located in `akka.actor.TypedActor`.

```
import akka.actor.TypedActor

//Returns the Typed Actor Extension
val extension = TypedActor(system) //system is an instance of ActorSystem

//Returns whether the reference is a Typed Actor Proxy or not
TypedActor(system).isTypedActor(someReference)

//Returns the backing Akka Actor behind an external Typed Actor Proxy
TypedActor(system).getActorRefFor(someReference)

//Returns the current ActorContext,
// method only valid within methods of a TypedActor implementation
val c: ActorContext = TypedActor.context

//Returns the external proxy of the current Typed Actor,
// method only valid within methods of a TypedActor implementation
val s: Squarer = TypedActor.self[Squarer]

//Returns a contextual instance of the Typed Actor Extension
//this means that if you create other Typed Actors with this,
//they will become children to the current Typed Actor.
TypedActor(TypedActor.context)
```

Warning: Same as not exposing `this` of an Akka Actor, it's important not to expose `this` of a Typed Actor, instead you should pass the external proxy reference, which is obtained from within your Typed Actor as `TypedActor.self`, this is your external identity, as the `ActorRef` is the external identity of an Akka Actor.

4.2.3 Creating Typed Actors

To create a Typed Actor you need to have one or more interfaces, and one implementation.

Our example interface:

```
import akka.dispatch.{ Promise, Future, Await }
import akka.util.duration._
import akka.actor.{ ActorContext, TypedActor, TypedProps }

trait Squarer {
  // typed actor iface methods ...
}
```

Our example implementation of that interface:

```
import akka.dispatch.{ Promise, Future, Await }
import akka.util.duration._
import akka.actor.{ ActorContext, TypedActor, TypedProps }

class SquarerImpl(val name: String) extends Squarer {

  def this() = this("default")
  // typed actor impl methods ...
}
```

The most trivial way of creating a Typed Actor instance of our Squarer:

```
val mySquarer: Squarer =
  TypedActor(system).typedActorOf(TypedProps[SquarerImpl]())
```

First type is the type of the proxy, the second type is the type of the implementation. If you need to call a specific constructor you do it like this:

```
val otherSquarer: Squarer =
  TypedActor(system).typedActorOf(TypedProps(classOf[Squarer], new SquarerImpl("foo")), "name")
```

Since you supply a Props, you can specify which dispatcher to use, what the default timeout should be used and more. Now, our Squarer doesn't have any methods, so we'd better add those.

```
import akka.dispatch.{ Promise, Future, Await }
import akka.util.duration._
import akka.actor.{ ActorContext, TypedActor, TypedProps }

trait Squarer {
  def squareDontCare(i: Int): Unit //fire-forget

  def square(i: Int): Future[Int] //non-blocking send-request-reply

  def squareNowPlease(i: Int): Option[Int] //blocking send-request-reply

  def squareNow(i: Int): Int //blocking send-request-reply
}
```

Alright, now we've got some methods we can call, but we need to implement those in SquarerImpl.

```
import akka.dispatch.{ Promise, Future, Await }
import akka.util.duration._
import akka.actor.{ ActorContext, TypedActor, TypedProps }

class SquarerImpl(val name: String) extends Squarer {

  def this() = this("default")
  import TypedActor.dispatcher //So we can create Promises

  def squareDontCare(i: Int): Unit = i * i //Nobody cares :(

  def square(i: Int): Future[Int] = Promise successful i * i

  def squareNowPlease(i: Int): Option[Int] = Some(i * i)

  def squareNow(i: Int): Int = i * i
}
```

Excellent, now we have an interface and an implementation of that interface, and we know how to create a Typed Actor from that, so let's look at calling these methods.

4.2.4 Method dispatch semantics

Methods returning:

- `Unit` will be dispatched with `fire-and-forget` semantics, exactly like `ActorRef.tell`
- `akka.dispatch.Future[_]` will use `send-request-reply` semantics, exactly like `ActorRef.ask`
- `scala.Option[_]` or `akka.japi.Option<?>` will use `send-request-reply` semantics, but *will* block to wait for an answer, and return None if no answer was produced within the timout, or scala.Some/akka.japi.Some containing the result otherwise. Any exception that was thrown during this call will be rethrown.
- Any other type of value will use `send-request-reply` semantics, but *will* block to wait for an answer, throwing `java.util.concurrent.TimeoutException` if there was a timeout or rethrow any exception that was thrown during this call.

4.2.5 Messages and immutability

While Akka cannot enforce that the parameters to the methods of your Typed Actors are immutable, we *strongly* recommend that parameters passed are immutable.

One-way message send

```
mySquarer.squareDontCare(10)
```

As simple as that! The method will be executed on another thread; asynchronously.

Request-reply message send

```
val oSquare = mySquarer.squareNowPlease(10) //Option[Int]
```

This will block for as long as the timeout that was set in the Props of the Typed Actor, if needed. It will return `None` if a timeout occurs.

```
val iSquare = mySquarer.squareNow(10) //Int
```

This will block for as long as the timeout that was set in the Props of the Typed Actor, if needed. It will throw a `java.util.concurrent.TimeoutException` if a timeout occurs.

Request-reply-with-future message send

```
val fSquare = mySquarer.square(10) //A Future[Int]
```

This call is asynchronous, and the Future returned can be used for asynchronous composition.

4.2.6 Stopping Typed Actors

Since Akkas Typed Actors are backed by Akka Actors they must be stopped when they aren't needed anymore.

```
TypedActor(system).stop(mySquarer)
```

This asynchronously stops the Typed Actor associated with the specified proxy ASAP.

```
TypedActor(system).poisonPill(otherSquarer)
```

This asynchronously stops the Typed Actor associated with the specified proxy after it's done with all calls that were made prior to this call.

4.2.7 Typed Actor Hierarchies

Since you can obtain a contextual Typed Actor Extension by passing in an `ActorContext` you can create child Typed Actors by invoking `typedActorOf(..)` on that.

This also works for creating child Typed Actors in regular Akka Actors.

4.2.8 Supervisor Strategy

By having your Typed Actor implementation class implement `TypedActor.Supervisor` you can define the strategy to use for supervising child actors, as described in *Supervision and Monitoring* and *Fault Tolerance (Scala)*.

4.2.9 Lifecycle callbacks

By having your Typed Actor implementation class implement any and all of the following:

- `TypedActor.PreStart`
- `TypedActor.PostStop`
- `TypedActor.PreRestart`
- `TypedActor.PostRestart`

You can hook into the lifecycle of your Typed Actor.

4.2.10 Receive arbitrary messages

If your implementation class of your TypedActor extends `akka.actor.TypedActor.Receiver`, all messages that are not `MethodCall``s will be passed into the ``onReceive`-method.

This allows you to react to DeathWatch `Terminated`-messages and other types of messages, e.g. when interfacing with untyped actors.

4.2.11 Proxying

You can use the `typedActorOf` that takes a TypedProps and an ActorRef to proxy the given ActorRef as a TypedActor. This is usable if you want to communicate remotely with TypedActors on other machines, just look them up with `actorFor` and pass the `ActorRef` to `typedActorOf`.

Note: The ActorRef needs to accept `MethodCall` messages.

4.2.12 Lookup & Remoting

Since `TypedActors` are backed by `Akka Actors`, you can use `actorFor` together with `typedActorOf` to proxy `ActorRefs` potentially residing on remote nodes.

```
val typedActor: Foo with Bar =
  TypedActor(system).
    typedActorOf(
      TypedProps[FooBar],
      system.actorFor("akka://SomeSystem@somehost:2552/user/some/foobar"))
//Use "typedActor" as a FooBar
```

4.2.13 Supercharging

Here's an example on how you can use traits to mix in behavior in your Typed Actors.

```
trait Foo {
  def doFoo(times: Int): Unit = println("doFoo(" + times + ")")
}

trait Bar {
  import TypedActor.dispatcher //So we have an implicit dispatcher for our Promise
  def doBar(str: String): Future[String] = Promise successful str.toUpperCase
}

class FooBar extends Foo with Bar
```

```
val awesomeFooBar: Foo with Bar = TypedActor(system).typedActorOf(TypedProps[FooBar]())

awesomeFooBar.doFoo(10)
val f = awesomeFooBar.doBar("yes")

TypedActor(system).poisonPill(awesomeFooBar)
```

4.3 Logging (Scala)

4.3.1 How to Log

Create a `LoggingAdapter` and use the `error`, `warning`, `info`, or `debug` methods, as illustrated in this example:

```
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)
  override def preStart() = {
    log.debug("Starting")
  }
```

```
  override def preRestart(reason: Throwable, message: Option[Any]) {
    log.error(reason, "Restarting due to [{}] when processing [{}]",
      reason.getMessage, message.getOrElse(""))
  }
  def receive = {
    case "test" ⇒ log.info("Received test")
    case x      ⇒ log.warning("Received unknown message: {}", x)
  }
}
```

For convenience you can mixin the `log` member into actors, instead of defining it as above.

```
class MyActor extends Actor with akka.actor.ActorLogging {
  ...
}
```

The second parameter to the `Logging` is the source of this logging channel. The source object is translated to a String according to the following rules:

- if it is an Actor or ActorRef, its path is used
- in case of a String it is used as is
- in case of a class an approximation of its simpleName
- and in all other cases a compile error occurs unless and implicit `LogSource[T]` is in scope for the type in question.

The log message may contain argument placeholders `{}`, which will be substituted if the log level is enabled. Giving more arguments as there are placeholders results in a warning being appended to the log statement (i.e. on the same line with the same severity). You may pass a Java array as the only substitution argument to have its elements be treated individually:

```
val args = Array("The", "brown", "fox", "jumps", 42)
system.log.debug("five parameters: {}, {}, {}, {}, {}", args)
```

The Java `Class` of the log source is also included in the generated `LogEvent`. In case of a simple string this is replaced with a "marker" class `akka.event.DummyClassForStringSources` in order to allow special treatment of this case, e.g. in the SLF4J event listener which will then use the string instead of the class' name for looking up the logger instance to use.

Auxiliary logging options

Akka has a couple of configuration options for very low level debugging, that makes most sense in for developers and not for operations.

You almost definitely need to have logging set to DEBUG to use any of the options below:

```
akka {
  loglevel = DEBUG
}
```

This config option is very good if you want to know what config settings are loaded by Akka:

```
akka {
  # Log the complete configuration at INFO level when the actor system is started.
  # This is useful when you are uncertain of what configuration is used.
  log-config-on-start = on
}
```

If you want very detailed logging of all user-level messages that are processed by Actors that use akka.event.LoggingReceive:

```
akka {
  actor {
    debug {
      # enable function of LoggingReceive, which is to log any received message at
      # DEBUG level
      receive = on
    }
  }
}
```

If you want very detailed logging of all automatically received messages that are processed by Actors:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of all AutoReceiveMessages (Kill, PoisonPill and the like)
      autoreceive = on
    }
  }
}
```

If you want very detailed logging of all lifecycle changes of Actors (restarts, deaths etc):

```
akka {
  actor {
    debug {
      # enable DEBUG logging of actor lifecycle changes
      lifecycle = on
    }
  }
}
```

If you want very detailed logging of all events, transitions and timers of FSM Actors that extend LoggingFSM:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of all LoggingFSMs for events, transitions and timers
      fsm = on
    }
  }
}
```

If you want to monitor subscriptions (subscribe/unsubscribe) on the ActorSystem.eventStream:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of subscription changes on the eventStream
      event-stream = on
    }
  }
}
```

Auxiliary remote logging options

If you want to see all messages that are sent through remoting at DEBUG log level: (This is logged as they are sent by the transport layer, not by the Actor)

```
akka {
  remote {
    # If this is "on", Akka will log all outbound messages at DEBUG level, if off then they are no
    log-sent-messages = on
```

```
  }
}
```

If you want to see all messages that are received through remoting at DEBUG log level: (This is logged as they are received by the transport layer, not by any Actor)

```
akka {
  remote {
    # If this is "on", Akka will log all inbound messages at DEBUG level, if off then they are not
    log-received-messages = on
  }
}
```

Also see the logging options for TestKit: *Tracing Actor Invocations*.

Translating Log Source to String and Class

The rules for translating the source object to the source string and class which are inserted into the `LogEvent` during runtime are implemented using implicit parameters and thus fully customizable: simply create your own instance of `LogSource[T]` and have it in scope when creating the logger.

```
import akka.event.LogSource
import akka.actor.ActorSystem

object MyType {
  implicit val logSource: LogSource[AnyRef] = new LogSource[AnyRef] {
    def genString(o: AnyRef): String = o.getClass.getName
    override def getClazz(o: AnyRef): Class[_] = o.getClass
  }
}

class MyType(system: ActorSystem) {
  import MyType._
  import akka.event.Logging

  val log = Logging(system, this)
}
```

This example creates a log source which mimics traditional usage of Java loggers, which are based upon the originating object's class name as log category. The override of `getClazz` is only included for demonstration purposes as it contains exactly the default behavior.

Note: You may also create the string representation up front and pass that in as the log source, but be aware that then the `Class[_]` which will be put in the `LogEvent` is `akka.event.DummyClassForStringSources`.

The SLF4J event listener treats this case specially (using the actual string to look up the logger instance to use instead of the class' name), and you might want to do this also in case you implement your own logging adapter.

4.3.2 Event Handler

Logging is performed asynchronously through an event bus. You can configure which event handlers that should subscribe to the logging events. That is done using the `event-handlers` element in the *Configuration*. Here you can also define the log level.

```
akka {
  # Event handlers to register at boot time (Logging$DefaultLogger logs to STDOUT)
  event-handlers = ["akka.event.Logging$DefaultLogger"]
  # Options: ERROR, WARNING, INFO, DEBUG
```

```
  loglevel = "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:

```
import akka.event.Logging.InitializeLogger
import akka.event.Logging.LoggerInitialized
import akka.event.Logging.Error
import akka.event.Logging.Warning
import akka.event.Logging.Info
import akka.event.Logging.Debug

class MyEventListener extends Actor {
  def receive = {
    case InitializeLogger(_)                          ⇒ sender ! LoggerInitialized
    case Error(cause, logSource, logClass, message) ⇒ // ...
    case Warning(logSource, logClass, message)      ⇒ // ...
    case Info(logSource, logClass, message)         ⇒ // ...
    case Debug(logSource, logClass, message)        ⇒ // ...
  }
}
```

4.3.3 SLF4J

Akka provides an event handler for SL4FJ. 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 Logback:

```
    lazy val logback = "ch.qos.logback" % "logback-classic" % "1.0.0" % "runtime"
```

You need to enable the Slf4jEventHandler in the 'event-handlers' element in the *Configuration*. Here you can also define the log level of the event bus. More fine grained log levels can be defined in the configuration of the SLF4J backend (e.g. logback.xml).

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

The SLF4J logger selected for each log event is chosen based on the `Class[_]` of the log source specified when creating the `LoggingAdapter`, unless that was given directly as a string in which case that string is used (i.e. `LoggerFactory.getLogger(c: Class[_])` is used in the first case and `LoggerFactory.getLogger(s: String)` in the second).

Note: Beware that the the actor system's name is appended to a `String` log source if the LoggingAdapter was created giving an `ActorSystem` to the factory. If this is not intended, give a `LoggingBus` instead as shown below:

```
val log = Logging(system.eventStream, "my.nice.string")
```

Logging Thread and Akka Source in MDC

Since the logging is done asynchronously the thread in which the logging was performed is captured in Mapped Diagnostic Context (MDC) with attribute name `sourceThread`. With Logback the thread name is available with `%X{sourceThread}` specifier within the pattern layout configuration:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%date{ISO8601} %-5level %logger{36} %X{sourceThread} - %msg%n</pattern>
  </encoder>
</appender>
```

Note: It will probably be a good idea to use the `sourceThread` MDC value also in non-Akka parts of the application in order to have this property consistently available in the logs.

Another helpful facility is that Akka captures the actor's address when instantiating a logger within it, meaning that the full instance identification is available for associating log messages e.g. with members of a router. This information is available in the MDC with attribute name `akkaSource`:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%date{ISO8601} %-5level %logger{36} %X{akkaSource} - %msg%n</pattern>
  </encoder>
</appender>
```

For more details on what this attribute contains—also for non-actors—please see How to Log.

4.4 Event Bus (Scala)

Originally conceived as a way to send messages to groups of actors, the `EventBus` has been generalized into a set of composable traits implementing a simple interface:

- `subscribe(subscriber: Subscriber, classifier: Classifier): Boolean` subscribes the given subscriber to events with the given classifier
- `unsubscribe(subscriber: Subscriber, classifier: Classifier): Boolean` undoes a specific subscription
- `unsubscribe(subscriber: Subscriber)` undoes all subscriptions for the given subscriber
- `publish(event: Event)` publishes an event, which first is classified according to the specific bus (see Classifiers) and then published to all subscribers for the obtained classifier

This mechanism is used in different places within Akka, e.g. the *DeathWatch* and the Event Stream. Implementations can make use of the specific building blocks presented below.

An event bus must define the following three abstract types:

- `Event` is the type of all events published on that bus
- `Subscriber` is the type of subscribers allowed to register on that event bus
- `Classifier` defines the classifier to be used in selecting subscribers for dispatching events

The traits below are still generic in these types, but they need to be defined for any concrete implementation.

4.4.1 Classifiers

The classifiers presented here are part of the Akka distribution, but rolling your own in case you do not find a perfect match is not difficult, check the implementation of the existing ones on github.

Lookup Classification

The simplest classification is just to extract an arbitrary classifier from each event and maintaining a set of subscribers for each possible classifier. This can be compared to tuning in on a radio station. The trait

`LookupClassification` is still generic in that it abstracts over how to compare subscribers and how exactly to classify. The necessary methods to be implemented are the following:

- `classify(event: Event): Classifier` is used for extracting the classifier from the incoming events.
- `compareSubscribers(a: Subscriber, b: Subscriber): Int` must define a partial order over the subscribers, expressed as expected from `java.lang.Comparable.compare`.
- `publish(event: Event, subscriber: Subscriber)` will be invoked for each event for all subscribers which registered themselves for the event's classifier.
- `mapSize: Int` determines the initial size of the index data structure used internally (i.e. the expected number of different classifiers).

This classifier is efficient in case no subscribers exist for a particular event.

Subchannel Classification

If classifiers form a hierarchy and it is desired that subscription be possible not only at the leaf nodes, this classification may be just the right one. It can be compared to tuning in on (possibly multiple) radio channels by genre. This classification has been developed for the case where the classifier is just the JVM class of the event and subscribers may be interested in subscribing to all subclasses of a certain class, but it may be used with any classifier hierarchy. The abstract members needed by this classifier are

- `subclassification: Subclassification[Classifier]` is an object providing `isEqual(a: Classifier, b: Classifier)` and `isSubclass(a: Classifier, b: Classifier)` to be consumed by the other methods of this classifier.
- `classify(event: Event): Classifier` is used for extracting the classifier from the incoming events.
- `publish(event: Event, subscriber: Subscriber)` will be invoked for each event for all subscribers which registered themselves for the event's classifier.

This classifier is also efficient in case no subscribers are found for an event, but it uses conventional locking to synchronize an internal classifier cache, hence it is not well-suited to use cases in which subscriptions change with very high frequency (keep in mind that "opening" a classifier by sending the first message will also have to re-check all previous subscriptions).

Scanning Classification

The previous classifier was built for multi-classifier subscriptions which are strictly hierarchical, this classifier is useful if there are overlapping classifiers which cover various parts of the event space without forming a hierarchy. It can be compared to tuning in on (possibly multiple) radio stations by geographical reachability (for old-school radio-wave transmission). The abstract members for this classifier are:

- `compareClassifiers(a: Classifier, b: Classifier): Int` is needed for determining matching classifiers and storing them in an ordered collection.
- `compareSubscribers(a: Subscriber, b: Subscriber): Int` is needed for storing subscribers in an ordered collection.
- `matches(classifier: Classifier, event: Event): Boolean` determines whether a given classifier shall match a given event; it is invoked for each subscription for all received events, hence the name of the classifier.
- `publish(event: Event, subscriber: Subscriber)` will be invoked for each event for all subscribers which registered themselves for a classifier matching this event.

This classifier takes always a time which is proportional to the number of subscriptions, independent of how many actually match.

Actor Classification

This classification has been developed specifically for implementing *DeathWatch*: subscribers as well as classifiers are of type `ActorRef`. The abstract members are

- `classify(event: Event): ActorRef` is used for extracting the classifier from the incoming events.
- `mapSize: Int` determines the initial size of the index data structure used internally (i.e. the expected number of different classifiers).

This classifier is still is generic in the event type, and it is efficient for all use cases.

4.4.2 Event Stream

The event stream is the main event bus of each actor system: it is used for carrying *log messages* and Dead Letters and may be used by the user code for other purposes as well. It uses Subchannel Classification which enables registering to related sets of channels (as is used for `RemoteLifeCycleMessage`). The following example demonstrates how a simple subscription works:

```
import akka.actor.{ Actor, DeadLetter, Props }

val listener = system.actorOf(Props(new Actor {
  def receive = {
    case d: DeadLetter ⇒ println(d)
  }
}))
system.eventStream.subscribe(listener, classOf[DeadLetter])
```

Default Handlers

Upon start-up the actor system creates and subscribes actors to the event stream for logging: these are the handlers which are configured for example in `application.conf`:

```
akka {
  event-handlers = ["akka.event.Logging$DefaultLogger"]
}
```

The handlers listed here by fully-qualified class name will be subscribed to all log event classes with priority higher than or equal to the configured log-level and their subscriptions are kept in sync when changing the log-level at runtime:

```
system.eventStream.setLogLevel(Logging.DebugLevel)
```

This means that log events for a level which will not be logged are typically not dispatched at all (unless manual subscriptions to the respective event class have been done)

Dead Letters

As described at *Stopping actors*, messages queued when an actor terminates or sent after its death are re-routed to the dead letter mailbox, which by default will publish the messages wrapped in `DeadLetter`. This wrapper holds the original sender, receiver and message of the envelope which was redirected.

Other Uses

The event stream is always there and ready to be used, just publish your own events (it accepts `AnyRef`) and subscribe listeners to the corresponding JVM classes.

4.5 Scheduler (Scala)

Sometimes the need for making things happen in the future arises, and where do you go look then? Look no further than `ActorSystem`! There you find the `scheduler` method that returns an instance of akka.actor.Scheduler, this instance is unique per ActorSystem and is used internally for scheduling things to happen at specific points in time. Please note that the scheduled tasks are executed by the default `MessageDispatcher` of the `ActorSystem`.

You can schedule sending of messages to actors and execution of tasks (functions or Runnable). You will get a `Cancellable` back that you can call `cancel` on to cancel the execution of the scheduled operation.

Warning: The default implementation of `Scheduler` used by Akka is based on the Netty `HashedWheelTimer`. It does not execute tasks at the exact time, but on every tick, it will run everything that is overdue. The accuracy of the default Scheduler can be modified by the "ticks-per-wheel" and "tick-duration" configuration properties. For more information, see: HashedWheelTimers.

4.5.1 Some examples

```
import akka.actor.Actor
import akka.actor.Props
import akka.util.duration._

    //Schedules to send the "foo"-message to the testActor after 50ms
    system.scheduler.scheduleOnce(50 milliseconds, testActor, "foo")
```

```
//Schedules a function to be executed (send the current time) to the testActor after 50ms
system.scheduler.scheduleOnce(50 milliseconds) {
  testActor ! System.currentTimeMillis
}
```

```
val Tick = "tick"
val tickActor = system.actorOf(Props(new Actor {
  def receive = {
    case Tick ⇒ //Do something
  }
}))
//This will schedule to send the Tick-message
//to the tickActor after 0ms repeating every 50ms
val cancellable =
  system.scheduler.schedule(0 milliseconds,
    50 milliseconds,
    tickActor,
    Tick)

//This cancels further Ticks to be sent
cancellable.cancel()
```

4.5.2 From `akka.actor.ActorSystem`

```
/**
 * Light-weight scheduler for running asynchronous tasks after some deadline
 * in the future. Not terribly precise but cheap.
 */
def scheduler: Scheduler
```

4.5.3 The Scheduler interface

```
/**
 * An Akka scheduler service. This one needs one special behavior: if
 * Closeable, it MUST execute all outstanding tasks upon .close() in order
 * to properly shutdown all dispatchers.
 *
 * Furthermore, this timer service MUST throw IllegalStateException if it
 * cannot schedule a task. Once scheduled, the task MUST be executed. If
 * executed upon close(), the task may execute before its timeout.
 */
trait Scheduler {
  /**
   * Schedules a message to be sent repeatedly with an initial delay and
   * frequency. E.g. if you would like a message to be sent immediately and
   * thereafter every 500ms you would set delay=Duration.Zero and
   * frequency=Duration(500, TimeUnit.MILLISECONDS)
   *
   * Java & Scala API
   */
  def schedule(
    initialDelay: Duration,
    frequency: Duration,
    receiver: ActorRef,
    message: Any): Cancellable

  /**
   * Schedules a function to be run repeatedly with an initial delay and a
   * frequency. E.g. if you would like the function to be run after 2 seconds
   * and thereafter every 100ms you would set delay = Duration(2, TimeUnit.SECONDS)
   * and frequency = Duration(100, TimeUnit.MILLISECONDS)
   *
   * Scala API
   */
  def schedule(
    initialDelay: Duration, frequency: Duration)(f: ⇒ Unit): Cancellable

  /**
   * Schedules a function to be run repeatedly with an initial delay and
   * a frequency. E.g. if you would like the function to be run after 2
   * seconds and thereafter every 100ms you would set delay = Duration(2,
   * TimeUnit.SECONDS) and frequency = Duration(100, TimeUnit.MILLISECONDS)
   *
   * Java API
   */
  def schedule(
    initialDelay: Duration, frequency: Duration, runnable: Runnable): Cancellable

  /**
   * Schedules a Runnable to be run once with a delay, i.e. a time period that
   * has to pass before the runnable is executed.
   *
   * Java & Scala API
   */
  def scheduleOnce(delay: Duration, runnable: Runnable): Cancellable

  /**
   * Schedules a message to be sent once with a delay, i.e. a time period that has
   * to pass before the message is sent.
   *
   * Java & Scala API
   */
  def scheduleOnce(delay: Duration, receiver: ActorRef, message: Any): Cancellable
```

```
  /**
   * Schedules a function to be run once with a delay, i.e. a time period that has
   * to pass before the function is run.
   *
   * Scala API
   */
  def scheduleOnce(delay: Duration)(f: ⇒ Unit): Cancellable
}
```

4.5.4 The Cancellable interface

This allows you to `cancel` something that has been scheduled for execution.

Warning: This does not abort the execution of the task, if it had already been started.

```
/**
 * Signifies something that can be cancelled
 * There is no strict guarantee that the implementation is thread-safe,
 * but it should be good practice to make it so.
 */
trait Cancellable {
  /**
   * Cancels this Cancellable
   *
   * Java & Scala API
   */
  def cancel(): Unit

  /**
   * Returns whether this Cancellable has been cancelled
   *
   * Java & Scala API
   */
  def isCancelled: Boolean
}
```

4.6 Futures (Scala)

4.6.1 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).

4.6.2 Execution Contexts

In order to execute callbacks and operations, Futures need something called an `ExecutionContext`, which is very similar to a `java.util.concurrent.Executor`. if you have an `ActorSystem` in scope, it will use its default dispatcher as the `ExecutionContext`, or you can use the factory methods provided by the `ExecutionContext` companion object to wrap `Executors` and `ExecutorServices`, or even create your own.

```
import akka.dispatch.{ ExecutionContext, Promise }

implicit val ec = ExecutionContext.fromExecutorService(yourExecutorServiceGoesHere)
```

```
// Do stuff with your brand new shiny ExecutionContext
val f = Promise.successful("foo")

// Then shut your ExecutionContext down at some
// appropriate place in your program/application
ec.shutdown()
```

4.6.3 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:

```
import akka.dispatch.Await
import akka.pattern.ask
import akka.util.Timeout
import akka.util.duration._

implicit val timeout = Timeout(5 seconds)
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

This will cause the current thread to block and wait for the `Actor` to 'complete' the `Future` with it's reply. Blocking is discouraged though as it will cause performance problems. The blocking operations are located in `Await.result` and `Await.ready` to make it easy to spot where blocking occurs. Alternatives to blocking are discussed further within this documentation. Also note that the `Future` returned by an `Actor` is a `Future[Any]` since an `Actor` is dynamic. That is why the `asInstanceOf` is used in the above sample. When using non-blocking it is better to use the `mapTo` method to safely try to cast a `Future` to an expected type:

```
import akka.dispatch.Future
import akka.pattern.ask

val future: Future[String] = ask(actor, msg).mapTo[String]
```

The `mapTo` method will return a new `Future` that contains the result if the cast was successful, or a `ClassCastException` if not. Handling `Exception`s will be discussed further within this documentation.

4.6.4 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 `Actor`s for the sole reason of performing a calculation in parallel, there is an easier (and faster) way:

```
import akka.dispatch.Await
import akka.dispatch.Future
import akka.util.duration._

val future = Future {
  "Hello" + "World"
}
val result = Await.result(future, 1 second)
```

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`.

You can also create already completed Futures using the `Promise` companion, which can be either successes:

```
val future = Promise.successful("Yay!")
```

Or failures:

```
val otherFuture = Promise.failed[String](new IllegalArgumentException("Bang!"))
```

4.6.5 Functional Futures

Akka's `Future` has 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 = Await.result(f2, 1 second)
result must be(10)
f1.value must be(Some(Right("HelloWorld")))
```

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 its 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 = Promise.successful(3)
val f3 = f1 map { x ⇒
  f2 map { y ⇒
    x.length * y
  }
}
```

`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 = Promise.successful(3)
val f3 = f1 flatMap { x ⇒
  f2 map { y ⇒
    x.length * y
  }
}
val result = Await.result(f3, 1 second)
result must be(30)
```

Composing futures using nested combinators it can sometimes become quite complicated and hard read, in these cases using Scala's 'for comprehensions' usually yields more readable code. See next section for examples.

If you need to do conditional propagation, you can use `filter`:

```
val future1 = Promise.successful(4)
val future2 = future1.filter(_ % 2 == 0)
val result = Await.result(future2, 1 second)
result must be(4)

val failedFilter = future1.filter(_ % 2 == 1).recover {
  case m: MatchError ⇒ 0 //When filter fails, it will have a MatchError
}
val result2 = Await.result(failedFilter, 1 second)
result2 must be(0) //Can only be 0 when there was a MatchError
```

For Comprehensions

Since `Future` has a `map`, `filter` 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
  if c > 3 // Future.filter
} yield b * c //  6 * 4 = 24

// Note that the execution of futures a, b, and c
// are not done in parallel.

val result = Await.result(f, 1 second)
result must be(24)
```

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 `Await.result` or `Await.ready` to block for each result. First an example of using `Await.result`:

```
val f1 = ask(actor1, msg1)
val f2 = ask(actor2, msg2)

val a = Await.result(f1, 1 second).asInstanceOf[Int]
val b = Await.result(f2, 1 second).asInstanceOf[Int]

val f3 = ask(actor3, (a + b))

val result = Await.result(f3, 1 second).asInstanceOf[Int]
```

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

```
val f1 = ask(actor1, msg1)
val f2 = ask(actor2, msg2)
```

```
val f3 = for {
  a ← f1.mapTo[Int]
  b ← f2.mapTo[Int]
  c ← ask(actor3, (a + b)).mapTo[Int]
} yield c

val result = Await.result(f3, 1 second).asInstanceOf[Int]
```

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 then 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 as a List[Future[Int]]
val listOfFutures = List.fill(100)(akka.pattern.ask(oddActor, GetNext).mapTo[Int])

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

// Find the sum of the odd numbers
val oddSum = Await.result(futureList.map(_.sum), 1 second).asInstanceOf[Int]
oddSum must be(10000)
```

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 `A => 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 futureList = Future.traverse((1 to 100).toList)(x ⇒ Future(x * 2 - 1))
val oddSum = Await.result(futureList.map(_.sum), 1 second).asInstanceOf[Int]
oddSum must be(10000)
```

This is the same result as this example:

```
val futureList = Future.sequence((1 to 100).toList.map(x ⇒ Future(x * 2 - 1)))
val oddSum = Await.result(futureList.map(_.sum), 1 second).asInstanceOf[Int]
oddSum must be(10000)
```

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, asynchronously, the execution will start when the last of the Futures is completed.

```
val futures = for (i ← 1 to 1000) yield Future(i * 2) // Create a sequence of Futures
val futureSum = Future.fold(futures)(0)(_ + _)
Await.result(futureSum, 1 second) must be(1001000)
```

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 = Future.reduce(futures)(_ + _)
Await.result(futureSum, 1 second) must be(1001000)
```

Same as with `fold`, the execution will be done asynchronously when the last of the Future is completed, you can also parallelize it by chunking your futures into sub-sequences and reduce them, and then reduce the reduced results again.

4.6.6 Callbacks

Sometimes you just want to listen to a `Future` being completed, and react to that not by creating a new Future, but by side-effecting. For this Akka supports `onComplete`, `onSuccess` and `onFailure`, of which the latter two are specializations of the first.

```
future onSuccess {
  case "bar"       ⇒ println("Got my bar alright!")
  case x: String ⇒ println("Got some random string: " + x)
}
```

```
future onFailure {
  case ise: IllegalStateException if ise.getMessage == "OHNOES" ⇒
  //OHNOES! We are in deep trouble, do something!
  case e: Exception ⇒
  //Do something else
}
```

```
future onComplete {
  case Right(result) ⇒ doSomethingOnSuccess(result)
  case Left(failure) ⇒ doSomethingOnFailure(failure)
}
```

4.6.7 Define Ordering

Since callbacks are executed in any order and potentially in parallel, it can be tricky at the times when you need sequential ordering of operations. But there's a solution and it's name is `andThen`. It creates a new `Future` with the specified callback, a `Future` that will have the same result as the `Future` it's called on, which allows for ordering like in the following sample:

```
val result = Future { loadPage(url) } andThen {
  case Left(exception) ⇒ log(exception)
} andThen {
  case _ ⇒ watchSomeTV
}
```

4.6.8 Auxiliary Methods

`Future fallbackTo` combines 2 Futures into a new `Future`, and will hold the successful value of the second `Future` if the first `Future` fails.

```
val future4 = future1 fallbackTo future2 fallbackTo future3
```

You can also combine two Futures into a new `Future` that will hold a tuple of the two Futures successful results, using the `zip` operation.

```
val future3 = future1 zip future2 map { case (a, b) ⇒ a + " " + b }
```

4.6.9 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 `Await.result` 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 = akka.pattern.ask(actor, msg1) recover {
  case e: ArithmeticException ⇒ 0
}
```

In this example, if the actor replied with a `akka.actor.Status.Failure` containing the `ArithmeticException`, our `Future` would have a result of 0. The `recover` 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 `recover` method.

You can also use the `recoverWith` method, which has the same relationship to `recover` as `flatMap` has to `map`, and is use like this:

```
val future = akka.pattern.ask(actor, msg1) recoverWith {
  case e: ArithmeticException        ⇒ Promise.successful(0)
  case foo: IllegalArgumentException ⇒ Promise.failed[Int](new IllegalStateException("All br0ken!
}
```

4.7 Dataflow Concurrency (Scala)

4.7.1 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 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

4.7.2 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:

```
autoCompilerPlugins := true,
libraryDependencies <+= scalaVersion { v => compilerPlugin("org.scala-lang.plugins" % "continuati
scalacOptions += "-P:continuations:enable",
```

4.7.3 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.

4.7.4 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
implicit val dispatcher = ...

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

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

val result = Await.result(c, timeout)
```

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
implicit val dispatcher = ...

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

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

val result = Await.result(c, timeout)
```

4.7.5 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). 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
implicit val dispatcher = ...

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
implicit val dispatcher = ...

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
implicit val dispatcher = ...

// 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())
}
```

4.8 Fault Tolerance (Scala)

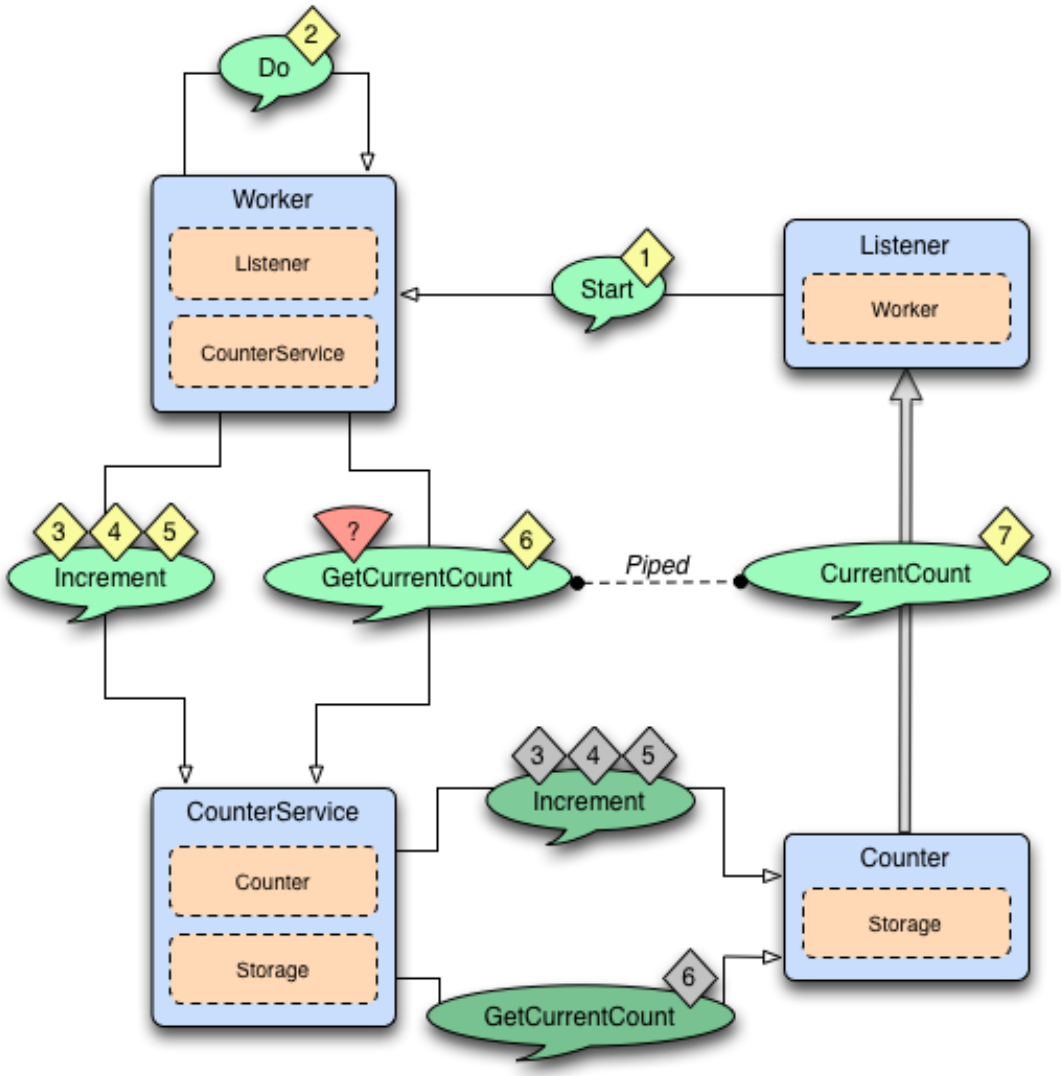
As explained in *Actor Systems* each actor is the supervisor of its children, and as such each actor defines fault handling supervisor strategy. This strategy cannot be changed afterwards as it is an integral part of the actor system's structure.

4.8.1 Fault Handling in Practice

First, let us look at a sample that illustrates one way to handle data store errors, which is a typical source of failure in real world applications. Of course it depends on the actual application what is possible to do when the data store is unavailable, but in this sample we use a best effort re-connect approach.

Read the following source code. The inlined comments explain the different pieces of the fault handling and why they are added. It is also highly recommended to run this sample as it is easy to follow the log output to understand what is happening in runtime.

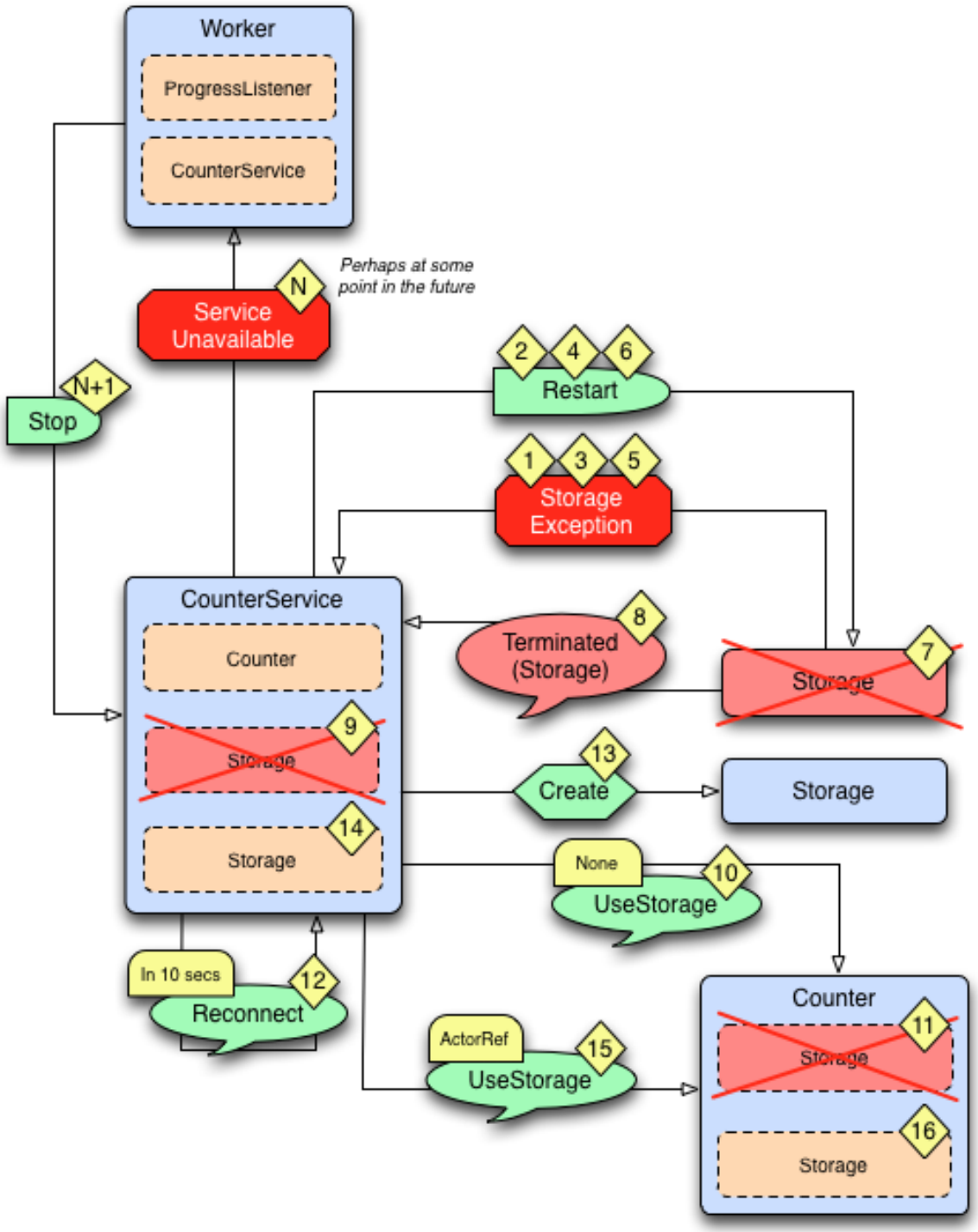
Diagrams of the Fault Tolerance Sample (Scala)



The above diagram illustrates the normal message flow.

Normal flow:

Step	Description
1	The progress `Listener` starts the work.
2	The `Worker` schedules work by sending `Do` messages periodically to itself
3, 4, 5	When receiving `Do` the `Worker` tells the `CounterService` to increment the counter, three times. The `Increment` message is forwarded to the `Counter`, which updates its counter variable and sends current value to the `Storage`.
6, 7	The `Worker` asks the `CounterService` of current value of the counter and pipes the result back to the `Listener`.



The above diagram illustrates what happens in case of storage failure.

Failure flow:

Step	Description
1	The `Storage` throws `StorageException`.
2	The `CounterService` is supervisor of the `Storage` and restarts the `Storage` when `StorageException` is thrown.
3, 4, 5, 6	The `Storage` continues to fail and is restarted.
7	After 3 failures and restarts within 5 seconds the `Storage` is stopped by its supervisor, i.e. the `CounterService`.
8	The `CounterService` is also watching the `Storage` for termination and receives the `Terminated` message when the `Storage` has been stopped ...
9, 10, 11	and tells the `Counter` that there is no `Storage`.
12	The `CounterService` schedules a `Reconnect` message to itself.
13, 14	When it receives the `Reconnect` message it creates a new `Storage` ...
15, 16	and tells the the `Counter` to use the new `Storage`

Full Source Code of the Fault Tolerance Sample (Scala)

```
// imports ...

/**
 * Runs the sample
 */
object FaultHandlingDocSample extends App {
  import Worker._

  val config = ConfigFactory.parseString("""
    akka.loglevel = DEBUG
    akka.actor.debug {
      receive = on
      lifecycle = on
    }
    """)

  val system = ActorSystem("FaultToleranceSample", config)
  val worker = system.actorOf(Props[Worker], name = "worker")
  val listener = system.actorOf(Props[Listener], name = "listener")
  // start the work and listen on progress
  // note that the listener is used as sender of the tell,
  // i.e. it will receive replies from the worker
  worker.tell(Start, sender = listener)
}

/**
 * Listens on progress from the worker and shuts down the system when enough
 * work has been done.
 */
class Listener extends Actor with ActorLogging {
  import Worker._
  // If we don't get any progress within 15 seconds then the service is unavailable
  context.setReceiveTimeout(15 seconds)

  def receive = {
    case Progress(percent) ⇒
      log.info("Current progress: {} %", percent)
      if (percent >= 100.0) {
        log.info("That's all, shutting down")
        context.system.shutdown()
```

```
      }

    case ReceiveTimeout ⇒
      // No progress within 15 seconds, ServiceUnavailable
      log.error("Shutting down due to unavailable service")
      context.system.shutdown()
  }
}

// messages ...

/**
 * Worker performs some work when it receives the `Start` message.
 * It will continuously notify the sender of the `Start` message
 * of current ``Progress``. The `Worker` supervise the `CounterService`.
 */
class Worker extends Actor with ActorLogging {
  import Worker._
  import CounterService._
  implicit val askTimeout = Timeout(5 seconds)

  // Stop the CounterService child if it throws ServiceUnavailable
  override val supervisorStrategy = OneForOneStrategy() {
    case _: CounterService.ServiceUnavailable ⇒ Stop
  }

  // The sender of the initial Start message will continuously be notified about progress
  var progressListener: Option[ActorRef] = None
  val counterService = context.actorOf(Props[CounterService], name = "counter")
  val totalCount = 51

  def receive = LoggingReceive {
    case Start if progressListener.isEmpty ⇒
      progressListener = Some(sender)
      context.system.scheduler.schedule(Duration.Zero, 1 second, self, Do)

    case Do ⇒
      counterService ! Increment(1)
      counterService ! Increment(1)
      counterService ! Increment(1)

      // Send current progress to the initial sender
      counterService ? GetCurrentCount map {
        case CurrentCount(_, count) ⇒ Progress(100.0 * count / totalCount)
      } pipeTo progressListener.get
  }
}

// messages ...

/**
 * Adds the value received in `Increment` message to a persistent
 * counter. Replies with `CurrentCount` when it is asked for `CurrentCount`.
 * `CounterService` supervise `Storage` and `Counter`.
 */
class CounterService extends Actor {
  import CounterService._
  import Counter._
  import Storage._

  // Restart the storage child when StorageException is thrown.
  // After 3 restarts within 5 seconds it will be stopped.
  override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 3, withinTimeRange = 5 seco
```

```
    case _: Storage.StorageException ⇒ Restart
  }

  val key = self.path.name
  var storage: Option[ActorRef] = None
  var counter: Option[ActorRef] = None
  var backlog = IndexedSeq.empty[(ActorRef, Any)]
  val MaxBacklog = 10000

  override def preStart() {
    initStorage()
  }

  /**
   * The child storage is restarted in case of failure, but after 3 restarts,
   * and still failing it will be stopped. Better to back-off than continuously
   * failing. When it has been stopped we will schedule a Reconnect after a delay.
   * Watch the child so we receive Terminated message when it has been terminated.
   */
  def initStorage() {
    storage = Some(context.watch(context.actorOf(Props[Storage], name = "storage")))
    // Tell the counter, if any, to use the new storage
    counter foreach { _ ! UseStorage(storage) }
    // We need the initial value to be able to operate
    storage.get ! Get(key)
  }

  def receive = LoggingReceive {

    case Entry(k, v) if k == key && counter == None ⇒
      // Reply from Storage of the initial value, now we can create the Counter
      val c = context.actorOf(Props(new Counter(key, v)))
      counter = Some(c)
      // Tell the counter to use current storage
      c ! UseStorage(storage)
      // and send the buffered backlog to the counter
      for ((replyTo, msg) ← backlog) c.tell(msg, sender = replyTo)
      backlog = IndexedSeq.empty

    case msg @ Increment(n)    ⇒ forwardOrPlaceInBacklog(msg)

    case msg @ GetCurrentCount ⇒ forwardOrPlaceInBacklog(msg)

    case Terminated(actorRef) if Some(actorRef) == storage ⇒
      // After 3 restarts the storage child is stopped.
      // We receive Terminated because we watch the child, see initStorage.
      storage = None
      // Tell the counter that there is no storage for the moment
      counter foreach { _ ! UseStorage(None) }
      // Try to re-establish storage after while
      context.system.scheduler.scheduleOnce(10 seconds, self, Reconnect)

    case Reconnect ⇒
      // Re-establish storage after the scheduled delay
      initStorage()
  }

  def forwardOrPlaceInBacklog(msg: Any) {
    // We need the initial value from storage before we can start delegate to the counter.
    // Before that we place the messages in a backlog, to be sent to the counter when
    // it is initialized.
    counter match {
      case Some(c) ⇒ c forward msg
```

```
      case None ⇒
        if (backlog.size >= MaxBacklog)
          throw new ServiceUnavailable("CounterService not available, lack of initial value")
        backlog = backlog :+ (sender, msg)
    }
  }

}

// messages ...

/**
 * The in memory count variable that will send current
 * value to the `Storage`, if there is any storage
 * available at the moment.
 */
class Counter(key: String, initialValue: Long) extends Actor {
  import Counter._
  import CounterService._
  import Storage._

  var count = initialValue
  var storage: Option[ActorRef] = None

  def receive = LoggingReceive {
    case UseStorage(s) ⇒
      storage = s
      storeCount()

    case Increment(n) ⇒
      count += n
      storeCount()

    case GetCurrentCount ⇒
      sender ! CurrentCount(key, count)

  }

  def storeCount() {
    // Delegate dangerous work, to protect our valuable state.
    // We can continue without storage.
    storage foreach { _ ! Store(Entry(key, count)) }
  }

}

// messages ...

/**
 * Saves key/value pairs to persistent storage when receiving `Store` message.
 * Replies with current value when receiving `Get` message.
 * Will throw StorageException if the underlying data store is out of order.
 */
class Storage extends Actor {
  import Storage._

  val db = DummyDB

  def receive = LoggingReceive {
    case Store(Entry(key, count)) ⇒ db.save(key, count)
    case Get(key)                 ⇒ sender ! Entry(key, db.load(key).getOrElse(0L))
  }
}
```

```
// dummydb ...
```

4.8.2 Creating a Supervisor Strategy

The following sections explain the fault handling mechanism and alternatives in more depth.

For the sake of demonstration let us consider the following strategy:

```
import akka.actor.OneForOneStrategy
import akka.actor.SupervisorStrategy._
import akka.util.duration._

override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minut
  case _: ArithmeticException      ⇒ Resume
  case _: NullPointerException     ⇒ Restart
  case _: IllegalArgumentException ⇒ Stop
  case _: Exception                ⇒ Escalate
}
```

I have chosen a few well-known exception types in order to demonstrate the application of the fault handling directives described in *Supervision and Monitoring*. First off, it is a one-for-one strategy, meaning that each child is treated separately (an all-for-one strategy works very similarly, the only difference is that any decision is applied to all children of the supervisor, not only the failing one). There are limits set on the restart frequency, namely maximum 10 restarts per minute; each of these settings could be left out, which means that the respective limit does not apply, leaving the possibility to specify an absolute upper limit on the restarts or to make the restarts work infinitely.

The match statement which forms the bulk of the body is of type `Decider`, which is a `PartialFunction[Throwable, Directive]`. This is the piece which maps child failure types to their corresponding directives.

4.8.3 Default Supervisor Strategy

`Escalate` is used if the defined strategy doesn't cover the exception that was thrown.

When the supervisor strategy is not defined for an actor the following exceptions are handled by default:

- `ActorInitializationException` will stop the failing child actor
- `ActorKilledException` will stop the failing child actor
- `Exception` will restart the failing child actor
- Other types of `Throwable` will be escalated to parent actor

If the exception escalate all the way up to the root guardian it will handle it in the same way as the default strategy defined above.

4.8.4 Test Application

The following section shows the effects of the different directives in practice, wherefor a test setup is needed. First off, we need a suitable supervisor:

```
import akka.actor.Actor

class Supervisor extends Actor {
  import akka.actor.OneForOneStrategy
  import akka.actor.SupervisorStrategy._
  import akka.util.duration._

  override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 mi
```

```
    case _: ArithmeticException      ⇒ Resume
    case _: NullPointerException     ⇒ Restart
    case _: IllegalArgumentException ⇒ Stop
    case _: Exception                ⇒ Escalate
  }

  def receive = {
    case p: Props ⇒ sender ! context.actorOf(p)
  }
}
```

This supervisor will be used to create a child, with which we can experiment:

```
import akka.actor.Actor

class Child extends Actor {
  var state = 0
  def receive = {
    case ex: Exception ⇒ throw ex
    case x: Int        ⇒ state = x
    case "get"         ⇒ sender ! state
  }
}
```

The test is easier by using the utilities described in *Testing Actor Systems (Scala)*, where `AkkaSpec` is a convenient mixture of `TestKit with WordSpec with MustMatchers`

```
import akka.testkit.{ AkkaSpec, ImplicitSender, EventFilter }
import akka.actor.{ ActorRef, Props, Terminated }

class FaultHandlingDocSpec extends AkkaSpec with ImplicitSender {

  "A supervisor" must {

    "apply the chosen strategy for its child" in {
      // code here
    }
  }
}
```

Let us create actors:

```
val supervisor = system.actorOf(Props[Supervisor], "supervisor")

supervisor ! Props[Child]
val child = expectMsgType[ActorRef] // retrieve answer from TestKit's testActor
```

The first test shall demonstrate the `Resume` directive, so we try it out by setting some non-initial state in the actor and have it fail:

```
child ! 42 // set state to 42
child ! "get"
expectMsg(42)

child ! new ArithmeticException // crash it
child ! "get"
expectMsg(42)
```

As you can see the value 42 survives the fault handling directive. Now, if we change the failure to a more serious `NullPointerException`, that will no longer be the case:

```
child ! new NullPointerException // crash it harder
child ! "get"
expectMsg(0)
```

And finally in case of the fatal `IllegalArgumentException` the child will be terminated by the supervisor:

```
watch(child) // have testActor watch "child"
child ! new IllegalArgumentException // break it
expectMsg(Terminated(child))
child.isTerminated must be(true)
```

Up to now the supervisor was completely unaffected by the child's failure, because the directives set did handle it. In case of an `Exception`, this is not true anymore and the supervisor escalates the failure.

```
supervisor ! Props[Child] // create new child
val child2 = expectMsgType[ActorRef]

watch(child2)
child2 ! "get" // verify it is alive
expectMsg(0)

child2 ! new Exception("CRASH") // escalate failure
expectMsg(Terminated(child2))
```

The supervisor itself is supervised by the top-level actor provided by the `ActorSystem`, which has the default policy to restart in case of all `Exception` cases (with the notable exceptions of `ActorInitializationException` and `ActorKilledException`). Since the default directive in case of a restart is to kill all children, we expected our poor child not to survive this failure.

In case this is not desired (which depends on the use case), we need to use a different supervisor which overrides this behavior.

```
class Supervisor2 extends Actor {
  import akka.actor.OneForOneStrategy
  import akka.actor.SupervisorStrategy._
  import akka.util.duration._

  override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 mi
    case _: ArithmeticException      ⇒ Resume
    case _: NullPointerException     ⇒ Restart
    case _: IllegalArgumentException ⇒ Stop
    case _: Exception                ⇒ Escalate
  }

  def receive = {
    case p: Props ⇒ sender ! context.actorOf(p)
  }
  // override default to kill all children during restart
  override def preRestart(cause: Throwable, msg: Option[Any]) {}
}
```

With this parent, the child survives the escalated restart, as demonstrated in the last test:

```
val supervisor2 = system.actorOf(Props[Supervisor2], "supervisor2")

supervisor2 ! Props[Child]
val child3 = expectMsgType[ActorRef]

child3 ! 23
child3 ! "get"
expectMsg(23)

child3 ! new Exception("CRASH")
child3 ! "get"
expectMsg(0)
```

4.9 Dispatchers (Scala)

An Akka `MessageDispatcher` is what makes Akka Actors "tick", it is the engine of the machine so to speak. All `MessageDispatcher` implementations are also an `ExecutionContext`, which means that they can be used to execute arbitrary code, for instance *Futures (Scala)*.

4.9.1 Default dispatcher

Every `ActorSystem` will have a default dispatcher that will be used in case nothing else is configured for an `Actor`. The default dispatcher can be configured, and is by default a `Dispatcher` with a "fork-join-executor", which gives excellent performance in most cases.

4.9.2 Setting the dispatcher for an Actor

So in case you want to give your `Actor` a different dispatcher than the default, you need to do two things, of which the first is:

```
import akka.actor.Props
val myActor =
  context.actorOf(Props[MyActor].withDispatcher("my-dispatcher"), "myactor1")
```

Note: The "dispatcherId" you specify in withDispatcher is in fact a path into your configuration. So in this example it's a top-level section, but you could for instance put it as a sub-section, where you'd use periods to denote sub-sections, like this: `"foo.bar.my-dispatcher"`

And then you just need to configure that dispatcher in your configuration:

```
my-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "fork-join-executor"
  # Configuration for the fork join pool
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 2
    # Parallelism (threads) ... ceil(available processors * factor)
    parallelism-factor = 2.0
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

And here's another example that uses the "thread-pool-executor":

```
my-thread-pool-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "thread-pool-executor"
  # Configuration for the thread pool
  thread-pool-executor {
    # minimum number of threads to cap factor-based core number to
    core-pool-size-min = 2
```

```
    # No of core threads ... ceil(available processors * factor)
    core-pool-size-factor = 2.0
    # maximum number of threads to cap factor-based number to
    core-pool-size-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

For more options, see the default-dispatcher section of the *Configuration*.

4.9.3 Types of dispatchers

There are 4 different types of message dispatchers:

- Dispatcher
  - This is an event-based dispatcher that binds a set of Actors to a thread pool. It is the default dispatcher used if one is not specified.
  - Sharability: Unlimited
  - Mailboxes: Any, creates one per Actor
  - Use cases: Default dispatcher, Bulkheading
  - **Driven by:** `java.util.concurrent.ExecutorService` specify using "executor" using "fork-join-executor", "thread-pool-executor" or the FQCN of an `akka.dispatcher.ExecutorServiceConfigurator`
- PinnedDispatcher
  - This dispatcher dedicates a unique thread for each actor using it; i.e. each actor will have its own thread pool with only one thread in the pool.
  - Sharability: None
  - Mailboxes: Any, creates one per Actor
  - Use cases: Bulkheading
  - **Driven by:** Any `akka.dispatch.ThreadPoolExecutorConfigurator` by default a "thread-pool-executor"
- BalancingDispatcher
  - This is an executor based event driven dispatcher that will try to redistribute work from busy actors to idle actors.
  - All the actors share a single Mailbox that they get their messages from.
  - It is assumed that all actors using the same instance of this dispatcher can process all messages that have been sent to one of the actors; i.e. the actors belong to a pool of actors, and to the client there is no guarantee about which actor instance actually processes a given message.
  - Sharability: Actors of the same type only
  - Mailboxes: Any, creates one for all Actors
  - Use cases: Work-sharing
  - **Driven by:** `java.util.concurrent.ExecutorService` specify using "executor" using "fork-join-executor", "thread-pool-executor" or the FQCN of an `akka.dispatcher.ExecutorServiceConfigurator`

  - Note that you can **not** use a `BalancingDispatcher` as a **Router Dispatcher**. (You can however use it for the **Routees**)

- CallingThreadDispatcher
  - This dispatcher runs invocations on the current thread only. This dispatcher does not create any new threads, but it can be used from different threads concurrently for the same actor. See *CallingThreadDispatcher* for details and restrictions.
  - Sharability: Unlimited
  - Mailboxes: Any, creates one per Actor per Thread (on demand)
  - Use cases: Testing
  - Driven by: The calling thread (duh)

More dispatcher configuration examples

Configuring a `PinnedDispatcher`:

```
my-pinned-dispatcher {
  executor = "thread-pool-executor"
  type = PinnedDispatcher
}
```

And then using it:

```
val myActor =
  context.actorOf(Props[MyActor].withDispatcher("my-pinned-dispatcher"), "myactor2")
```

Note that `thread-pool-executor` configuration as per the above `my-thread-pool-dispatcher` example is NOT applicable. This is because every actor will have its own thread pool when using `PinnedDispatcher`,

4.9.4 Mailboxes

An Akka `Mailbox` holds the messages that are destined for an `Actor`. Normally each `Actor` has its own mailbox, but with example a `BalancingDispatcher` all actors with the same `BalancingDispatcher` will share a single instance.

Builtin implementations

Akka comes shipped with a number of default mailbox implementations:

- UnboundedMailbox
  - Backed by a `java.util.concurrent.ConcurrentLinkedQueue`
  - Blocking: No
  - Bounded: No
- BoundedMailbox
  - Backed by a `java.util.concurrent.LinkedBlockingQueue`
  - Blocking: Yes
  - Bounded: Yes
- UnboundedPriorityMailbox
  - Backed by a `java.util.concurrent.PriorityBlockingQueue`
  - Blocking: Yes

  - Bounded: No
- BoundedPriorityMailbox
  - Backed by a `java.util.PriorityBlockingQueue` wrapped in an `akka.util.BoundedBlockingQueue`
  - Blocking: Yes
  - Bounded: Yes
- Durable mailboxes, see *Durable Mailboxes*.

Mailbox configuration examples

How to create a PriorityMailbox:

```
import akka.dispatch.PriorityGenerator
import akka.dispatch.UnboundedPriorityMailbox
import com.typesafe.config.Config

// We inherit, in this case, from UnboundedPriorityMailbox
// and seed it with the priority generator
class MyPrioMailbox(settings: ActorSystem.Settings, config: Config) extends UnboundedPriorityMailb
  // Create a new PriorityGenerator, lower prio means more important
  PriorityGenerator {
    // 'highpriority messages should be treated first if possible
    case 'highpriority ⇒ 0

    // 'lowpriority messages should be treated last if possible
    case 'lowpriority  ⇒ 2

    // PoisonPill when no other left
    case PoisonPill    ⇒ 3

    // We default to 1, which is in between high and low
    case otherwise     ⇒ 1
  })
```

And then add it to the configuration:

```
prio-dispatcher {
  mailbox-type = "akka.docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
}
```

And then an example on how you would use it:

```
// We create a new Actor that just prints out what it processes
val a = system.actorOf(
  Props(new Actor {
    val log: LoggingAdapter = Logging(context.system, this)

    self ! 'lowpriority
    self ! 'lowpriority
    self ! 'highpriority
    self ! 'pigdog
    self ! 'pigdog2
    self ! 'pigdog3
    self ! 'highpriority
    self ! PoisonPill

    def receive = {
      case x ⇒ log.info(x.toString)
    }
  }).withDispatcher("prio-dispatcher"))
```

```
/*
Logs:
  'highpriority
  'highpriority
  'pigdog
  'pigdog2
  'pigdog3
  'lowpriority
  'lowpriority
*/
```

Creating your own Mailbox type

An example is worth a thousand quacks:

```
case class MyUnboundedMailbox() extends akka.dispatch.MailboxType {
  import akka.actor.ActorContext
  import com.typesafe.config.Config
  import java.util.concurrent.ConcurrentLinkedQueue
  import akka.dispatch.{
    Envelope,
    MessageQueue,
    QueueBasedMessageQueue,
    UnboundedMessageQueueSemantics
  }

  // This constructor signature must exist, it will be called by Akka
  def this(settings: ActorSystem.Settings, config: Config) = this()

  // The create method is called to create the MessageQueue
  final override def create(owner: Option[ActorContext]): MessageQueue =
    new QueueBasedMessageQueue with UnboundedMessageQueueSemantics {
      final val queue = new ConcurrentLinkedQueue[Envelope]()
    }
```

And then you just specify the FQCN of your MailboxType as the value of the "mailbox-type" in the dispatcher configuration.

Note: Make sure to include a constructor which takes `akka.actor.ActorSystem.Settings` and `com.typesafe.config.Config` arguments, as this constructor is invoked reflectively to construct your mailbox type. The config passed in as second argument is that section from the configuration which describes the dispatcher using this mailbox type; the mailbox type will be instantiated once for each dispatcher using it.

4.10 Routing (Scala)

A Router is an actor that routes incoming messages to outbound actors. The router routes the messages sent to it to its underlying actors called 'routees'.

Akka comes with some defined routers out of the box, but as you will see in this chapter it is really easy to create your own. The routers shipped with Akka are:

- `akka.routing.RoundRobinRouter`
- `akka.routing.RandomRouter`
- `akka.routing.SmallestMailboxRouter`
- `akka.routing.BroadcastRouter`

- `akka.routing.ScatterGatherFirstCompletedRouter`

4.10.1 Routers In Action

This is an example of how to create a router that is defined in configuration:

```
akka.actor.deployment {
  /router {
    router = round-robin
    nr-of-instances = 5
  }
}
```

```
val router = system.actorOf(Props[ExampleActor].withRouter(FromConfig()),
  "router")
```

This is an example of how to programmatically create a router and set the number of routees it should create:

```
val router1 = system.actorOf(Props[ExampleActor1].withRouter(
  RoundRobinRouter(nrOfInstances = 5)))
```

You can also give the router already created routees as in:

```
val actor1 = system.actorOf(Props[ExampleActor1])
val actor2 = system.actorOf(Props[ExampleActor1])
val actor3 = system.actorOf(Props[ExampleActor1])
val routees = Vector[ActorRef](actor1, actor2, actor3)
val router2 = system.actorOf(Props[ExampleActor1].withRouter(
  RoundRobinRouter(routees = routees)))
```

When you create a router programmatically you define the number of routees *or* you pass already created routees to it. If you send both parameters to the router *only* the latter will be used, i.e. `nrOfInstances` is disregarded.

It is also worth pointing out that if you define the ''router'' in the configuration file then this value will be used instead of any programmatically sent parameters. The decision whether to create a router at all, on the other hand, must be taken within the code, i.e. you cannot make something a router by external configuration alone (see below for details).

Once you have the router actor it is just to send messages to it as you would to any actor:

```
router ! MyMsg
```

The router will apply its behavior to the message it receives and forward it to the routees.

Remotely Deploying Routees

In addition to being able to supply looked-up remote actors as routees, you can make the router deploy its created children on a set of remote hosts; this will be done in round-robin fashion. In order to do that, wrap the router configuration in a `RemoteRouterConfig`, attaching the remote addresses of the nodes to deploy to. Naturally, this requires you to include the `akka-remote` module on your classpath:

```
import akka.actor.{ Address, AddressFromURIString }
val addresses = Seq(
  Address("akka", "remotesys", "otherhost", 1234),
  AddressFromURIString("akka://othersys@anotherhost:1234"))
val routerRemote = system.actorOf(Props[ExampleActor1].withRouter(
  RemoteRouterConfig(RoundRobinRouter(5), addresses)))
```

4.10.2 How Routing is Designed within Akka

Routers behave like single actors, but they should also not hinder scalability. This apparent contradiction is solved by making routers be represented by a special `RoutedActorRef`, which dispatches incoming messages destined for the routees without actually invoking the router actor's behavior (and thus avoiding its mailbox; the single router actor's task is to manage all aspects related to the lifecycle of the routees). This means that the code which decides which route to take is invoked concurrently from all possible senders and hence must be thread-safe, it cannot live the simple and happy life of code within an actor.

There is one part in the above paragraph which warrants some more background explanation: Why does a router need a "head" which is actual parent to all the routees? The initial design tried to side-step this issue, but location transparency as well as mandatory parental supervision required a redesign. Each of the actors which the router spawns must have its unique identity, which translates into a unique actor path. Since the router has only one given name in its parent's context, another level in the name space is needed, which according to the addressing semantics implies the existence of an actor with the router's name. This is not only necessary for the internal messaging involved in creating, restarting and terminating actors, it is also needed when the pooled actors need to converse with other actors and receive replies in a deterministic fashion. Since each actor knows its own external representation as well as that of its parent, the routees decide where replies should be sent when reacting to a message:

```
sender.tell("reply", context.parent) // replies will go back to parent
sender.!("reply")(context.parent) // alternative syntax (beware of the parens!)
```

```
sender ! x // replies will go to this actor
```

It is apparent now why routing needs to be enabled in code rather than being possible to "bolt on" later: whether or not an actor is routed means a change to the actor hierarchy, changing the actor paths of all children of the router. The routees especially do need to know that they are routed to in order to choose the sender reference for any messages they dispatch as shown above.

4.10.3 Routers vs. Supervision

As explained in the previous section, routers create new actor instances as children of the "head" router, who therefore also is their supervisor. The supervisor strategy of this actor can be configured by means of the `RouterConfig.supervisorStrategy` property, which is supported for all built-in router types. It defaults to "always escalate", which leads to the application of the router's parent's supervision directive to all children of the router uniformly (i.e. not only the one which failed). It should be mentioned that the router overrides the default behavior of terminating all children upon restart, which means that a restart—while re-creating them—does not have an effect on the number of actors in the pool.

Setting the strategy is easily done:

```
val escalator = OneForOneStrategy() {
  // custom strategy ...
}
val router = system.actorOf(Props.empty.withRouter(
  RoundRobinRouter(1, supervisorStrategy = escalator)))
```

Another potentially useful approach is to give the router the same strategy as its parent, which effectively treats all actors in the pool as if they were direct children of their grand-parent instead.

4.10.4 Router usage

In this section we will describe how to use the different router types. First we need to create some actors that will be used in the examples:

```
class PrintlnActor extends Actor {
  def receive = {
    case msg ⇒
      println("Received message '%s' in actor %s".format(msg, self.path.name))
```

```
  }
}
```

and

```
class FibonacciActor extends Actor {
  def receive = {
    case FibonacciNumber(nbr) ⇒ sender tell fibonacci(nbr)
  }

  private def fibonacci(n: Int): Int = {
    @tailrec
    def fib(n: Int, b: Int, a: Int): Int = n match {
      case 0 ⇒ a
      case _ ⇒ fib(n - 1, a + b, b)
    }

    fib(n, 1, 0)
  }
}
```

RoundRobinRouter

Routes in a round-robin fashion to its routees. Code example:

```
val roundRobinRouter =
  context.actorOf(Props[PrintlnActor].withRouter(RoundRobinRouter(5)), "router")
1 to 10 foreach {
  i ⇒ roundRobinRouter ! i
}
```

When run you should see a similar output to this:

```
Received message '1' in actor $b
Received message '2' in actor $c
Received message '3' in actor $d
Received message '6' in actor $b
Received message '4' in actor $e
Received message '8' in actor $d
Received message '5' in actor $f
Received message '9' in actor $e
Received message '10' in actor $f
Received message '7' in actor $c
```

If you look closely to the output you can see that each of the routees received two messages which is exactly what you would expect from a round-robin router to happen. (The name of an actor is automatically created in the format `$letter` unless you specify it - hence the names printed above.)

RandomRouter

As the name implies this router type selects one of its routees randomly and forwards the message it receives to this routee. This procedure will happen each time it receives a message. Code example:

```
val randomRouter =
  context.actorOf(Props[PrintlnActor].withRouter(RandomRouter(5)), "router")
1 to 10 foreach {
  i ⇒ randomRouter ! i
}
```

When run you should see a similar output to this:

```
Received message '1' in actor $e
Received message '2' in actor $c
Received message '4' in actor $b
Received message '5' in actor $d
Received message '3' in actor $e
Received message '6' in actor $c
Received message '7' in actor $d
Received message '8' in actor $e
Received message '9' in actor $d
Received message '10' in actor $d
```

The result from running the random router should be different, or at least random, every time you run it. Try to run it a couple of times to verify its behavior if you don't trust us.

SmallestMailboxRouter

A Router that tries to send to the non-suspended routee with fewest messages in mailbox. The selection is done in this order:

- pick any idle routee (not processing message) with empty mailbox
- pick any routee with empty mailbox
- pick routee with fewest pending messages in mailbox
- pick any remote routee, remote actors are consider lowest priority, since their mailbox size is unknown

Code example:

```
val smallestMailboxRouter =
  context.actorOf(Props[PrintlnActor].withRouter(SmallestMailboxRouter(5)), "router")
1 to 10 foreach {
  i ⇒ smallestMailboxRouter ! i
}
```

BroadcastRouter

A broadcast router forwards the message it receives to *all* its routees. Code example:

```
val broadcastRouter =
  context.actorOf(Props[PrintlnActor].withRouter(BroadcastRouter(5)), "router")
broadcastRouter ! "this is a broadcast message"
```

When run you should see a similar output to this:

```
Received message 'this is a broadcast message' in actor $f
Received message 'this is a broadcast message' in actor $d
Received message 'this is a broadcast message' in actor $e
Received message 'this is a broadcast message' in actor $c
Received message 'this is a broadcast message' in actor $b
```

As you can see here above each of the routees, five in total, received the broadcast message.

ScatterGatherFirstCompletedRouter

The ScatterGatherFirstCompletedRouter will send the message on to all its routees as a future. It then waits for first result it gets back. This result will be sent back to original sender. Code example:

```
val scatterGatherFirstCompletedRouter = context.actorOf(
  Props[FibonacciActor].withRouter(ScatterGatherFirstCompletedRouter(
    nrOfInstances = 5, within = 2 seconds)), "router")
implicit val timeout = Timeout(5 seconds)
```

```
val futureResult = scatterGatherFirstCompletedRouter ? FibonacciNumber(10)
val result = Await.result(futureResult, timeout.duration)
```

When run you should see this:

```
The result of calculating Fibonacci for 10 is 55
```

From the output above you can't really see that all the routees performed the calculation, but they did! The result you see is from the first routee that returned its calculation to the router.

4.10.5 Broadcast Messages

There is a special type of message that will be sent to all routees regardless of the router. This message is called `Broadcast` and is used in the following manner:

```
router ! Broadcast("Watch out for Davy Jones' locker")
```

Only the actual message is forwarded to the routees, i.e. "Watch out for Davy Jones' locker" in the example above. It is up to the routee implementation whether to handle the broadcast message or not.

4.10.6 Dynamically Resizable Routers

All routers can be used with a fixed number of routees or with a resize strategy to adjust the number of routees dynamically.

This is an example of how to create a resizable router that is defined in configuration:

```
akka.actor.deployment {
  /router2 {
    router = round-robin
    resizer {
      lower-bound = 2
      upper-bound = 15
    }
  }
}
```

```
val router2 = system.actorOf(Props[ExampleActor].withRouter(FromConfig()),
  "router2")
```

Several more configuration options are available and described in `akka.actor.deployment.default.resizer` section of the reference *Configuration*.

This is an example of how to programmatically create a resizable router:

```
val resizer = DefaultResizer(lowerBound = 2, upperBound = 15)
val router3 = system.actorOf(Props[ExampleActor1].withRouter(
  RoundRobinRouter(resizer = Some(resizer))))
```

It is also worth pointing out that if you define the ''router'' in the configuration file then this value will be used instead of any programmatically sent parameters.

Note: Resizing is triggered by sending messages to the actor pool, but it is not completed synchronously; instead a message is sent to the "head" `Router` to perform the size change. Thus you cannot rely on resizing to instantaneously create new workers when all others are busy, because the message just sent will be queued to the mailbox of a busy actor. To remedy this, configure the pool to use a balancing dispatcher, see Configuring Dispatchers for more information.

4.10.7 Custom Router

You can also create your own router should you not find any of the ones provided by Akka sufficient for your needs. In order to roll your own router you have to fulfill certain criteria which are explained in this section.

The router created in this example is a simple vote counter. It will route the votes to specific vote counter actors. In this case we only have two parties the Republicans and the Democrats. We would like a router that forwards all democrat related messages to the Democrat actor and all republican related messages to the Republican actor.

We begin with defining the class:

```
case class VoteCountRouter() extends RouterConfig {

  def routerDispatcher: String = Dispatchers.DefaultDispatcherId
  def supervisorStrategy: SupervisorStrategy = SupervisorStrategy.defaultStrategy

  // crRoute ...

}
```

The next step is to implement the `createRoute` method in the class just defined:

```
def createRoute(routeeProps: Props, routeeProvider: RouteeProvider): Route = {
  val democratActor = routeeProvider.context.actorOf(Props(new DemocratActor()), "d")
  val republicanActor = routeeProvider.context.actorOf(Props(new RepublicanActor()), "r")
  val routees = Vector[ActorRef](democratActor, republicanActor)

  routeeProvider.registerRoutees(routees)

  {
    case (sender, message) ⇒
      message match {
        case DemocratVote | DemocratCountResult ⇒
          List(Destination(sender, democratActor))
        case RepublicanVote | RepublicanCountResult ⇒
          List(Destination(sender, republicanActor))
      }
  }
}
```

As you can see above we start off by creating the routees and put them in a collection.

Make sure that you don't miss to implement the line below as it is *really* important. It registers the routees internally and failing to call this method will cause a `ActorInitializationException` to be thrown when the router is used. Therefore always make sure to do the following in your custom router:

```
routeeProvider.registerRoutees(routees)
```

The routing logic is where your magic sauce is applied. In our example it inspects the message types and forwards to the correct routee based on this:

```
{
  case (sender, message) ⇒
    message match {
      case DemocratVote | DemocratCountResult ⇒
        List(Destination(sender, democratActor))
      case RepublicanVote | RepublicanCountResult ⇒
        List(Destination(sender, republicanActor))
    }
}
```

As you can see above what's returned in the partial function is a `List` of `Destination(sender, routee)`. The sender is what "parent" the routee should see - changing this could be useful if you for example want another actor than the original sender to intermediate the result of the routee (if there is a result). For more information about how to alter the original sender we refer to the source code of ScatterGatherFirstCompletedRouter

All in all the custom router looks like this:

```
case object DemocratVote
case object DemocratCountResult
case object RepublicanVote
case object RepublicanCountResult

class DemocratActor extends Actor {
  var counter = 0

  def receive = {
    case DemocratVote        ⇒ counter += 1
    case DemocratCountResult ⇒ sender ! counter
  }
}

class RepublicanActor extends Actor {
  var counter = 0

  def receive = {
    case RepublicanVote        ⇒ counter += 1
    case RepublicanCountResult ⇒ sender ! counter
  }
}

case class VoteCountRouter() extends RouterConfig {

  def routerDispatcher: String = Dispatchers.DefaultDispatcherId
  def supervisorStrategy: SupervisorStrategy = SupervisorStrategy.defaultStrategy

  def createRoute(routeeProps: Props, routeeProvider: RouteeProvider): Route = {
    val democratActor = routeeProvider.context.actorOf(Props(new DemocratActor()), "d")
    val republicanActor = routeeProvider.context.actorOf(Props(new RepublicanActor()), "r")
    val routees = Vector[ActorRef](democratActor, republicanActor)

    routeeProvider.registerRoutees(routees)

    {
      case (sender, message) ⇒
        message match {
          case DemocratVote | DemocratCountResult ⇒
            List(Destination(sender, democratActor))
          case RepublicanVote | RepublicanCountResult ⇒
            List(Destination(sender, republicanActor))
        }
    }
  }

}
```

If you are interested in how to use the VoteCountRouter you can have a look at the test class RoutingSpec

Configured Custom Router

It is possible to define configuration properties for custom routers. In the `router` property of the deployment configuration you define the fully qualified class name of the router class. The router class must extend `akka.routing.RouterConfig` and have a constructor with one `com.typesafe.config.Config` parameter. The deployment section of the configuration is passed to the constructor.

Custom Resizer

A router with dynamically resizable number of routees is implemented by providing a `akka.routing.Resizer` in `resizer` method of the `RouterConfig`. See `akka.routing.DefaultResizer` for inspiration of how to write your own resize strategy.

4.10.8 Configuring Dispatchers

The dispatcher for created children of the router will be taken from `Props` as described in *Dispatchers (Scala)*. For a dynamic pool it makes sense to configure the `BalancingDispatcher` if the precise routing is not so important (i.e. no consistent hashing or round-robin is required); this enables newly created routees to pick up work immediately by stealing it from their siblings. Note that you can **not** use a `BalancingDispatcher` as a **Router Dispatcher**. (You can however use it for the **Routees**)

Note: If you provide a collection of actors to route to, then they will still use the same dispatcher that was configured for them in their `Props`, it is not possible to change an actors dispatcher after it has been created.

The "head" router cannot always run on the same dispatcher, because it does not process the same type of messages, hence this special actor does not use the dispatcher configured in `Props`, but takes the `routerDispatcher` from the `RouterConfig` instead, which defaults to the actor system's default dispatcher. All standard routers allow setting this property in their constructor or factory method, custom routers have to implement the method in a suitable way.

```
val router = system.actorOf(Props[MyActor]
  .withRouter(RoundRobinRouter(5, routerDispatcher = "router")) // "head" will run on "router" di
  .withDispatcher("workers")) // MyActor workers will run on "workers" dispatcher
```

Note: It is not allowed to configure the `routerDispatcher` to be a `BalancingDispatcher` since the messages meant for the special router actor cannot be processed by any other actor.

At first glance there seems to be an overlap between the `BalancingDispatcher` and Routers, but they complement each other. The balancing dispatcher is in charge of running the actors while the routers are in charge of deciding which message goes where. A router can also have children that span multiple actor systems, even remote ones, but a dispatcher lives inside a single actor system.

When using a `RoundRobinRouter` with a `BalancingDispatcher` there are some configuration settings to take into account.

- There can only be `nr-of-instances` messages being processed at the same time no matter how many threads are configured for the `BalancingDispatcher`.
- Having `throughput` set to a low number makes no sense since you will only be handing off to another actor that processes the same `MailBox` as yourself, which can be costly. Either the message just got into the mailbox and you can receive it as well as anybody else, or everybody else is busy and you are the only one available to receive the message.
- Resizing the number of routees only introduce inertia, since resizing is performed at specified intervals, but work stealing is instantaneous.

4.11 Remoting (Scala)

For an introduction of remoting capabilities of Akka please see *Location Transparency*.

4.11.1 Preparing your ActorSystem for Remoting

The Akka remoting is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" % "akka-remote" % "2.0.4"
```

To enable remote capabilities in your Akka project you should, at a minimum, add the following changes to your `application.conf` file:

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    transport = "akka.remote.netty.NettyRemoteTransport"
    netty {
      hostname = "127.0.0.1"
      port = 2552
    }
 }
}
```

As you can see in the example above there are four things you need to add to get started:

- Change provider from `akka.actor.LocalActorRefProvider` to `akka.remote.RemoteActorRefProvider`
- Add host name - the machine you want to run the actor system on; this host name is exactly what is passed to remote systems in order to identify this system and consequently used for connecting back to this system if need be, hence set it to a reachable IP address or resolvable name in case you want to communicate across the network.
- Add port number - the port the actor system should listen on, set to 0 to have it chosen automatically

---

Note: The port number needs to be unique for each actor system on the same machine even if the actor systems have different names. This is because each actor system has its own network subsystem listening for connections and handling messages as not to interfere with other actor systems.

The example above only illustrates the bare minimum of properties you have to add to enable remoting. There are lots of more properties that are related to remoting in Akka. We refer to the following reference file for more information:

```
####################################
# Akka Remote Reference Config File #
####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

# comments about akka.actor settings left out where they are already in akka-
# actor.jar, because otherwise they would be repeated in config rendering.

akka {

  actor {

    serializers {
      proto = "akka.serialization.ProtobufSerializer"
    }


    serialization-bindings {
      # Since com.google.protobuf.Message does not extend Serializable but GeneratedMessage
      # does, need to use the more specific one here in order to avoid ambiguity
      "com.google.protobuf.GeneratedMessage" = proto
    }
```

```
    deployment {

      default {

        # if this is set to a valid remote address, the named actor will be deployed
        # at that node e.g. "akka://sys@host:port"
        remote = ""

        target {

          # A list of hostnames and ports for instantiating the children of a
          # non-direct router
          #   The format should be on "akka://sys@host:port", where:
          #    - sys is the remote actor system name
          #    - hostname can be either hostname or IP address the remote actor
          #      should connect to
          #    - port should be the port for the remote server on the other node
          # The number of actor instances to be spawned is still taken from the
          # nr-of-instances setting as for local routers; the instances will be
          # distributed round-robin among the given nodes.
          nodes = []

        }
      }
    }
  }

  remote {

    # Which implementation of akka.remote.RemoteTransport to use
    # default is a TCP-based remote transport based on Netty
    transport = "akka.remote.netty.NettyRemoteTransport"

    # Enable untrusted mode for full security of server managed actors, allows
    # untrusted clients to connect.
    untrusted-mode = off

    # Timeout for ACK of cluster operations, like checking actor out etc.
    remote-daemon-ack-timeout = 30s

    # If this is "on", Akka will log all inbound messages at DEBUG level, if off then they are not
    log-received-messages = off

    # If this is "on", Akka will log all outbound messages at DEBUG level, if off then they are no
    log-sent-messages = off

    # If this is "on", Akka will log all RemoteLifeCycleEvents at the level defined for each, if o
    log-remote-lifecycle-events = off

    # Each property is annotated with (I) or (O) or (I&O), where I stands for "inbound" and O for
    # The NettyRemoteTransport always starts the server role to allow inbound connections, and it
    # active client connections whenever sending to a destination which is not yet connected; if o
    # it reuses inbound connections for replies, which is called a passive client connection (i.e
    # to client).
    netty {

      # (O) In case of increased latency / overflow how long should we wait (blocking the sender)
      # until we deem the send to be cancelled?
      # 0 means "never backoff", any positive number will indicate time to block at most.
      backoff-timeout = 0ms

      # (I&O) Generate your own with '$AKKA_HOME/scripts/generate_config_with_secure_cookie.sh'
      # or using 'akka.util.Crypt.generateSecureCookie'
```

```
    secure-cookie = ""

    # (I) Should the remote server require that its peers share the same secure-cookie
    # (defined in the 'remote' section)?
    require-cookie = off

    # (I) Reuse inbound connections for outbound messages
    use-passive-connections = on

    # (I) The hostname or ip to bind the remoting to,
    # InetAddress.getLocalHost.getHostAddress is used if empty
    hostname = ""

    # (I) The default remote server port clients should connect to.
    # Default is 2552 (AKKA), use 0 if you want a random available port
    # This port needs to be unique for each actor system on the same machine.
    port = 2552

    # (O) The address of a local network interface (IP Address) to bind to when creating
    # outbound connections. Set to "" or "auto" for automatic selection of local address.
    outbound-local-address = "auto"

    # (I&O) Increase this if you want to be able to send messages with large payloads
    message-frame-size = 1 MiB

    # (O) Timeout duration
    connection-timeout = 120s

    # (I) Sets the size of the connection backlog
    backlog = 4096

    # (I) Length in akka.time-unit how long core threads will be kept alive if idling
    execution-pool-keepalive = 60s

    # (I) Size of the core pool of the remote execution unit
    execution-pool-size = 4

    # (I) Maximum channel size, 0 for off
    max-channel-memory-size = 0b

    # (I) Maximum total size of all channels, 0 for off
    max-total-memory-size = 0b

    # (O) Time between reconnect attempts for active clients
    reconnect-delay = 5s

    # (O) Read inactivity period (lowest resolution is seconds)
    # after which active client connection is shutdown;
    # will be re-established in case of new communication requests.
    # A value of 0 will turn this feature off
    read-timeout = 0s

    # (O) Write inactivity period (lowest resolution is seconds)
    # after which a heartbeat is sent across the wire.
    # A value of 0 will turn this feature off
    write-timeout = 10s

    # (O) Inactivity period of both reads and writes (lowest resolution is seconds)
    # after which active client connection is shutdown;
    # will be re-established in case of new communication requests
    # A value of 0 will turn this feature off
    all-timeout = 0s
```

```
      # (O) Maximum time window that a client should try to reconnect for
      reconnection-time-window = 600s

      # (I&O) Used to configure the number of I/O worker threads on server sockets
      server-socket-worker-pool {
        # Min number of threads to cap factor-based number to
        pool-size-min = 2

        # The pool size factor is used to determine thread pool size
        # using the following formula: ceil(available processors * factor).
        # Resulting size is then bounded by the pool-size-min and
        # pool-size-max values.
        pool-size-factor = 2.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 128
      }

      # (I&O) Used to configure the number of I/O worker threads on client sockets
      client-socket-worker-pool {
        # Min number of threads to cap factor-based number to
        pool-size-min = 2

        # The pool size factor is used to determine thread pool size
        # using the following formula: ceil(available processors * factor).
        # Resulting size is then bounded by the pool-size-min and
        # pool-size-max values.
        pool-size-factor = 2.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 128
      }
    }

    # The dispatcher used for the system actor "network-event-sender"
    network-event-sender-dispatcher {
      executor = thread-pool-executor
      type = PinnedDispatcher
    }
  }
}
```

4.11.2 Types of Remote Interaction

Akka has two ways of using remoting:

- Lookup : used to look up an actor on a remote node with `actorFor(path)`
- Creation : used to create an actor on a remote node with `actorOf(Props(...), actorName)`

In the next sections the two alternatives are described in detail.

4.11.3 Looking up Remote Actors

`actorFor(path)` will obtain an `ActorRef` to an Actor on a remote node, e.g.:

```
val actor = context.actorFor("akka://actorSystemName@10.0.0.1:2552/user/actorName")
```

As you can see from the example above the following pattern is used to find an `ActorRef` on a remote node:

```
akka://<actor system>@<hostname>:<port>/<actor path>
```

Once you obtained a reference to the actor you can interact with it they same way you would with a local actor, e.g.:

```
actor ! "Pretty awesome feature"
```

For more details on how actor addresses and paths are formed and used, please refer to *Actor References, Paths and Addresses*.

4.11.4 Creating Actors Remotely

If you want to use the creation functionality in Akka remoting you have to further amend the `application.conf` file in the following way (only showing deployment section):

```
akka {
  actor {
    deployment {
      /sampleActor {
        remote = "akka://sampleActorSystem@127.0.0.1:2553"
      }
    }
  }
}
```

The configuration above instructs Akka to react when an actor with path `/sampleActor` is created, i.e. using `system.actorOf(Props(...)`, `sampleActor)`. This specific actor will not be directly instantiated, but instead the remote daemon of the remote system will be asked to create the actor, which in this sample corresponds to `sampleActorSystem@127.0.0.1:2553`.

Once you have configured the properties above you would do the following in code:

```
class SampleActor extends Actor { def receive = { case _ => println("Got something") } }

val actor = context.actorOf(Props[SampleActor], "sampleActor")
actor ! "Pretty slick"
```

`SampleActor` has to be available to the runtimes using it, i.e. the classloader of the actor systems has to have a JAR containing the class.

Programmatic Remote Deployment

To allow dynamically deployed systems, it is also possible to include deployment configuration in the `Props` which are used to create an actor: this information is the equivalent of a deployment section from the configuration file, and if both are given, the external configuration takes precedence.

With these imports:

```
import akka.actor.{ Props, Deploy, Address, AddressFromURIString }
import akka.remote.RemoteScope
```

and a remote address like this:

```
val one = AddressFromURIString("akka://sys@host:1234")
val two = Address("akka", "sys", "host", 1234) // this gives the same
```

you can advise the system to create a child on that remote node like so:

```
val ref = system.actorOf(Props[Echo].withDeploy(Deploy(scope = RemoteScope(address))))
```

4.11.5 Serialization

When using remoting for actors you must ensure that the `props` and `messages` used for those actors are serializable. Failing to do so will cause the system to behave in an unintended way.

For more information please see *Serialization (Scala)*

4.11.6 Routers with Remote Destinations

It is absolutely feasible to combine remoting with *Routing (Scala)*. This is also done via configuration:

```
akka {
  actor {
    deployment {
      /serviceA/aggregation {
        router = "round-robin"
        nr-of-instances = 10
        target {
          nodes = ["akka://app@10.0.0.2:2552", "akka://app@10.0.0.3:2552"]
        }
      }
    }
  }
}
```

This configuration setting will clone the actor “aggregation” 10 times and deploy it evenly distributed across the two given target nodes.

4.11.7 Description of the Remoting Sample

There is a more extensive remote example that comes with the Akka distribution. Please have a look here for more information: Remote Sample This sample demonstrates both, remote deployment and look-up of remote actors. First, let us have a look at the common setup for both scenarios (this is `common.conf`):

```
akka {

  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }

  remote {
    netty {
      hostname = "127.0.0.1"
    }
  }
}
```

This enables the remoting by installing the `RemoteActorRefProvider` and chooses the default remote transport. All other options will be set specifically for each show case.

Note: Be sure to replace the default IP 127.0.0.1 with the real address the system is reachable by if you deploy onto multiple machines!

Remote Lookup

In order to look up a remote actor, that one must be created first. For this purpose, we configure an actor system to listen on port 2552 (this is a snippet from `application.conf`):

```
calculator {
  include "common"

  akka {
    remote.netty.port = 2552
  }
}
```

Then the actor must be created. For all code which follows, assume these imports:

```
import com.typesafe.config.ConfigFactory
import akka.actor.{ ActorRef, Props, Actor, ActorSystem }
```

The actor doing the work will be this one:

```
class SimpleCalculatorActor extends Actor {
  def receive = {
    case Add(n1, n2) ⇒
      println("Calculating %d + %d".format(n1, n2))
      sender ! AddResult(n1, n2, n1 + n2)
    case Subtract(n1, n2) ⇒
      println("Calculating %d - %d".format(n1, n2))
      sender ! SubtractResult(n1, n2, n1 - n2)
  }
}
```

and we start it within an actor system using the above configuration

```
val system = ActorSystem("CalculatorApplication", ConfigFactory.load.getConfig("calculator"))
val actor = system.actorOf(Props[SimpleCalculatorActor], "simpleCalculator")
```

With the service actor up and running, we may look it up from another actor system, which will be configured to use port 2553 (this is a snippet from `application.conf`).

```
remotelookup {
  include "common"

  akka {
    remote.netty.port = 2553
  }
}
```

The actor which will query the calculator is a quite simple one for demonstration purposes

```
class LookupActor extends Actor {
  def receive = {
    case (actor: ActorRef, op: MathOp) ⇒ actor ! op
    case result: MathResult ⇒ result match {
      case AddResult(n1, n2, r)      ⇒ println("Add result: %d + %d = %d".format(n1, n2, r))
      case SubtractResult(n1, n2, r) ⇒ println("Sub result: %d - %d = %d".format(n1, n2, r))
    }
  }
}
```

and it is created from an actor system using the aforementioned client's config.

```
val system = ActorSystem("LookupApplication", ConfigFactory.load.getConfig("remotelookup"))
val actor = system.actorOf(Props[LookupActor], "lookupActor")
val remoteActor = system.actorFor("akka://CalculatorApplication@127.0.0.1:2552/user/simpleCalculat

def doSomething(op: MathOp) = {
  actor ! (remoteActor, op)
}
```

Requests which come in via `doSomething` will be sent to the client actor along with the reference which was looked up earlier. Observe how the actor system name using in `actorFor` matches the remote system's name, as do IP and port number. Top-level actors are always created below the `"/user"` guardian, which supervises them.

Remote Deployment

Creating remote actors instead of looking them up is not visible in the source code, only in the configuration file. This section is used in this scenario (this is a snippet from `application.conf`):

```
remotecreation {
  include "common"

  akka {
    actor {
      deployment {
        /advancedCalculator {
          remote = "akka://CalculatorApplication@127.0.0.1:2552"
        }
      }
    }

    remote.netty.port = 2554
  }
}
```

For all code which follows, assume these imports:

```
import com.typesafe.config.ConfigFactory
import akka.actor.{ ActorRef, Props, Actor, ActorSystem }
```

The client actor looks like in the previous example

```
class CreationActor extends Actor {
  def receive = {
    case (actor: ActorRef, op: MathOp) ⇒ actor ! op
    case result: MathResult ⇒ result match {
      case MultiplicationResult(n1, n2, r) ⇒ println("Mul result: %d * %d = %d".format(n1, n2, r)
      case DivisionResult(n1, n2, r)       ⇒ println("Div result: %.0f / %d = %.2f".format(n1, n2
    }
  }
}
```

but the setup uses only `actorOf`:

```
val system = ActorSystem("RemoteCreation", ConfigFactory.load.getConfig("remotecreation"))
val localActor = system.actorOf(Props[CreationActor], "creationActor")
val remoteActor = system.actorOf(Props[AdvancedCalculatorActor], "advancedCalculator")

def doSomething(op: MathOp) = {
  localActor ! (remoteActor, op)
}
```

Observe how the name of the server actor matches the deployment given in the configuration file, which will transparently delegate the actor creation to the remote node.

Remote Events

It is possible to listen to events that occur in Akka Remote, and to subscribe/unsubscribe to there events, you simply register as listener to the below described types in on the `ActorSystem.eventStream`.

Note: To subscribe to any outbound-related events, subscribe to `RemoteClientLifeCycleEvent` To subscribe to any inbound-related events, subscribe to `RemoteServerLifeCycleEvent` To subscribe to any remote events, subscribe to `RemoteLifeCycleEvent`

To intercept when an outbound connection is disconnected, you listen to `RemoteClientDisconnected` which holds the transport used (RemoteTransport) and the outbound address that was disconnected (Address).

To intercept when an outbound connection is connected, you listen to `RemoteClientConnected` which holds the transport used (RemoteTransport) and the outbound address that was connected to (Address).

To intercept when an outbound client is started you listen to `RemoteClientStarted` which holds the transport used (RemoteTransport) and the outbound address that it is connected to (Address).

To intercept when an outbound client is shut down you listen to `RemoteClientShutdown` which holds the transport used (RemoteTransport) and the outbound address that it was connected to (Address).

To intercept when an outbound message cannot be sent, you listen to `RemoteClientWriteFailed` which holds the payload that was not written (AnyRef), the cause of the failed send (Throwable), the transport used (RemoteTransport) and the outbound address that was the destination (Address).

For general outbound-related errors, that do not classify as any of the others, you can listen to `RemoteClientError`, which holds the cause (Throwable), the transport used (RemoteTransport) and the outbound address (Address).

To intercept when an inbound server is started (typically only once) you listen to `RemoteServerStarted` which holds the transport that it will use (RemoteTransport).

To intercept when an inbound server is shut down (typically only once) you listen to `RemoteServerShutdown` which holds the transport that it used (RemoteTransport).

To intercept when an inbound connection has been established you listen to `RemoteServerClientConnected` which holds the transport used (RemoteTransport) and optionally the address that connected (Option[Address]).

To intercept when an inbound connection has been disconnected you listen to `RemoteServerClientDisconnected` which holds the transport used (RemoteTransport) and optionally the address that disconnected (Option[Address]).

To intercept when an inbound remote client has been closed you listen to `RemoteServerClientClosed` which holds the transport used (RemoteTransport) and optionally the address of the remote client that was closed (Option[Address]).

4.12 Serialization (Scala)

Akka has a built-in Extension for serialization, and it is both possible to use the built-in serializers and to write your own.

The serialization mechanism is both used by Akka internally to serialize messages, and available for ad-hoc serialization of whatever you might need it for.

4.12.1 Usage

Configuration

For Akka to know which `Serializer` to use for what, you need edit your *Configuration*, in the "akka.actor.serializers"-section you bind names to implementations of the `akka.serialization.Serializer` you wish to use, like this:

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
```

```
      serializers {
        java = "akka.serialization.JavaSerializer"
        proto = "akka.serialization.ProtobufSerializer"
        myown = "akka.docs.serialization.MyOwnSerializer"
      }
    }
  }
""")
```

After you've bound names to different implementations of `Serializer` you need to wire which classes should be serialized using which `Serializer`, this is done in the "akka.actor.serialization-bindings"-section:

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
      serializers {
        java = "akka.serialization.JavaSerializer"
        proto = "akka.serialization.ProtobufSerializer"
        myown = "akka.docs.serialization.MyOwnSerializer"
      }

      serialization-bindings {
        "java.lang.String" = java
        "akka.docs.serialization.Customer" = java
        "com.google.protobuf.Message" = proto
        "akka.docs.serialization.MyOwnSerializable" = myown
        "java.lang.Boolean" = myown
      }
    }
  }
""")
```

You only need to specify the name of an interface or abstract base class of the messages. In case of ambiguity, i.e. the message implements several of the configured classes, the most specific configured class will be used, i.e. the one of which all other candidates are superclasses. If this condition cannot be met, because e.g. `java.io.Serializable` and `MyOwnSerializable` both apply and neither is a subtype of the other, a warning will be issued

Akka provides serializers for `java.io.Serializable` and protobuf `com.google.protobuf.GeneratedMessage` by default (the latter only if depending on the akka-remote module), so normally you don't need to add configuration for that; since `com.google.protobuf.GeneratedMessage` implements `java.io.Serializable`, protobuf messages will always by serialized using the protobuf protocol unless specifically overridden. In order to disable a default serializer, map its marker type to "none":

```
akka.actor.serialization-bindings {
  "java.io.Serializable" = none
}
```

Verification

If you want to verify that your messages are serializable you can enable the following config option:

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
      serialize-messages = on
    }
  }
""")
```

Warning: We only recommend using the config option turned on when you're running tests. It is completely pointless to have it turned on in other scenarios.

If you want to verify that your `Props` are serializable you can enable the following config option:

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
      serialize-creators = on
    }
  }
""")
```

Warning: We only recommend using the config option turned on when you're running tests. It is completely pointless to have it turned on in other scenarios.

Programmatic

If you want to programmatically serialize/deserialize using Akka Serialization, here's some examples:

```
import akka.actor.{ ActorRef, ActorSystem }
import akka.serialization._
import com.typesafe.config.ConfigFactory

    val system = ActorSystem("example")

    // Get the Serialization Extension
    val serialization = SerializationExtension(system)

    // Have something to serialize
    val original = "woohoo"

    // Find the Serializer for it
    val serializer = serialization.findSerializerFor(original)

    // Turn it into bytes
    val bytes = serializer.toBinary(original)

    // Turn it back into an object
    val back = serializer.fromBinary(bytes, manifest = None)

    // Voilá!
    back must equal(original)
```

For more information, have a look at the `ScalaDoc` for `akka.serialization._`

4.12.2 Customization

So, lets say that you want to create your own `Serializer`, you saw the `akka.docs.serialization.MyOwnSerializer` in the config example above?

Creating new Serializers

First you need to create a class definition of your `Serializer` like so:

```
import akka.actor.{ ActorRef, ActorSystem }
import akka.serialization._
import com.typesafe.config.ConfigFactory
```

```
class MyOwnSerializer extends Serializer {

  // This is whether "fromBinary" requires a "clazz" or not
  def includeManifest: Boolean = false

  // Pick a unique identifier for your Serializer,
  // you've got a couple of billions to choose from,
  // 0 - 16 is reserved by Akka itself
  def identifier = 1234567

  // "toBinary" serializes the given object to an Array of Bytes
  def toBinary(obj: AnyRef): Array[Byte] = {
    // Put the code that serializes the object here
    // ... ...
  }

  // "fromBinary" deserializes the given array,
  // using the type hint (if any, see "includeManifest" above)
  // into the optionally provided classLoader.
  def fromBinary(bytes: Array[Byte],
                 clazz: Option[Class[_]]): AnyRef = {
    // Put your code that deserializes here
    // ... ...
  }
}
```

Then you only need to fill in the blanks, bind it to a name in your *Configuration* and then list which classes that should be serialized using it.

Serializing ActorRefs

All ActorRefs are serializable using JavaSerializer, but in case you are writing your own serializer, you might want to know how to serialize and deserialize them properly, here's the magic incantation:

```
import akka.actor.{ ActorRef, ActorSystem }
import akka.serialization._
import com.typesafe.config.ConfigFactory

    // Serialize
    // (beneath toBinary)

    // If there is no transportAddress,
    // it means that either this Serializer isn't called
    // within a piece of code that sets it,
    // so either you need to supply your own,
    // or simply use the local path.
    val identifier: String = Serialization.currentTransportAddress.value match {
      case null    ⇒ theActorRef.path.toString
      case address ⇒ theActorRef.path.toStringWithAddress(address)
    }
    // Then just serialize the identifier however you like

    // Deserialize
    // (beneath fromBinary)
    val deserializedActorRef = theActorSystem actorFor identifier
    // Then just use the ActorRef
```

Note: `ActorPath.toStringWithAddress` only differs from `toString` if the address does not already have `host` and `port` components, i.e. it only inserts address information for local addresses.

This assumes that serialization happens in the context of sending a message through the remote transport. There are other uses of serialization, though, e.g. storing actor references outside of an actor application (database, durable mailbox, etc.). In this case, it is important to keep in mind that the address part of an actor's path determines how that actor is communicated with. Storing a local actor path might be the right choice if the retrieval happens in the same logical context, but it is not enough when deserializing it on a different network host: for that it would need to include the system's remote transport address. An actor system is not limited to having just one remote transport per se, which makes this question a bit more interesting.

In the general case, the local address to be used depends on the type of remote address which shall be the recipient of the serialized information. Use `ActorRefProvider.getExternalAddressFor(remoteAddr)` to query the system for the appropriate address to use when sending to `remoteAddr`:

```
object ExternalAddress extends ExtensionKey[ExternalAddressExt]

class ExternalAddressExt(system: ExtendedActorSystem) extends Extension {
  def addressFor(remoteAddr: Address): Address =
    system.provider.getExternalAddressFor(remoteAddr) getOrElse
      (throw new UnsupportedOperationException("cannot send to " + remoteAddr))
}

def serializeTo(ref: ActorRef, remote: Address): String =
  ref.path.toStringWithAddress(ExternalAddress(theActorSystem).addressFor(remote))
```

This requires that you know at least which type of address will be supported by the system which will deserialize the resulting actor reference; if you have no concrete address handy you can create a dummy one for the right protocol using `Address(protocol, "", "", 0)` (assuming that the actual transport used is as lenient as Akka's RemoteActorRefProvider).

There is a possible simplification available if you are just using the default `NettyRemoteTransport` with the `RemoteActorRefProvider`, which is enabled by the fact that this combination has just a single remote address:

```
object ExternalAddress extends ExtensionKey[ExternalAddressExt]

class ExternalAddressExt(system: ExtendedActorSystem) extends Extension {
  def addressForAkka: Address = system.provider match {
    case r: RemoteActorRefProvider ⇒ r.transport.address
    case _ ⇒
      throw new UnsupportedOperationException(
        "this method requires the RemoteActorRefProvider to be configured")
  }
}

def serializeAkkaDefault(ref: ActorRef): String =
  ref.path.toStringWithAddress(ExternalAddress(theActorSystem).addressForAkka)
```

This solution has to be adapted once other providers are used (like the planned extensions for clustering).

Deep serialization of Actors

The current recommended approach to do deep serialization of internal actor state is to use Event Sourcing, for more reading on the topic, see these examples:

Martin Krasser on EventSourcing Part1

Martin Krasser on EventSourcing Part2

Note: Built-in API support for persisting Actors will come in a later release, see the roadmap for more info:

Akka 2.0 roadmap

4.12.3 A Word About Java Serialization

When using Java serialization without employing the `JavaSerializer` for the task, you must make sure to supply a valid `ExtendedActorSystem` in the dynamic variable `JavaSerializer.currentSystem`. This is used when reading in the representation of an `ActorRef` for turning the string representation into a real reference. `DynamicVariable` is a thread-local variable, so be sure to have it set while deserializing anything which might contain actor references.

4.13 FSM

4.13.1 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'.*

4.13.2 A Simple Example

To demonstrate most of the features of the `FSM` trait, consider an actor which shall receive and queue messages while they arrive in a burst and send them on after the burst ended or a flush request is received.

First, consider all of the below to use these import statements:

```
import akka.actor.{ Actor, ActorRef, FSM }
import akka.util.duration._
```

The contract of our "Buncher" actor is that is accepts or produces the following messages:

```
// received events
case class SetTarget(ref: ActorRef)
case class Queue(obj: Any)
case object Flush

// sent events
case class Batch(obj: Seq[Any])
```

`SetTarget` is needed for starting it up, setting the destination for the `Batches` to be passed on; `Queue` will add to the internal queue while `Flush` will mark the end of a burst.

```
// states
sealed trait State
case object Idle extends State
case object Active extends State

sealed trait Data
case object Uninitialized extends Data
case class Todo(target: ActorRef, queue: Seq[Any]) extends Data
```

The actor can be in two states: no message queued (aka `Idle`) or some message queued (aka `Active`). It will stay in the active state as long as messages keep arriving and no flush is requested. The internal state data of the actor is made up of the target actor reference to send the batches to and the actual queue of messages.

Now let's take a look at the skeleton for our FSM actor:

```
class Buncher extends Actor with FSM[State, Data] {

  startWith(Idle, Uninitialized)

  when(Idle) {
    case Event(SetTarget(ref), Uninitialized) ⇒ stay using Todo(ref, Vector.empty)
  }

  // transition elided ...

  when(Active, stateTimeout = 1 second) {
    case Event(Flush | FSM.StateTimeout, t: Todo) ⇒ goto(Idle) using t.copy(queue = Vector.empty)
  }

  // unhandled elided ...

  initialize
}
```

The basic strategy is to declare the actor, mixing in the `FSM` trait and specifying the possible states and data values as type parameters. Within the body of the actor a DSL is used for declaring the state machine:

- `startsWith` defines the initial state and initial data
- then there is one `when(<state>) { ... }` declaration per state to be handled (could potentially be multiple ones, the passed `PartialFunction` will be concatenated using `orElse`)
- finally starting it up using `initialize`, which performs the transition into the initial state and sets up timers (if required).

In this case, we start out in the `Idle` and `Uninitialized` state, where only the `SetTarget()` message is handled; `stay` prepares to end this event's processing for not leaving the current state, while the `using` modifier makes the FSM replace the internal state (which is `Uninitialized` at this point) with a fresh `Todo()` object containing the target actor reference. The `Active` state has a state timeout declared, which means that if no message is received for 1 second, a `FSM.StateTimeout` message will be generated. This has the same effect as receiving the `Flush` command in this case, namely to transition back into the `Idle` state and resetting the internal queue to the empty vector. But how do messages get queued? Since this shall work identically in both states, we make use of the fact that any event which is not handled by the `when()` block is passed to the `whenUnhandled()` block:

```
whenUnhandled {
  // common code for both states
  case Event(Queue(obj), t @ Todo(_, v)) ⇒
    goto(Active) using t.copy(queue = v :+ obj)

  case Event(e, s) ⇒
    log.warning("received unhandled request {} in state {}/{}", e, stateName, s)
    stay
}
```

The first case handled here is adding `Queue()` requests to the internal queue and going to the `Active` state (this does the obvious thing of staying in the `Active` state if already there), but only if the FSM data are not `Uninitialized` when the `Queue()` event is received. Otherwise—and in all other non-handled cases—the second case just logs a warning and does not change the internal state.

The only missing piece is where the `Batches` are actually sent to the target, for which we use the `onTransition` mechanism: you can declare multiple such blocks and all of them will be tried for matching behavior in case a state transition occurs (i.e. only when the state actually changes).

```
onTransition {
  case Active -> Idle ⇒
    stateData match {
      case Todo(ref, queue) ⇒ ref ! Batch(queue)
```

```
    }
}
```

The transition callback is a partial function which takes as input a pair of states—the current and the next state. The FSM trait includes a convenience extractor for these in form of an arrow operator, which conveniently reminds you of the direction of the state change which is being matched. During the state change, the old state data is available via `stateData` as shown, and the new state data would be available as `nextStateData`.

To verify that this buncher actually works, it is quite easy to write a test using the *Testing Actor Systems (Scala)*, which is conveniently bundled with ScalaTest traits into `AkkaSpec`:

```
import akka.testkit.AkkaSpec
import akka.actor.Props

class FSMDocSpec extends AkkaSpec {

  "simple finite state machine" must {
    // fsm code elided ...

    "batch correctly" in {
      val buncher = system.actorOf(Props(new Buncher))
      buncher ! SetTarget(testActor)
      buncher ! Queue(42)
      buncher ! Queue(43)
      expectMsg(Batch(Seq(42, 43)))
      buncher ! Queue(44)
      buncher ! Flush
      buncher ! Queue(45)
      expectMsg(Batch(Seq(44)))
      expectMsg(Batch(Seq(45)))
    }

    "batch not if uninitialized" in {
      val buncher = system.actorOf(Props(new Buncher))
      buncher ! Queue(42)
      expectNoMsg
    }
  }
}
```

4.13.3 Reference

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 if you want to 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 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 Event(Start(msg), _) =>
    goto(Timer) using (msg, sender)
}

when(Timer, stateTimeout = 12 seconds) {
  case Event(StateTimeout, (msg, sender)) =>
    sender ! msg
    goto(Idle)
}
```

The `Event(msg: Any, data: D)` case class is parameterized with the data type held by the FSM for convenient pattern matching.

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) =>
    log.info(this, "Received unhandled event: " + x)
    stay
  case Event(msg, _) =>
    log.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 Event(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 => log.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 Event(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`.

4.13.4 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 *:ref:'configuration* 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) =>
      log.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.

4.13.5 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

4.14 Software Transactional Memory (Scala)

4.14.1 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.

The use of STM in Akka is inspired by the concepts and views in Clojure's STM. Please take the time to read this excellent document about state in clojure and view this presentation by Rich Hickey (the genius behind Clojure).

4.14.2 Scala STM

The STM supported in Akka is ScalaSTM which will be soon included in the Scala standard library.

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.

4.14.3 Persistent Datastructures

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.

4.14.4 Integration with Actors

In Akka we've also integrated Actors and STM in *Agents (Scala)* and *Transactors (Scala)*.

4.15 Agents (Scala)

Agents in Akka are 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 Scala STM - any dispatches made in a transaction are held until that transaction commits, and are discarded if it is retried or aborted.

4.15.1 Creating and stopping Agents

Agents are created by invoking `Agent(value)` passing in the Agent's initial value:

```
import akka.agent.Agent

val agent = Agent(5)
```

Note that creating an Agent requires an implicit `ActorSystem` (for creating the underlying actors). See *Actor Systems* for more information about actor systems. An ActorSystem can be in implicit scope when creating an Agent:

```
import akka.actor.ActorSystem
import akka.agent.Agent

implicit val system = ActorSystem("app")
```

```
val agent = Agent(5)
```

Or the ActorSystem can be passed explicitly when creating an Agent:

```
import akka.actor.ActorSystem
import akka.agent.Agent

val system = ActorSystem("app")

val agent = Agent(5)(system)
```

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()
```

4.15.2 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)
```

4.15.3 Reading an Agent's value

Agents can be dereferenced (you can get an Agent's value) by invoking the Agent with parentheses 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.

4.15.4 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`:

```
import akka.util.duration._
import akka.util.Timeout
```

```
implicit val timeout = Timeout(5 seconds)
val result = agent.await
```

You can also get a `Future` to this value, that will be completed after the currently queued updates have completed:

```
import akka.dispatch.Await

implicit val timeout = Timeout(5 seconds)
val future = agent.future
val result = Await.result(future, timeout.duration)
```

4.15.5 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. Here's an example:

```
import akka.agent.Agent
import akka.util.duration._
import akka.util.Timeout
import scala.concurrent.stm._

def transfer(from: Agent[Int], to: Agent[Int], amount: Int): Boolean = {
  atomic { txn ⇒
    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)

implicit val timeout = Timeout(5 seconds)
val fromValue = from.await // -> 50
val toValue = to.await // -> 70
```

4.15.6 Monadic usage

Agents are also monadic, allowing you to compose operations using for-comprehensions. In 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 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
```

```
// or using map directly
val agent4 = agent1 map (_ + 1)

// uses flatMap
val agent5 = for {
  value1 ← agent1
  value2 ← agent2
} yield value1 + value2
```

4.16 Transactors (Scala)

4.16.1 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.

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.

4.16.2 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.

4.16.3 Coordinated transactions

Akka provides an explicit mechanism for coordinating transactions across Actors. Under the hood it uses a `CommitBarrier`, 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.actor._
import akka.transactor._
import scala.concurrent.stm._
```

```
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 { implicit t ⇒
        count transform (_ + 1)
      }
    }
    case GetCount ⇒ sender ! count.single.get
  }
}
```

```
import akka.dispatch.Await
import akka.util.duration._
import akka.util.Timeout
import akka.pattern.ask

val system = ActorSystem("app")

val counter1 = system.actorOf(Props[Counter], name = "counter1")
val counter2 = system.actorOf(Props[Counter], name = "counter2")

implicit val timeout = Timeout(5 seconds)

counter1 ! Coordinated(Increment(Some(counter2)))

val count = Await.result(counter1 ? GetCount, timeout.duration)

// count == 1
```

Note that creating a `Coordinated` object requires a `Timeout` to be specified for the coordinated transaction. This can be done implicitly, by having an implicit `Timeout` in scope, or explicitly, by passing the timeout when creating a a `Coordinated` object. Here's an example of specifying an implicit timeout:

```
import akka.util.duration._
import akka.util.Timeout

implicit val timeout = Timeout(5 seconds)
```

To start a new coordinated transaction that you will also participate in, just create a `Coordinated` object (this assumes an implicit timeout):

```
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) ⇒ {
    // coordinated atomic ...
  }
}
```

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 { implicit t ⇒
  // do something in the coordinated transaction ...
}
```

The coordinated transaction will wait for the other transactions before committing. If any of the coordinated transactions fail then they all fail.

Note: The same actor should not be added to a coordinated transaction more than once. The transaction will not be able to complete as an actor only processes a single message at a time. When processing the first message the coordinated transaction will wait for the commit barrier, which in turn needs the second message to be received to proceed.

4.16.4 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._
import scala.concurrent.stm._

case object Increment

class Counter extends Transactor {
  val count = Ref(0)

  def atomically = implicit txn ⇒ {
    case Increment ⇒ count transform (_ + 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.actor._
import akka.transactor._
import scala.concurrent.stm._

case object Increment

class FriendlyCounter(friend: ActorRef) extends Transactor {
  val count = Ref(0)

  override def coordinate = {
    case Increment ⇒ include(friend)
```

```
  }

  def atomically = implicit txn ⇒ {
    case Increment ⇒ count transform (_ + 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 SomeMessage  ⇒ sendTo(someActor -> SomeOtherMessage)
  case OtherMessage ⇒ 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.

4.17 IO (Scala)

4.17.1 Introduction

This documentation is in progress and some sections may be incomplete. More will be coming.

4.17.2 Components

ByteString

A primary goal of Akka's IO module is to only communicate between actors with immutable objects. When dealing with network IO on the jvm `Array[Byte]` and `ByteBuffer` are commonly used to represent collections of `Bytes`, but they are mutable. Scala's collection library also lacks a suitably efficient immutable collection for `Bytes`. Being able to safely and efficiently move `Bytes` around is very important for this IO module, so `ByteString` was developed.

`ByteString` is a Rope-like data structure that is immutable and efficient. When 2 `ByteStrings` are concatenated together they are both stored within the resulting `ByteString` instead of copying both to a new `Array`. Operations such as `drop` and `take` return `ByteStrings` that still reference the original `Array`, but just change the offset and length that is visible. Great care has also been taken to make sure that the internal `Array` cannot be modified. Whenever a potentially unsafe `Array` is used to create a new `ByteString` a defensive copy is created.

`ByteString` inherits all methods from `IndexedSeq`, and it also has some new ones. For more information, look up the `akka.util.ByteString` class and it's companion object in the ScalaDoc.

IO.Handle

`IO.Handle` is an immutable reference to a Java NIO `Channel`. Passing mutable `Channels` between `Actors` could lead to unsafe behavior, so instead subclasses of the `IO.Handle` trait are used. Currently there are 2

concrete subclasses: `IO.SocketHandle` (representing a `SocketChannel`) and `IO.ServerHandle` (representing a `ServerSocketChannel`).

IOManager

The `IOManager` takes care of the low level IO details. Each `ActorSystem` has it's own `IOManager`, which can be accessed calling `IOManager(system: ActorSystem)`. `Actor`s communicate with the `IOManager` with specific messages. The messages sent from an `Actor` to the `IOManager` are handled automatically when using certain methods and the messages sent from an `IOManager` are handled within an `Actor`'s `receive` method.

Connecting to a remote host:

```
val address = new InetSocketAddress("remotehost", 80)
val socket = IOManager(actorSystem).connect(address)
```

```
val socket = IOManager(actorSystem).connect("remotehost", 80)
```

Creating a server:

```
val address = new InetSocketAddress("localhost", 80)
val serverSocket = IOManager(actorSystem).listen(address)
```

```
val serverSocket = IOManager(actorSystem).listen("localhost", 80)
```

Receiving messages from the `IOManager`:

```
def receive = {

  case IO.Listening(server, address) =>
    println("The server is listening on socket " + address)

  case IO.Connected(socket, address) =>
    println("Successfully connected to " + address)

  case IO.NewClient(server) =>
    println("New incoming connection on server")
    val socket = server.accept()
    println("Writing to new client socket")
    socket.write(bytes)
    println("Closing socket")
    socket.close()

  case IO.Read(socket, bytes) =>
    println("Received incoming data from socket")

  case IO.Closed(socket: IO.SocketHandle, cause) =>
    println("Socket has closed, cause: " + cause)

  case IO.Closed(server: IO.ServerHandle, cause) =>
    println("Server socket has closed, cause: " + cause)

}
```

IO.Iteratee

Included with Akka's IO module is a basic implementation of `Iteratees`. `Iteratees` are an effective way of handling a stream of data without needing to wait for all the data to arrive. This is especially useful when dealing with non blocking IO since we will usually receive data in chunks which may not include enough information to process, or it may contain much more data then we currently need.

This `Iteratee` implementation is much more basic then what is usually found. There is only support for `ByteString` input, and enumerators aren't used. The reason for this limited implementation is to reduce the amount of explicit type signatures needed and to keep things simple. It is important to note that Akka's `Iteratees` are completely optional, incoming data can be handled in any way, including other `Iteratee` libraries.

`Iteratees` work by processing the data that it is given and returning either the result (with any unused input) or a continuation if more input is needed. They are monadic, so methods like `flatMap` can be used to pass the result of an `Iteratee` to another.

The basic `Iteratees` included in the IO module can all be found in the ScalaDoc under `akka.actor.IO`, and some of them are covered in the example below.

4.17.3 Examples

Http Server

This example will create a simple high performance HTTP server. We begin with our imports:

```
import akka.actor._
import akka.util.{ ByteString, ByteStringBuilder }
import java.net.InetSocketAddress
```

Some commonly used constants:

```
object HttpConstants {
  val SP = ByteString(" ")
  val HT = ByteString("\t")
  val CRLF = ByteString("\r\n")
  val COLON = ByteString(":")
  val PERCENT = ByteString("%")
  val PATH = ByteString("/")
  val QUERY = ByteString("?")
}
```

And case classes to hold the resulting request:

```
case class Request(meth: String, path: List[String], query: Option[String], httpver: String, heade
case class Header(name: String, value: String)
```

Now for our first `Iteratee`. There are 3 main sections of a HTTP request: the request line, the headers, and an optional body. The main request `Iteratee` handles each section separately:

```
object HttpIteratees {
  import HttpConstants._

  def readRequest =
    for {
      requestLine ← readRequestLine
      (meth, (path, query), httpver) = requestLine
      headers ← readHeaders
      body ← readBody(headers)
    } yield Request(meth, path, query, httpver, headers, body)
```

In the above code `readRequest` takes the results of 3 different `Iteratees` (`readRequestLine`, `readHeaders`, `readBody`) and combines them into a single `Request` object. `readRequestLine` actually returns a tuple, so we extract it's individual components. `readBody` depends on values contained within the header section, so we must pass those to the method.

The request line has 3 parts to it: the HTTP method, the requested URI, and the HTTP version. The parts are separated by a single space, and the entire request line ends with a `CRLF`.

```
def ascii(bytes: ByteString): String = bytes.decodeString("US-ASCII").trim

def readRequestLine =
  for {
    meth ← IO takeUntil SP
    uri ← readRequestURI
    _ ← IO takeUntil SP // ignore the rest
    httpver ← IO takeUntil CRLF
  } yield (ascii(meth), uri, ascii(httpver))
```

Reading the request method is simple as it is a single string ending in a space. The simple `Iteratee` that performs this is `IO.takeUntil(delimiter: ByteString): Iteratee[ByteString]`. It keeps consuming input until the specified delimiter is found. Reading the HTTP version is also a simple string that ends with a `CRLF`.

The `ascii` method is a helper that takes a `ByteString` and parses it as a `US-ASCII String`.

Reading the request URI is a bit more complicated because we want to parse the individual components of the URI instead of just returning a simple string:

```
def readRequestURI = IO peek 1 flatMap {
  case PATH ⇒
    for {
      path ← readPath
      query ← readQuery
    } yield (path, query)
  case _ ⇒ sys.error("Not Implemented")
}
```

For this example we are only interested in handling absolute paths. To detect if we the URI is an absolute path we use `IO.peek(length: Int): Iteratee[ByteString]`, which returns a `ByteString` of the request length but doesn't actually consume the input. We peek at the next bit of input and see if it matches our `PATH` constant (defined above as `ByteString("/")`). If it doesn't match we throw an error, but for a more robust solution we would want to handle other valid URIs.

Next we handle the path itself:

```
def readPath = {
  def step(segments: List[String]): IO.Iteratee[List[String]] = IO peek 1 flatMap {
    case PATH ⇒ IO drop 1 flatMap (_ ⇒ readUriPart(pathchar) flatMap (segment ⇒ step(segment ::
    case _ ⇒ segments match {
      case "" :: rest ⇒ IO Done rest.reverse
      case _          ⇒ IO Done segments.reverse
    }
  }
  step(Nil)
}
```

The `step` method is a recursive method that takes a `List` of the accumulated path segments. It first checks if the remaining input starts with the `PATH` constant, and if it does, it drops that input, and returns the `readUriPart Iteratee` which has it's result added to the path segment accumulator and the `step` method is run again.

If after reading in a path segment the next input does not start with a path, we reverse the accumulated segments and return it (dropping the last segment if it is blank).

Following the path we read in the query (if it exists):

```
def readQuery: IO.Iteratee[Option[String]] = IO peek 1 flatMap {
  case QUERY ⇒ IO drop 1 flatMap (_ ⇒ readUriPart(querychar) map (Some(_)))
  case _     ⇒ IO Done None
}
```

It is much simpler then reading the path since we aren't doing any parsing of the query since there is no standard format of the query string.

Both the path and query used the `readUriPart` `Iteratee`, which is next:

```
val alpha = Set.empty ++ ('a' to 'z') ++ ('A' to 'Z') map (_.toByte)
val digit = Set.empty ++ ('0' to '9') map (_.toByte)
val hexdigit = digit ++ (Set.empty ++ ('a' to 'f') ++ ('A' to 'F') map (_.toByte))
val subdelim = Set('!', '$', '&', '\'', '(', ')', '*', '+', ',', ';', '=') map (_.toByte)
val pathchar = alpha ++ digit ++ subdelim ++ (Set(':', '@') map (_.toByte))
val querychar = pathchar ++ (Set('/', '?') map (_.toByte))

def readUriPart(allowed: Set[Byte]): IO.Iteratee[String] = for {
  str ← IO takeWhile allowed map ascii
  pchar ← IO peek 1 map (_ == PERCENT)
  all ← if (pchar) readPChar flatMap (ch ⇒ readUriPart(allowed) map (str + ch + _)) else IO Done
} yield all

def readPChar = IO take 3 map {
  case Seq('%', rest @ _*) if rest forall hexdigit ⇒
    java.lang.Integer.parseInt(rest map (_.toChar) mkString, 16).toChar
}
```

Here we have several `Sets` that contain valid characters pulled from the URI spec. The `readUriPart` method takes a `Set` of valid characters (already mapped to `Bytes`) and will continue to match characters until it reaches on that is not part of the `Set`. If it is a percent encoded character then that is handled as a valid character and processing continues, or else we are done collecting this part of the URI.

Headers are next:

```
def readHeaders = {
  def step(found: List[Header]): IO.Iteratee[List[Header]] = {
    IO peek 2 flatMap {
      case CRLF ⇒ IO takeUntil CRLF flatMap (_ ⇒ IO Done found)
      case _    ⇒ readHeader flatMap (header ⇒ step(header :: found))
    }
  }
  step(Nil)
}

def readHeader =
  for {
    name ← IO takeUntil COLON
    value ← IO takeUntil CRLF flatMap readMultiLineValue
  } yield Header(ascii(name), ascii(value))

def readMultiLineValue(initial: ByteString): IO.Iteratee[ByteString] = IO peek 1 flatMap {
  case SP ⇒ IO takeUntil CRLF flatMap (bytes ⇒ readMultiLineValue(initial ++ bytes))
  case _  ⇒ IO Done initial
}
```

And if applicable, we read in the message body:

```
def readBody(headers: List[Header]) =
  if (headers.exists(header ⇒ header.name == "Content-Length" || header.name == "Transfer-Encodir
    IO.takeAll map (Some(_))
  else
    IO Done None
```

Finally we get to the actual `Actor`:

```
class HttpServer(port: Int) extends Actor {

  val state = IO.IterateeRef.Map.async[IO.Handle]()(context.dispatcher)

  override def preStart {
    IOManager(context.system) listen new InetSocketAddress(port)
  }
```

```
  def receive = {

    case IO.NewClient(server) ⇒
      val socket = server.accept()
      state(socket) flatMap (_ ⇒ HttpServer.processRequest(socket))

    case IO.Read(socket, bytes) ⇒
      state(socket)(IO Chunk bytes)

    case IO.Closed(socket, cause) ⇒
      state(socket)(IO EOF None)
      state -= socket

  }

}
```

And it's companion object:

```
object HttpServer {
  import HttpIteratees._

  def processRequest(socket: IO.SocketHandle): IO.Iteratee[Unit] =
    IO repeat {
      for {
        request ← readRequest
      } yield {
        val rsp = request match {
          case Request("GET", "ping" :: Nil, _, _, headers, _) ⇒
            OKResponse(ByteString("<p>pong</p>"),
              request.headers.exists { case Header(n, v) ⇒ n.toLowerCase == "connection" && v.toL
          case req ⇒
            OKResponse(ByteString("<p>" + req.toString + "</p>"),
              request.headers.exists { case Header(n, v) ⇒ n.toLowerCase == "connection" && v.toL
        }
        socket write OKResponse.bytes(rsp).compact
        if (!rsp.keepAlive) socket.close()
      }
    }

}
```

And the OKResponse:

```
object OKResponse {
  import HttpConstants.CRLF

  val okStatus = ByteString("HTTP/1.1 200 OK")
  val contentType = ByteString("Content-Type: text/html; charset=utf-8")
  val cacheControl = ByteString("Cache-Control: no-cache")
  val date = ByteString("Date: ")
  val server = ByteString("Server: Akka")
  val contentLength = ByteString("Content-Length: ")
  val connection = ByteString("Connection: ")
  val keepAlive = ByteString("Keep-Alive")
  val close = ByteString("Close")

  def bytes(rsp: OKResponse) = {
    new ByteStringBuilder ++=
      okStatus ++= CRLF ++=
      contentType ++= CRLF ++=
      cacheControl ++= CRLF ++=
      date ++= ByteString(new java.util.Date().toString) ++= CRLF ++=
```

```
      server ++= CRLF ++=
      contentLength ++= ByteString(rsp.body.length.toString) ++= CRLF ++=
      connection ++= (if (rsp.keepAlive) keepAlive else close) ++= CRLF ++= CRLF ++= rsp.body resu
  }

}
case class OKResponse(body: ByteString, keepAlive: Boolean)
```

A `main` method to start everything up:

```
object Main extends App {
  val port = Option(System.getenv("PORT")) map (_.toInt) getOrElse 8080
  val system = ActorSystem()
  val server = system.actorOf(Props(new HttpServer(port)))
}
```

4.18 Testing Actor Systems (Scala)

4.18.1 TestKit Example (Scala)

Ray Roestenburg'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 system = ActorSystem()
  import system._
  val echoRef = actorOf(Props(new EchoActor))
  val forwardRef = actorOf(Props(new ForwardingActor(testActor)))
  val filterRef = actorOf(Props(new FilteringActor(testActor)))
  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(Props(new SequencingActor(testActor, headList, tailList)))

  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 ca
 */
class SequencingActor(next: ActorRef, head: List[String], tail: List[String]) extends Actor {
  def receive = {
    case msg => {
      head map (next ! _)
      next ! msg
      tail map (next ! _)
    }
  }
}
```

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.

4.18.2 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.

Expecting Exceptions

Testing that an expected exception is thrown while processing a message sent to the actor under test can be done by using a `TestActorRef` `receive` based invocation:

```
import akka.testkit.TestActorRef

val actorRef = TestActorRef(new Actor {
  def receive = {
    case boom ⇒ throw new IllegalArgumentException("boom")
  }
})
intercept[IllegalArgumentException] { actorRef.receive("hello") }
```

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.actor.FSM
import akka.util.duration._

val fsm = TestFSMRef(new Actor with FSM[Int, String] {
  startWith(1, "")
  when(1) {
    case Event("go", _) ⇒ goto(2) using "go"
  }
  when(2) {
    case Event("back", _) ⇒ goto(1) using "back"
  }
})
```

```
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` in response to external messages. 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`.

```
import akka.testkit.TestActorRef
import akka.util.duration._
import akka.dispatch.Await
import akka.pattern.ask

val actorRef = TestActorRef(new MyActor)
// hypothetical message stimulating a '42' answer
val result = Await.result((actorRef ? Say42), 5 seconds).asInstanceOf[Int]
result must be(42)
```

As the `TestActorRef` is a subclass of `LocalActorRef` with a few special extras, also aspects like supervision and restarting work properly, but beware that execution is only strictly synchronous 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 asynchronicity again (in most cases the problem will be to wait 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 None.

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 `receive` method `TestActorRef`, which will be forwarded to the underlying actor:

```
import akka.testkit.TestActorRef
system.eventStream.subscribe(testActor, classOf[UnhandledMessage])
val ref = TestActorRef[MyActor]
ref.receive(Unknown)
expectMsg(1 second, UnhandledMessage(Unknown, system.deadLetters, ref))
```

The above sample assumes the default behavior for unhandled messages, i.e. that the actor doesn't swallow all messages and doesn't override `unhandled`.

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.

4.18.3 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.

The `TestKit` class contains a collection of tools which makes this common task easy.

```
import akka.actor.ActorSystem
import akka.actor.Actor
import akka.actor.Props
import akka.testkit.TestKit
import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers
import org.scalatest.BeforeAndAfterAll
import akka.testkit.ImplicitSender

object MySpec {
  class EchoActor extends Actor {
    def receive = {
      case x ⇒ sender ! x
    }
  }
}

class MySpec(_system: ActorSystem) extends TestKit(_system) with ImplicitSender
  with WordSpec with MustMatchers with BeforeAndAfterAll {

  def this() = this(ActorSystem("MySpec"))

  import MySpec._

  override def afterAll {
    system.shutdown()
  }
```

```
  "An Echo actor" must {

    "send back messages unchanged" in {
      val echo = system.actorOf(Props[EchoActor])
      echo ! "hello world"
      expectMsg("hello world")
    }

  }
}
```

The `TestKit` contains an actor named `testActor` which is the entry point for messages to be examined with the various `expectMsg...` assertions detailed below. When mixing in the trait `ImplicitSender` this test actor is implicitly used as sender reference when dispatching messages from the test procedure. 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.

Remember to shut down the actor system after the test is finished (also in case of failure) so that all actors—including the test actor—are stopped.

Built-In Assertions

The above mentioned `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.

- `fishForMessage(max: Duration, hint: String)(pf: PartialFunction[Any, Boolean]): Any`

  Keep receiving messages as long as the time is not used up and the partial function matches and returns `false`. Returns the message received for which it returned `true` or throws an exception, which will include the provided hint for easier debugging.

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, messages: Int)(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
  - the number of messages has not yet reached the maximum

  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). The number of expected messages defaults to `Int.MaxValue`, which effectively disables this limit.

- `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

Since an integration test does not allow to the internal processing of the participating actors, verifying expected exceptions cannot be done directly. Instead, use the logging system for this purpose: replacing the normal event handler with the `TestEventListener` and using an `EventFilter` allows assertions on log messages, including those which are generated by exceptions:

```
import akka.testkit.EventFilter
import com.typesafe.config.ConfigFactory

implicit val system = ActorSystem("testsystem", ConfigFactory.parseString("""
  akka.event-handlers = ["akka.testkit.TestEventListener"]
  """))
try {
  val actor = system.actorOf(Props.empty)
  EventFilter[ActorKilledException](occurrences = 1) intercept {
    actor ! Kill
  }
} finally {
  system.shutdown()
}
```

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 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.

```
import akka.actor.Props
import akka.util.duration._

val worker = system.actorOf(Props[Worker])
within(200 millis) {
  worker ! "some work"
  expectMsg("some result")
  expectNoMsg // will block for the rest of the 200ms
  Thread.sleep(300) // 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 the *Configuration*, `akka.test.timefactor`, which defaults to 1.

You can scale other durations with the same factor by using the implicit conversion in `akka.testkit` package object to add dilated function to `Duration`.

```
import akka.util.duration._
import akka.testkit._
10.milliseconds.dilated
```

Resolving Conflicts with Implicit ActorRef

If you want the sender of messages inside your TestKit-based tests to be the `testActor` simply mix in `ÌmplicitSender` into your test.

```
class MySpec(_system: ActorSystem) extends TestKit(_system) with ImplicitSender
  with WordSpec with MustMatchers with BeforeAndAfterAll {
```

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:

```
import akka.testkit.TestProbe
import akka.util.duration._
import akka.actor._
import akka.dispatch.Futures

  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 = system.actorOf(Props[MyDoubleEcho])
    actor ! (probe1.ref, probe2.ref)
    actor ! "hello"
    probe1.expectMsg(500 millis, "hello")
    probe2.expectMsg(500 millis, "hello")
```

Here a the system under test is simulated by `MyDoubleEcho`, which is supposed to mirror its input to two outputs. Attaching two test probes enables verification of the (simplistic) behavior. Another example would be two actors A and B which collaborate by A sending messages to B. In order to verify this message flow, a `TestProbe` could be inserted as target of A, using the forwarding capabilities or auto-pilot described below to include a real B in the test setup.

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(system) {
  def expectUpdate(x: Int) = {
    expectMsgPF() {
      case Update(id, _) if id == x ⇒ true
    }
    sender ! "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.sender ! "world"
assert(future.isCompleted && future.value == Some(Right("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:

```
class Source(target: ActorRef) extends Actor {
  def receive = {
    case "start" ⇒ target ! "work"
  }
}

class Destination extends Actor {
  def receive = {
    case x ⇒ // Do something..
  }
}

  val probe = TestProbe()
  val source = system.actorOf(Props(new Source(probe.ref)))
  val dest = system.actorOf(Props[Destination])
  source ! "start"
  probe.expectMsg("work")
  probe.forward(dest)
```

The `dest` actor will receive the same message invocation as if no test probe had intervened.

Auto-Pilot

Receiving messages in a queue for later inspection is nice, but in order to keep a test running and verify traces later you can also install an `AutoPilot` in the participating test probes (actually in any `TestKit`) which is invoked

before enqueueing to the inspection queue. This code can be used to forward messages, e.g. in a chain `A --> Probe --> B`, as long as a certain protocol is obeyed.

```
val probe = TestProbe()
probe.setAutoPilot(new TestActor.AutoPilot {
  def run(sender: ActorRef, msg: Any): Option[TestActor.AutoPilot] =
    msg match {
      case "stop" ⇒ None
      case x      ⇒ testActor.tell(x, sender); Some(this)
    }
})
```

The `run` method must return the auto-pilot for the next message, wrapped in an `Option`; setting it to `None` terminates the auto-pilot.

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(_system: ActorSystem) with ImplicitSender {

  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.

4.18.4 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:

```
import akka.testkit.CallingThreadDispatcher
val ref = system.actorOf(Props[MyActor].withDispatcher(CallingThreadDispatcher.Id))
```

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 a 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 has 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

4.18.5 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 the *Configuration* — namely `akka.actor.debug.receive` — which enables the `loggable` statement to be applied to an actor's `receive` function:

```
import akka.event.LoggingReceive
def receive = LoggingReceive {
  case msg ⇒ // Do something...
}
```

. If the abovementioned setting is not given in the *Configuration*, 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, monitor start, monitor stop 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 {
  loglevel = DEBUG
  actor {
    debug {
      receive = on
      autoreceive = on
      lifecycle = on
    }
  }
}
```

4.19 Akka Extensions (Scala)

If you want to add features to Akka, there is a very elegant, but powerful mechanism for doing so. It's called Akka Extensions and is comprised of 2 basic components: an `Extension` and an `ExtensionId`.

Extensions will only be loaded once per `ActorSystem`, which will be managed by Akka. You can choose to have your Extension loaded on-demand or at `ActorSystem` creation time through the Akka configuration. Details on how to make that happens are below, in the "Loading from Configuration" section.

Warning: Since an extension is a way to hook into Akka itself, the implementor of the extension needs to ensure the thread safety of his/her extension.

4.19.1 Building an Extension

So let's create a sample extension that just lets us count the number of times something has happened.

First, we define what our `Extension` should do:

```
import akka.actor.Extension

class CountExtensionImpl extends Extension {
  //Since this Extension is a shared instance
  // per ActorSystem we need to be threadsafe
  private val counter = new AtomicLong(0)

  //This is the operation this Extension provides
  def increment() = counter.incrementAndGet()
}
```

Then we need to create an `ExtensionId` for our extension so we can grab ahold of it.

```
import akka.actor.ExtensionId
import akka.actor.ExtensionIdProvider
import akka.actor.ExtendedActorSystem

object CountExtension
  extends ExtensionId[CountExtensionImpl]
  with ExtensionIdProvider {
  //The lookup method is required by ExtensionIdProvider,
  // so we return ourselves here, this allows us
  // to configure our extension to be loaded when
  // the ActorSystem starts up
  override def lookup = CountExtension

  //This method will be called by Akka
  // to instantiate our Extension
  override def createExtension(system: ExtendedActorSystem) = new CountExtensionImpl
}
```

Wicked! Now all we need to do is to actually use it:

```
CountExtension(system).increment
```

Or from inside of an Akka Actor:

```
class MyActor extends Actor {
  def receive = {
    case someMessage ⇒
      CountExtension(context.system).increment()
  }
}
```

You can also hide extension behind traits:

```
trait Counting { self: Actor ⇒
  def increment() = CountExtension(context.system).increment()
}
class MyCounterActor extends Actor with Counting {
  def receive = {
    case someMessage ⇒ increment()
  }
}
```

That's all there is to it!

4.19.2 Loading from Configuration

To be able to load extensions from your Akka configuration you must add FQCNs of implementations of either `ExtensionId` or `ExtensionIdProvider` in the `akka.extensions` section of the config you provide to your `ActorSystem`.

```
akka {
  extensions = ["akka.docs.extension.CountExtension$"]
}
```

Note that in this case `CountExtension` is an object and therefore the class name ends with `$`.

4.19.3 Applicability

The sky is the limit! By the way, did you know that Akka's `Typed Actors`, `Serialization` and other features are implemented as Akka Extensions?

Application specific settings

The *Configuration* can be used for application specific settings. A good practice is to place those settings in an Extension.

Sample configuration:

```
myapp {
  db {
    uri = "mongodb://example1.com:27017,example2.com:27017"
  }
  circuit-breaker {
    timeout = 30 seconds
  }
}
```

The `Extension`:

```
import akka.actor.Extension
import akka.actor.ExtensionId
import akka.actor.ExtensionIdProvider
import akka.actor.ExtendedActorSystem
import akka.util.Duration
import com.typesafe.config.Config
import java.util.concurrent.TimeUnit

class SettingsImpl(config: Config) extends Extension {
  val DbUri: String = config.getString("myapp.db.uri")
  val CircuitBreakerTimeout: Duration = Duration(config.getMilliseconds("myapp.circuit-breaker.ti
}
object Settings extends ExtensionId[SettingsImpl] with ExtensionIdProvider {

  override def lookup = Settings

  override def createExtension(system: ExtendedActorSystem) = new SettingsImpl(system.settings.co
}
```

Use it:

```
class MyActor extends Actor {
  val settings = Settings(context.system)
  val connection = connect(settings.DbUri, settings.CircuitBreakerTimeout)
```

4.20 ZeroMQ (Scala)

Akka provides a ZeroMQ module which abstracts a ZeroMQ connection and therefore allows interaction between Akka actors to take place over ZeroMQ connections. The messages can be of a proprietary format or they can be

defined using Protobuf. The socket actor is fault-tolerant by default and when you use the newSocket method to create new sockets it will properly reinitialize the socket.

ZeroMQ is very opinionated when it comes to multi-threading so configuration option *akka.zeromq.socket-dispatcher* always needs to be configured to a PinnedDispatcher, because the actual ZeroMQ socket can only be accessed by the thread that created it.

The ZeroMQ module for Akka is written against an API introduced in JZMQ, which uses JNI to interact with the native ZeroMQ library. Instead of using JZMQ, the module uses ZeroMQ binding for Scala that uses the native ZeroMQ library through JNA. In other words, the only native library that this module requires is the native ZeroMQ library. The benefit of the scala library is that you don't need to compile and manage native dependencies at the cost of some runtime performance. The scala-bindings are compatible with the JNI bindings so they are a drop-in replacement, in case you really need to get that extra bit of performance out.

4.20.1 Connection

ZeroMQ supports multiple connectivity patterns, each aimed to meet a different set of requirements. Currently, this module supports publisher-subscriber connections and connections based on dealers and routers. For connecting or accepting connections, a socket must be created. Sockets are always created using the `akka.zeromq.ZeroMQExtension`, for example:

```
import akka.zeromq.ZeroMQExtension
val pubSocket = ZeroMQExtension(system).newSocket(SocketType.Pub, Bind("tcp://127.0.0.1:1234"))
```

or by importing the `akka.zeromq._` package to make newSocket method available on system, via an implicit conversion.

```
import akka.zeromq._
val pubSocket2 = system.newSocket(SocketType.Pub, Bind("tcp://127.0.0.1:1234"))
```

Above examples will create a ZeroMQ Publisher socket that is Bound to the port 1234 on localhost.

Similarly you can create a subscription socket, with a listener, that subscribes to all messages from the publisher using:

```
import akka.zeromq._
val listener = system.actorOf(Props(new Actor {
  def receive: Receive = {
    case Connecting    ⇒ //...
    case m: ZMQMessage ⇒ //...
    case _             ⇒ //...
  }
}))
val subSocket = system.newSocket(SocketType.Sub, Listener(listener), Connect("tcp://127.0.0.1:123
```

The following sub-sections describe the supported connection patterns and how they can be used in an Akka environment. However, for a comprehensive discussion of connection patterns, please refer to ZeroMQ – The Guide.

Publisher-Subscriber Connection

In a publisher-subscriber (pub-sub) connection, the publisher accepts one or more subscribers. Each subscriber shall subscribe to one or more topics, whereas the publisher publishes messages to a set of topics. Also, a subscriber can subscribe to all available topics. In an Akka environment, pub-sub connections shall be used when an actor sends messages to one or more actors that do not interact with the actor that sent the message.

When you're using zeromq pub/sub you should be aware that it needs multicast - check your cloud - to work properly and that the filtering of events for topics happens client side, so all events are always broadcasted to every subscriber.

An actor is subscribed to a topic as follows:

```
val subTopicSocket = system.newSocket(SocketType.Sub, Listener(listener), Connect("tcp://127.0.0.1
```

It is a prefix match so it is subscribed to all topics starting with `foo.bar`. Note that if the given string is empty or `SubscribeAll` is used, the actor is subscribed to all topics.

To unsubscribe from a topic you do the following:

```
subTopicSocket ! Unsubscribe("foo.bar")
```

To publish messages to a topic you must use two Frames with the topic in the first frame.

```
pubSocket ! ZMQMessage(Seq(Frame("foo.bar"), Frame(payload)))
```

Pub-Sub in Action

The following example illustrates one publisher with two subscribers.

The publisher monitors current heap usage and system load and periodically publishes `Heap` events on the `"health.heap"` topic and `Load` events on the `"health.load"` topic.

```
import akka.zeromq._
import akka.actor.Actor
import akka.actor.Props
import akka.actor.ActorLogging
import akka.serialization.SerializationExtension
import java.lang.management.ManagementFactory

case object Tick
case class Heap(timestamp: Long, used: Long, max: Long)
case class Load(timestamp: Long, loadAverage: Double)

class HealthProbe extends Actor {

  val pubSocket = context.system.newSocket(SocketType.Pub, Bind("tcp://127.0.0.1:1235"))
  val memory = ManagementFactory.getMemoryMXBean
  val os = ManagementFactory.getOperatingSystemMXBean
  val ser = SerializationExtension(context.system)

  override def preStart() {
    context.system.scheduler.schedule(1 second, 1 second, self, Tick)
  }

  override def postRestart(reason: Throwable) {
    // don't call preStart, only schedule once
  }

  def receive: Receive = {
    case Tick =>
      val currentHeap = memory.getHeapMemoryUsage
      val timestamp = System.currentTimeMillis

      // use akka SerializationExtension to convert to bytes
      val heapPayload = ser.serialize(Heap(timestamp, currentHeap.getUsed, currentHeap.getMax)).fo
      // the first frame is the topic, second is the message
      pubSocket ! ZMQMessage(Seq(Frame("health.heap"), Frame(heapPayload)))

      // use akka SerializationExtension to convert to bytes
      val loadPayload = ser.serialize(Load(timestamp, os.getSystemLoadAverage)).fold(throw _, ide
      // the first frame is the topic, second is the message
      pubSocket ! ZMQMessage(Seq(Frame("health.load"), Frame(loadPayload)))
  }
}
```

```
  system.actorOf(Props[HealthProbe], name = "health")
```

Let's add one subscriber that logs the information. It subscribes to all topics starting with `"health"`, i.e. both `Heap` and `Load` events.

```
class Logger extends Actor with ActorLogging {

  context.system.newSocket(SocketType.Sub, Listener(self), Connect("tcp://127.0.0.1:1235"), Subsc
  val ser = SerializationExtension(context.system)
  val timestampFormat = new SimpleDateFormat("HH:mm:ss.SSS")

  def receive = {
    // the first frame is the topic, second is the message
    case m: ZMQMessage if m.firstFrameAsString == "health.heap" ⇒
      ser.deserialize(m.payload(1), classOf[Heap]) match {
        case Right(Heap(timestamp, used, max)) ⇒
          log.info("Used heap {} bytes, at {}", used, timestampFormat.format(new Date(timestamp))
        case Left(e) ⇒ throw e
      }

    case m: ZMQMessage if m.firstFrameAsString == "health.load" ⇒
      ser.deserialize(m.payload(1), classOf[Load]) match {
        case Right(Load(timestamp, loadAverage)) ⇒
          log.info("Load average {}, at {}", loadAverage, timestampFormat.format(new Date(timestar
        case Left(e) ⇒ throw e
      }
  }
}

  system.actorOf(Props[Logger], name = "logger")
```

Another subscriber keep track of used heap and warns if too much heap is used. It only subscribes to `Heap` events.

```
class HeapAlerter extends Actor with ActorLogging {

  context.system.newSocket(SocketType.Sub, Listener(self), Connect("tcp://127.0.0.1:1235"), Subsc
  val ser = SerializationExtension(context.system)
  var count = 0

  def receive = {
    // the first frame is the topic, second is the message
    case m: ZMQMessage if m.firstFrameAsString == "health.heap" ⇒
      ser.deserialize(m.payload(1), classOf[Heap]) match {
        case Right(Heap(timestamp, used, max)) ⇒
          if ((used.toDouble / max) > 0.9) count += 1
          else count = 0
          if (count > 10) log.warning("Need more memory, using {} %", (100.0 * used / max))
        case Left(e) ⇒ throw e
      }
  }
}

  system.actorOf(Props[HeapAlerter], name = "alerter")
```

Router-Dealer Connection

While Pub/Sub is nice the real advantage of zeromq is that it is a "lego-box" for reliable messaging. And because there are so many integrations the multi-language support is fantastic. When you're using ZeroMQ to integrate many systems you'll probably need to build your own ZeroMQ devices. This is where the router and dealer socket types come in handy. With those socket types you can build your own reliable pub sub broker that uses TCP/IP and does publisher side filtering of events.

To create a Router socket that has a high watermark configured, you would do:

```
val highWatermarkSocket = system.newSocket(
  SocketType.Router,
  Listener(listener),
  Bind("tcp://127.0.0.1:1234"),
  HighWatermark(50000))
```

The akka-zeromq module accepts most if not all the available configuration options for a zeromq socket.

Push-Pull Connection

Akka ZeroMQ module supports `Push-Pull` connections.

You can create a `Push` connection through the:

```
def newPushSocket(socketParameters: Array[SocketOption]): ActorRef
```

You can create a `Pull` connection through the:

```
def newPullSocket(socketParameters: Array[SocketOption]): ActorRef
```

More documentation and examples will follow soon.

Rep-Req Connection

Akka ZeroMQ module supports `Rep-Req` connections.

You can create a `Rep` connection through the:

```
def newRepSocket(socketParameters: Array[SocketOption]): ActorRef
```

You can create a `Req` connection through the:

```
def newReqSocket(socketParameters: Array[SocketOption]): ActorRef
```

More documentation and examples will follow soon.

4.21 Microkernel (Scala)

The Akka Microkernel is included in the Akka download found at downloads.

To run an application with the microkernel you need to create a Bootable class that handles the startup and shutdown the application. An example is included below.

Put your application jar in the `deploy` directory to have it automatically loaded.

To start the kernel use the scripts in the `bin` directory, passing the boot classes for your application.

There is a simple example of an application setup for running with the microkernel included in the akka download. This can be run with the following command (on a unix-based system):

```
bin/akka sample.kernel.hello.HelloKernel
```

Use `Ctrl-C` to interrupt and exit the microkernel.

On a Windows machine you can also use the bin/akka.bat script.

The code for the Hello Kernel example (see the `HelloKernel` class for an example of creating a Bootable):

```
/**
 *  Copyright (C) 2009-2012 Typesafe Inc. <http://www.typesafe.com>
 */
package sample.kernel.hello

import akka.actor.{ Actor, ActorSystem, Props }
import akka.kernel.Bootable

case object Start

class HelloActor extends Actor {
  val worldActor = context.actorOf(Props[WorldActor])

  def receive = {
    case Start ⇒ worldActor ! "Hello"
    case message: String ⇒
      println("Received message '%s'" format message)
  }
}

class WorldActor extends Actor {
  def receive = {
    case message: String ⇒ sender ! (message.toUpperCase + " world!")
  }
}

class HelloKernel extends Bootable {
  val system = ActorSystem("hellokernel")

  def startup = {
    system.actorOf(Props[HelloActor]) ! Start
  }

  def shutdown = {
    system.shutdown()
  }
}
```

4.21.1 Distribution of microkernel application

To make a distribution package of the microkernel and your application the `akka-sbt-plugin` provides `AkkaKernelPlugin`. It creates the directory structure, with jar files, configuration files and start scripts.

To use the sbt plugin you define it in your `project/plugins.sbt`:

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

addSbtPlugin("com.typesafe.akka" % "akka-sbt-plugin" % "2.0.4")
```

Then you add it to the settings of your `project/Build.scala`. It is also important that you add the `akka-kernel` dependency. This is an example of a complete sbt build file:

```
import sbt._
import Keys._
import akka.sbt.AkkaKernelPlugin
import akka.sbt.AkkaKernelPlugin.{ Dist, outputDirectory, distJvmOptions}

object HelloKernelBuild extends Build {
  val Organization = "akka.sample"
  val Version      = "2.0.4"
  val ScalaVersion = "2.9.1"
```

```
  lazy val HelloKernel = Project(
    id = "hello-kernel",
    base = file("."),
    settings = defaultSettings ++ AkkaKernelPlugin.distSettings ++ Seq(
      libraryDependencies ++= Dependencies.helloKernel,
      distJvmOptions in Dist := "-Xms256M -Xmx1024M",
      outputDirectory in Dist := file("target/hello-dist")
    )
  )

  lazy val buildSettings = Defaults.defaultSettings ++ Seq(
    organization := Organization,
    version      := Version,
    scalaVersion := ScalaVersion,
    crossPaths   := false,
    organizationName := "Typesafe Inc.",
    organizationHomepage := Some(url("http://www.typesafe.com"))
  )

  lazy val defaultSettings = buildSettings ++ Seq(
    resolvers += "Typesafe Repo" at "http://repo.typesafe.com/typesafe/releases/",

    // compile options
    scalacOptions ++= Seq("-encoding", "UTF-8", "-deprecation", "-unchecked"),
    javacOptions  ++= Seq("-Xlint:unchecked", "-Xlint:deprecation")

  )
}

object Dependencies {
  import Dependency._

  val helloKernel = Seq(
    akkaKernel, akkaSlf4j, logback
  )
}

object Dependency {
  // Versions
  object V {
    val Akka      = "2.0.4"
  }

  val akkaKernel        = "com.typesafe.akka" % "akka-kernel"         % V.Akka
  val akkaSlf4j         = "com.typesafe.akka" % "akka-slf4j"          % V.Akka
  val logback           = "ch.qos.logback"    % "logback-classic"     % "1.0.0"
}
```

Run the plugin with sbt:

```
> dist
> dist:clean
```

There are several settings that can be defined:

- `outputDirectory` - destination directory of the package, default `target/dist`
- `distJvmOptions` - JVM parameters to be used in the start script
- `configSourceDirs` - Configuration files are copied from these directories, default `src/config`, `src/main/config`, `src/main/resources`
- `distMainClass` - Kernel main class to use in start script
- `libFilter` - Filter of dependency jar files

- `additionalLibs` - Additional dependency jar files

CHAPTER

FIVE

JAVA API

5.1 Actors (Java)

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.

5.1.1 Creating Actors

Since Akka enforces parental supervision every actor is supervised and (potentially) the supervisor of its children; it is advisable that you familiarize yourself with *Actor Systems* and *Supervision and Monitoring* and it may also help to read *Summary: actorOf vs. actorFor*.

Defining an Actor class

Actor in Java are implemented 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.Logging;
import akka.event.LoggingAdapter;

public class MyUntypedActor extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  public void onReceive(Object message) throws Exception {
    if (message instanceof String)
      log.info("Received String message: {}", message);
    else
      unhandled(message);
  }
}
```

Props

`Props` is a configuration class to specify options for the creation of actors. Here are some examples on how to create a `Props` instance.

```
Props props1 = new Props();
Props props2 = new Props(MyUntypedActor.class);
Props props3 = new Props(new UntypedActorFactory() {
  public UntypedActor create() {
    return new MyUntypedActor();
  }
});
Props props4 = props1.withCreator(new UntypedActorFactory() {
  public UntypedActor create() {
    return new MyUntypedActor();
  }
});
```

Creating Actors with Props

Actors are created by passing in a `Props` instance into the `actorOf` factory method.

```
ActorRef myActor = system.actorOf(new Props(MyUntypedActor.class).withDispatcher("my-dispatcher"),
```

Creating Actors with default constructor

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;
    ActorSystem system = ActorSystem.create("MySystem");
    ActorRef myActor = system.actorOf(new Props(MyUntypedActor.class), "myactor");
```

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 `UntypedActor`. 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.

In the above example the actor was created from the system. It is also possible to create actors from other actors with the actor `context`. The difference is how the supervisor hierarchy is arranged. When using the context the current actor will be supervisor of the created child actor. When using the system it will be a top level actor, that is supervised by the system (internal guardian actor).

```
public class FirstUntypedActor extends UntypedActor {
  ActorRef myActor = getContext().actorOf(new Props(MyActor.class), "myactor");
```

The name parameter is optional, but you should preferably name your actors, since that is used in log messages and for identifying actors. The name must not be empty or start with `$`. If the given name is already in use by another child to the same parent actor an *InvalidActorNameException* is thrown.

> **Warning:** Creating top-level actors with `system.actorOf` is a blocking operation, hence it may deadlock due to starvation if the default dispatcher is overloaded. To avoid problems, do not call this method from within actors or futures which run on the default dispatcher.

Actors are automatically started asynchronously when created. When you create the `UntypedActor` then it will automatically call the `preStart` callback method on the `UntypedActor` class. This is an excellent place to add initialization code for the actor.

```
@Override
public void preStart() {
  ... // initialization code
}
```

Creating Actors with non-default constructor

If your UntypedActor has a constructor that takes parameters then you can't create it using 'actorOf(new Props(clazz))'. Then you can instead pass in 'new Props(new UntypedActorFactory() {..})' in which you can create the Actor in any way you like.

Here is an example:

```
// allows passing in arguments to the MyActor constructor
ActorRef myActor = system.actorOf(new Props(new UntypedActorFactory() {
  public UntypedActor create() {
    return new MyActor("...");
  }
}), "myactor");
```

This way of creating the Actor is also great for integrating with Dependency Injection (DI) frameworks like Guice or Spring.

Warning: You might be tempted at times to offer an `UntypedActor` factory which always returns the same instance, e.g. by using a static field. This is not supported, as it goes against the meaning of an actor restart, which is described here: *What Restarting Means*.

5.1.2 UntypedActor API

The `UntypedActor` class defines only one abstract method, the above mentioned `onReceive(Object message)`, which implements the behavior of the actor.

If the current actor behavior does not match a received message, `unhandled` is called, which by default publishes a `new akka.actor.UnhandledMessage(message, sender, recipient)` on the actor system's event stream.

In addition, it offers:

- `getSelf()` reference to the `ActorRef` of the actor
- `getSender()` reference sender Actor of the last received message, typically used as described in *Reply to messages*
- `supervisorStrategy()` user overridable definition the strategy to use for supervising child actors
- `getContext()` exposes contextual information for the actor and the current message, such as:
  - factory methods to create child actors (`actorOf`)
  - system that the actor belongs to
  - parent supervisor
  - supervised children
  - lifecycle monitoring
  - hotswap behavior stack as described in *HotSwap*

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

```
public void preStart() {
}

public void preRestart(Throwable reason, Option<Object> message) {
  for (ActorRef each : getContext().getChildren())
    getContext().stop(each);
  postStop();
}
```

```
public void postRestart(Throwable reason) {
  preStart();
}

public void postStop() {
}
```

The implementations shown above are the defaults provided by the `UntypedActor` class.

Lifecycle Monitoring aka DeathWatch

In order to be notified when another actor terminates (i.e. stops permanently, not temporary failure and restart), an actor may register itself for reception of the `Terminated` message dispatched by the other actor upon termination (see Stopping Actors). This service is provided by the `DeathWatch` component of the actor system.

Registering a monitor is easy (see fourth line, the rest is for demonstrating the whole functionality):

```
public static class WatchActor extends UntypedActor {
  final ActorRef child = this.getContext().actorOf(Props.empty(), "child");
  {
    this.getContext().watch(child); // <-- this is the only call needed for registration
  }
  ActorRef lastSender = getContext().system().deadLetters();

  @Override
  public void onReceive(Object message) {
    if (message.equals("kill")) {
      getContext().stop(child);
      lastSender = getSender();
    } else if (message instanceof Terminated) {
      final Terminated t = (Terminated) message;
      if (t.getActor() == child) {
        lastSender.tell("finished");
      }
    } else {
      unhandled(message);
    }
  }
}
```

It should be noted that the `Terminated` message is generated independent of the order in which registration and termination occur. Registering multiple times does not necessarily lead to multiple messages being generated, but there is no guarantee that only exactly one such message is received: if termination of the watched actor has generated and queued the message, and another registration is done before this message has been processed, then a second message will be queued, because registering for monitoring of an already terminated actor leads to the immediate generation of the `Terminated` message.

It is also possible to deregister from watching another actor's liveliness using `context.unwatch(target)`, but obviously this cannot guarantee non-reception of the `Terminated` message because that may already have been queued.

Start Hook

Right after starting the actor, its `preStart` method is invoked.

```
@Override
public void preStart() {
  // registering with other actors
  someService.tell(Register(getSelf());
}
```

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors will be restarted in case an exception is thrown while processing a message. This restart involves 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. By default it stops all children and calls `postStop`.
2. The initial factory from the `actorOf` call is used to produce the fresh instance.
3. The new actor's `postRestart` method is invoked with the exception which caused the restart. By default the `preStart` is called, just as in the normal start-up case.

An actor restart replaces only the actual actor object; the contents of the mailbox is unaffected by the restart, so processing of messages will resume after the `postRestart` hook returns. The message that triggered the exception will not be received again. 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. messages sent to a stopped actor will be redirected to the `deadLetters` of the `ActorSystem`.

5.1.3 Identifying Actors

As described in *Actor References, Paths and Addresses*, each actor has a unique logical path, which is obtained by following the chain of actors from child to parent until reaching the root of the actor system, and it has a physical path, which may differ if the supervision chain includes any remote supervisors. These paths are used by the system to look up actors, e.g. when a remote message is received and the recipient is searched, but they are also useful more directly: actors may look up other actors by specifying absolute or relative paths—logical or physical—and receive back an `ActorRef` with the result:

```
getContext().actorFor("/user/serviceA/aggregator") // will look up this absolute path
getContext().actorFor("../joe")                    // will look up sibling beneath same supervisor
```

The supplied path is parsed as a `java.net.URI`, which basically means that it is split on `/` into path elements. If the path starts with `/`, it is absolute and the look-up starts at the root guardian (which is the parent of `"/user"`); otherwise it starts at the current actor. If a path element equals `..`, the look-up will take a step "up" towards the supervisor of the currently traversed actor, otherwise it will step "down" to the named child. It should be noted that the `..` in actor paths here always means the logical structure, i.e. the supervisor.

If the path being looked up does not exist, a special actor reference is returned which behaves like the actor system's dead letter queue but retains its identity (i.e. the path which was looked up).

Remote actor addresses may also be looked up, if remoting is enabled:

```
getContext().actorFor("akka://app@otherhost:1234/user/serviceB")
```

These look-ups return a (possibly remote) actor reference immediately, so you will have to send to it and await a reply in order to verify that `serviceB` is actually reachable and running. An example demonstrating actor look-up is given in *Remote Lookup*.

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.

Here is an example of an immutable message:

```
public class ImmutableMessage {
  private final int sequenceNumber;
  private final List<String> values;

  public ImmutableMessage(int sequenceNumber, List<String> values) {
    this.sequenceNumber = sequenceNumber;
    this.values = Collections.unmodifiableList(new ArrayList<String>(values));
  }

  public int getSequenceNumber() {
    return sequenceNumber;
  }

  public List<String> getValues() {
    return values;
  }
}
```

5.1.5 Send messages

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

- `tell` means "fire-and-forget", e.g. send a message asynchronously and return immediately.
- `ask` sends a message asynchronously and returns a `Future` representing a possible reply.

Message ordering is guaranteed on a per-sender basis.

Note: There are performance implications of using `ask` since something needs to keep track of when it times out, there needs to be something that bridges a `Promise` into an `ActorRef` and it also needs to be reachable through remoting. So always prefer `tell` for performance, and only `ask` if you must.

In all these methods you have the option of passing along your own `ActorRef`. 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.

Tell: 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.tell("Hello");
```

Or with the sender reference passed along with the message and available to the receiving Actor in its `getSender: ActorRef` member field. The target actor can use this to reply to the original sender, by using `getSender().tell(replyMsg)`.

```
actor.tell("Hello", getSelf());
```

If invoked without the sender parameter the sender will be `deadLetters` actor reference in the target actor.

Ask: Send-And-Receive-Future

The `ask` pattern involves actors as well as futures, hence it is offered as a use pattern rather than a method on `ActorRef`:

```
import static akka.pattern.Patterns.ask;
import static akka.pattern.Patterns.pipe;
import akka.dispatch.Future;
import akka.dispatch.Futures;
import akka.util.Duration;
import akka.util.Timeout;
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;
```

```
final Timeout t = new Timeout(Duration.create(5, TimeUnit.SECONDS));

final ArrayList<Future<Object>> futures = new ArrayList<Future<Object>>();
futures.add(ask(actorA, "request", 1000)); // using 1000ms timeout
futures.add(ask(actorB, "request", t)); // using timeout from above

final Future<Iterable<Object>> aggregate = Futures.sequence(futures, system.dispatcher());

final Future<Result> transformed = aggregate.map(new Mapper<Iterable<Object>, Result>() {
  public Result apply(Iterable<Object> coll) {
    final Iterator<Object> it = coll.iterator();
    final String s = (String) it.next();
    final int x = (Integer) it.next();
    return new Result(x, s);
  }
});

pipe(transformed).to(actorC);
```

This example demonstrates `ask` together with the `pipe` pattern on futures, because this is likely to be a common combination. Please note that all of the above is completely non-blocking and asynchronous: `ask` produces a `Future`, two of which are composed into a new future using the `Futures.sequence` and `map` methods and then `pipe` installs an `onComplete`-handler on the future to effect the submission of the aggregated `Result` to another actor.

Using `ask` will send a message to the receiving Actor as with `tell`, and the receiving actor must reply with `getSender().tell(reply)` in order to complete the returned `Future` with a value. The `ask` operation involves creating an internal actor for handling this reply, which needs to have a timeout after which it is destroyed in order not to leak resources; see more below.

To complete the future with an exception you need send a Failure message to the sender. This is *not done automatically* when an actor throws an exception while processing a message.

```
try {
  String result = operation();
  getSender().tell(result);
} catch (Exception e) {
  getSender().tell(new akka.actor.Status.Failure(e));
  throw e;
}
```

If the actor does not complete the future, it will expire after the timeout period, specified as parameter to the `ask` method; this will complete the `Future` with an `AskTimeoutException`.

See *Futures (Java)* for more information on how to await or query a future.

The `onComplete`, `onResult` methods of the `Future` can be used to register a callback to get a notification when the Future completes. Gives you a way to avoid blocking.

Warning: When using future callbacks, inside actors you need to carefully avoid closing over the containing actor's reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time. See also: *Actors and shared mutable state*

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 context variable as well.

```
myActor.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:

```
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

public class MyUntypedActor extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  public void onReceive(Object message) throws Exception {
    if (message instanceof String)
      log.info("Received String message: {}", message);
    else
      unhandled(message);
  }
}
```

An alternative to using if-instanceof checks is to use Apache Commons MethodUtils to invoke a named method whose parameter type matches the message type.

5.1.7 Reply to messages

If you want to have a handle for replying to a message, you can use `getSender()`, which gives you an ActorRef. You can reply by sending to that ActorRef with `getSender().tell(replyMsg)`. You can also store the ActorRef for replying later, or passing on to other actors. If there is no sender (a message was sent without an actor or future context) then the sender defaults to a 'dead-letter' actor ref.

```
public void onReceive(Object request) {
  String result = process(request);
  getSender().tell(result);       // will have dead-letter actor as default
}
```

5.1.8 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 handing for the ReceiveTimeout message.

```
import akka.actor.Actors;
import akka.actor.ReceiveTimeout;
import akka.actor.UntypedActor;
import akka.util.Duration;
```

```
public class MyReceivedTimeoutUntypedActor extends UntypedActor {

  public MyReceivedTimeoutUntypedActor() {
    getContext().setReceiveTimeout(Duration.parse("30 seconds"));
  }

  public void onReceive(Object message) {
    if (message.equals("Hello")) {
      getSender().tell("Hello world");
    } else if (message == Actors.receiveTimeout()) {
      throw new RuntimeException("received timeout");
    } else {
      unhandled(message);
    }
  }
}
```

5.1.9 Stopping actors

Actors are stopped by invoking the `stop` method of a `ActorRefFactory`, i.e. `ActorContext` or `ActorSystem`. Typically the context is used for stopping child actors and the system for stopping top level actors. The actual termination of the actor is performed asynchronously, i.e. `stop` may return before the actor is stopped.

Processing of the current message, if any, will continue before the actor is stopped, but additional messages in the mailbox will not be processed. By default these messages are sent to the `deadLetters` of the `ActorSystem`, but that depends on the mailbox implementation.

Termination of an actor proceeds in two steps: first the actor suspends its mailbox processing and sends a stop command to all its children, then it keeps processing the termination messages from its children until the last one is gone, finally terminating itself (invoking `postStop`, dumping mailbox, publishing `Terminated` on the *DeathWatch*, telling its supervisor). This procedure ensures that actor system sub-trees terminate in an orderly fashion, propagating the stop command to the leaves and collecting their confirmation back to the stopped supervisor. If one of the actors does not respond (i.e. processing a message for extended periods of time and therefore not receiving the stop command), this whole process will be stuck.

Upon `ActorSystem.shutdown`, the system guardian actors will be stopped, and the aforementioned process will ensure proper termination of the whole system.

The `postStop` hook is invoked after an actor is fully stopped. This enables cleaning up of resources:

```
@Override
public void postStop() {
  // close some file or database connection
}
```

Note: Since stopping an actor is asynchronous, you cannot immediately reuse the name of the child you just stopped; this will result in an `InvalidActorNameException`. Instead, `watch` the terminating actor and create its replacement in response to the `Terminated` message which will eventually arrive.

PoisonPill

You can also send an actor the `akka.actor.PoisonPill` message, which will stop the actor when the message is processed. `PoisonPill` is enqueued as ordinary messages and will be handled after messages that were already queued in the mailbox.

Use it like this:

```
import static akka.actor.Actors.*;
    myActor.tell(poisonPill());
```

Graceful Stop

`gracefulStop` is useful if you need to wait for termination or compose ordered termination of several actors:

```
import static akka.pattern.Patterns.gracefulStop;
import akka.dispatch.Future;
import akka.dispatch.Await;
import akka.util.Duration;
import akka.actor.ActorTimeoutException;

    try {
      Future<Boolean> stopped = gracefulStop(actorRef, Duration.create(5, TimeUnit.SECONDS), syste
      Await.result(stopped, Duration.create(6, TimeUnit.SECONDS));
      // the actor has been stopped
    } catch (ActorTimeoutException e) {
      // the actor wasn't stopped within 5 seconds
    }
```

When `gracefulStop()` returns successfully, the actor's `postStop()` hook will have been executed: there exists a happens-before edge between the end of `postStop()` and the return of `gracefulStop()`.

Warning: Keep in mind that an actor stopping and its name being deregistered are separate events which happen asynchronously from each other. Therefore it may be that you will find the name still in use after `gracefulStop()` returned. In order to guarantee proper deregistration, only reuse names from within a supervisor you control and only in response to a `Terminated` message, i.e. not for top-level actors.

5.1.10 HotSwap

Upgrade

Akka supports hotswapping the Actor's message loop (e.g. its implementation) at runtime. Use the `getContext().become` method from within the Actor. The hotswapped code is kept in a Stack which can be pushed and popped.

Warning: Please note that the actor will revert to its original behavior when restarted by its Supervisor.

To hotswap the Actor using `getContext().become`:

```
import akka.japi.Procedure;
  public static class HotSwapActor extends UntypedActor {

    Procedure<Object> angry = new Procedure<Object>() {
      @Override
      public void apply(Object message) {
        if (message.equals("bar")) {
          getSender().tell("I am already angry?");
        } else if (message.equals("foo")) {
          getContext().become(happy);
        }
      }
    };

    Procedure<Object> happy = new Procedure<Object>() {
      @Override
      public void apply(Object message) {
```

```
        if (message.equals("bar")) {
          getSender().tell("I am already happy :-)");
        } else if (message.equals("foo")) {
          getContext().become(angry);
        }
      }
    };

    public void onReceive(Object message) {
      if (message.equals("bar")) {
        getContext().become(angry);
      } else if (message.equals("foo")) {
        getContext().become(happy);
      } else {
        unhandled(message);
      }
    }
  }
```

The `become` method is useful for many different things, such as to implement a Finite State Machine (FSM).

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

```
public class UntypedActorSwapper {

  public static class Swap {
    public static Swap SWAP = new Swap();

    private Swap() {
    }
  }

  public static class Swapper extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    public void onReceive(Object message) {
      if (message == SWAP) {
        log.info("Hi");
        getContext().become(new Procedure<Object>() {
          @Override
          public void apply(Object message) {
            log.info("Ho");
            getContext().unbecome(); // resets the latest 'become' (just for fun)
          }
        });
      } else {
        unhandled(message);
      }
    }
  }

  public static void main(String... args) {
    ActorSystem system = ActorSystem.create("MySystem");
    ActorRef swap = system.actorOf(new Props(Swapper.class));
    swap.tell(SWAP); // logs Hi
    swap.tell(SWAP); // logs Ho
    swap.tell(SWAP); // logs Hi
    swap.tell(SWAP); // logs Ho
    swap.tell(SWAP); // logs Hi
    swap.tell(SWAP); // logs Ho
  }

}
```

Downgrade

Since the hotswapped code is pushed to a Stack you can downgrade the code as well. Use the `getContext().unbecome` method from within the Actor.

```
public void onReceive(Object message) {
  if (message.equals("revert")) getContext().unbecome();
}
```

5.1.11 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:

```
import static akka.actor.Actors.*;
    victim.tell(kill());
```

5.1.12 Actors and exceptions

It can happen that while a message is being processed by an actor, that some kind of exception is thrown, e.g. a database exception.

What happens to the Message

If an exception is thrown while a message is being processed (so taken of his mailbox and handed over the the receive), then this message will be lost. It is important to understand that it is not put back on the mailbox. So if you want to retry processing of a message, you need to deal with it yourself by catching the exception and retry your flow. Make sure that you put a bound on the number of retries since you don't want a system to livelock (so consuming a lot of cpu cycles without making progress).

What happens to the mailbox

If an exception is thrown while a message is being processed, nothing happens to the mailbox. If the actor is restarted, the same mailbox will be there. So all messages on that mailbox, will be there as well.

What happens to the actor

If an exception is thrown, the actor instance is discarded and a new instance is created. This new instance will now be used in the actor references to this actor (so this is done invisible to the developer). Note that this means that current state of the failing actor instance is lost if you don't store and restore it in `preRestart` and `postRestart` callbacks.

5.2 Typed Actors (Java)

Akka Typed Actors is an implementation of the Active Objects pattern. Essentially turning method invocations into asynchronous dispatch instead of synchronous that has been the default way since Smalltalk came out.

Typed Actors consist of 2 "parts", a public interface and an implementation, and if you've done any work in "enterprise" Java, this will be very familiar to you. As with normal Actors you have an external API (the public interface instance) that will delegate methodcalls asynchronously to a private instance of the implementation.

The advantage of Typed Actors vs. Actors is that with TypedActors you have a static contract, and don't need to define your own messages, the downside is that it places some limitations on what you can do and what you can't, i.e. you can't use become/unbecome.

Typed Actors are implemented using JDK Proxies which provide a pretty easy-worked API to intercept method calls.

Note: Just as with regular Akka Untyped Actors, Typed Actors process one call at a time.

5.2.1 When to use Typed Actors

Typed actors are nice for bridging between actor systems (the "inside") and non-actor code (the "outside"), because they allow you to write normal OO-looking code on the outside. Think of them like doors: their practicality lies in interfacing between private sphere and the public, but you don't want that many doors inside your house, do you? For a longer discussion see this blog post.

A bit more background: TypedActors can very easily be abused as RPC, and that is an abstraction which is well-known to be leaky. Hence TypedActors are not what we think of first when we talk about making highly scalable concurrent software easier to write correctly. They have their niche, use them sparingly.

5.2.2 The tools of the trade

Before we create our first Typed Actor we should first go through the tools that we have at our disposal, it's located in `akka.actor.TypedActor`.

```
//Returns the Typed Actor Extension
TypedActorExtension extension =
        TypedActor.get(system); //system is an instance of ActorSystem

//Returns whether the reference is a Typed Actor Proxy or not
TypedActor.get(system).isTypedActor(someReference);

//Returns the backing Akka Actor behind an external Typed Actor Proxy
TypedActor.get(system).getActorRefFor(someReference);

//Returns the current ActorContext,
// method only valid within methods of a TypedActor implementation
ActorContext context = TypedActor.context();

//Returns the external proxy of the current Typed Actor,
// method only valid within methods of a TypedActor implementation
Squarer sq = TypedActor.<Squarer>self();

//Returns a contextual instance of the Typed Actor Extension
//this means that if you create other Typed Actors with this,
//they will become children to the current Typed Actor.
TypedActor.get(TypedActor.context());
```

Warning: Same as not exposing `this` of an Akka Actor, it's important not to expose `this` of a Typed Actor, instead you should pass the external proxy reference, which is obtained from within your Typed Actor as `TypedActor.self()`, this is your external identity, as the `ActorRef` is the external identity of an Akka Actor.

5.2.3 Creating Typed Actors

To create a Typed Actor you need to have one or more interfaces, and one implementation.

Our example interface:

```
import akka.dispatch.*;
import akka.actor.*;
import akka.japi.*;
import akka.util.Duration;
import java.util.concurrent.TimeUnit;

    public static interface Squarer {
      // typed actor iface methods ...
    }
```

Our example implementation of that interface:

```
import akka.dispatch.*;
import akka.actor.*;
import akka.japi.*;
import akka.util.Duration;
import java.util.concurrent.TimeUnit;

    static class SquarerImpl implements Squarer {
      private String name;

      public SquarerImpl() {
          this.name = "default";
      }

      public SquarerImpl(String name) {
        this.name = name;
      }

      // typed actor impl methods ...
    }
```

The most trivial way of creating a Typed Actor instance of our `Squarer`:

```
Squarer mySquarer =
  TypedActor.get(system).typedActorOf(new TypedProps<SquarerImpl>(Squarer.class, SquarerImpl.class
```

First type is the type of the proxy, the second type is the type of the implementation. If you need to call a specific constructor you do it like this:

```
Squarer otherSquarer =
  TypedActor.get(system).typedActorOf(new TypedProps<SquarerImpl>(Squarer.class,
    new Creator<SquarerImpl>() {
      public SquarerImpl create() { return new SquarerImpl("foo"); }
    }),
    "name");
```

Since you supply a `Props`, you can specify which dispatcher to use, what the default timeout should be used and more. Now, our `Squarer` doesn't have any methods, so we'd better add those.

```
import akka.dispatch.*;
import akka.actor.*;
import akka.japi.*;
import akka.util.Duration;
import java.util.concurrent.TimeUnit;

    public static interface Squarer {
      void squareDontCare(int i); //fire-forget

      Future<Integer> square(int i); //non-blocking send-request-reply

      Option<Integer> squareNowPlease(int i);//blocking send-request-reply
```

```
    int squareNow(int i); //blocking send-request-reply
  }
```

Alright, now we've got some methods we can call, but we need to implement those in `SquarerImpl`.

```
import akka.dispatch.*;
import akka.actor.*;
import akka.japi.*;
import akka.util.Duration;
import java.util.concurrent.TimeUnit;

  static class SquarerImpl implements Squarer {
    private String name;

    public SquarerImpl() {
        this.name = "default";
    }

    public SquarerImpl(String name) {
      this.name = name;
    }


    public void squareDontCare(int i) {
        int sq = i * i; //Nobody cares :(
    }

    public Future<Integer> square(int i) {
        return Futures.successful(i * i, TypedActor.dispatcher());
    }

    public Option<Integer> squareNowPlease(int i) {
        return Option.some(i * i);
    }

    public int squareNow(int i) {
        return i * i;
    }
  }
```

Excellent, now we have an interface and an implementation of that interface, and we know how to create a Typed Actor from that, so let's look at calling these methods.

5.2.4 Method dispatch semantics

Methods returning:

- `void` will be dispatched with `fire-and-forget` semantics, exactly like `ActorRef.tell`
- `akka.dispatch.Future<?>` will use `send-request-reply` semantics, exactly like `ActorRef.ask`
- `scala.Option<?>` or `akka.japi.Option<?>` will use `send-request-reply` semantics, but *will* block to wait for an answer, and return None if no answer was produced within the timout, or scala.Some/akka.japi.Some containing the result otherwise. Any exception that was thrown during this call will be rethrown.
- Any other type of value will use `send-request-reply` semantics, but *will* block to wait for an answer, throwing `java.util.concurrent.TimeoutException` if there was a timeout or rethrow any exception that was thrown during this call.

5.2.5 Messages and immutability

While Akka cannot enforce that the parameters to the methods of your Typed Actors are immutable, we *strongly* recommend that parameters passed are immutable.

One-way message send

```
mySquarer.squareDontCare(10);
```

As simple as that! The method will be executed on another thread; asynchronously.

Request-reply message send

```
Option<Integer> oSquare = mySquarer.squareNowPlease(10); //Option[Int]
```

This will block for as long as the timeout that was set in the `Props` of the Typed Actor, if needed. It will return `None` if a timeout occurs.

```
int iSquare = mySquarer.squareNow(10); //Int
```

This will block for as long as the timeout that was set in the `Props` of the Typed Actor, if needed. It will throw a ``java.util.concurrent.TimeoutException if a timeout occurs.

Request-reply-with-future message send

```
Future<Integer> fSquare = mySquarer.square(10); //A Future[Int]
```

This call is asynchronous, and the Future returned can be used for asynchronous composition.

5.2.6 Stopping Typed Actors

Since Akka's Typed Actors are backed by Akka Actors they must be stopped when they aren't needed anymore.

```
TypedActor.get(system).stop(mySquarer);
```

This asynchronously stops the Typed Actor associated with the specified proxy ASAP.

```
TypedActor.get(system).poisonPill(otherSquarer);
```

This asynchronously stops the Typed Actor associated with the specified proxy after it's done with all calls that were made prior to this call.

5.2.7 Typed Actor Hierarchies

Since you can obtain a contextual Typed Actor Extension by passing in an `ActorContext` you can create child Typed Actors by invoking `typedActorOf(..)` on that.

This also works for creating child Typed Actors in regular Akka Actors.

5.2.8 Supervisor Strategy

By having your Typed Actor implementation class implement `TypedActor.Supervisor` you can define the strategy to use for supervising child actors, as described in *Supervision and Monitoring* and *Fault Tolerance (Java)*.

5.2.9 Receive arbitrary messages

If your implementation class of your TypedActor extends `akka.actor.TypedActor.Receiver`, all messages that are not `MethodCall``s will be passed into the ``onReceive`-method.

This allows you to react to DeathWatch `Terminated`-messages and other types of messages, e.g. when interfacing with untyped actors.

5.2.10 Lifecycle callbacks

By having your Typed Actor implementation class implement any and all of the following:

- `TypedActor.PreStart`
- `TypedActor.PostStop`
- `TypedActor.PreRestart`
- `TypedActor.PostRestart`

You can hook into the lifecycle of your Typed Actor.

5.2.11 Proxying

You can use the `typedActorOf` that takes a TypedProps and an ActorRef to proxy the given ActorRef as a TypedActor. This is usable if you want to communicate remotely with TypedActors on other machines, just look them up with `actorFor` and pass the `ActorRef` to `typedActorOf`.

5.2.12 Lookup & Remoting

Since `TypedActors` are backed by `Akka Actors`, you can use `actorFor` together with `typedActorOf` to proxy `ActorRefs` potentially residing on remote nodes.

```
Squarer typedActor =
  TypedActor.get(system).
    typedActorOf(
      new TypedProps<Squarer>(Squarer.class),
      system.actorFor("akka://SomeSystem@somehost:2552/user/some/foobar")
    );
//Use "typedActor" as a FooBar
```

5.3 Logging (Java)

5.3.1 How to Log

Create a `LoggingAdapter` and use the `error`, `warning`, `info`, or `debug` methods, as illustrated in this example:

```
import akka.event.Logging;
import akka.event.LoggingAdapter;

  class MyActor extends UntypedActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    @Override
    public void preStart() {
      log.debug("Starting");
    }
```

```
    @Override
    public void preRestart(Throwable reason, Option<Object> message) {
      log.error(reason, "Restarting due to [{}] when processing [{}]", reason.getMessage(),
          message.isDefined() ? message.get() : "");
    }

    public void onReceive(Object message) {
      if (message.equals("test")) {
        log.info("Received test");
      } else {
        log.warning("Received unknown message: {}", message);
      }
    }
  }
```

The first parameter to `Logging.getLogger` could also be any `LoggingBus`, specifically `system.eventStream()`; in the demonstrated case, the actor system's address is included in the `akkaSource` representation of the log source (see Logging Thread and Akka Source in MDC) while in the second case this is not automatically done. The second parameter to `Logging.getLogger` is the source of this logging channel. The source object is translated to a String according to the following rules:

- if it is an Actor or ActorRef, its path is used
- in case of a String it is used as is
- in case of a class an approximation of its simpleName
- and in all other cases the simpleName of its class

The log message may contain argument placeholders `{}`, which will be substituted if the log level is enabled. Giving more arguments as there are placeholders results in a warning being appended to the log statement (i.e. on the same line with the same severity). You may pass a Java array as the only substitution argument to have its elements be treated individually:

```
final Object[] args = new Object[] { "The", "brown", "fox", "jumps", 42 };
system.log().debug("five parameters: {}, {}, {}, {}, {}", args);
```

The Java `Class` of the log source is also included in the generated `LogEvent`. In case of a simple string this is replaced with a "marker" class `akka.event.DummyClassForStringSources` in order to allow special treatment of this case, e.g. in the SLF4J event listener which will then use the string instead of the class' name for looking up the logger instance to use.

Auxiliary logging options

Akka has a couple of configuration options for very low level debugging, that makes most sense in for developers and not for operations.

You almost definitely need to have logging set to DEBUG to use any of the options below:

```
akka {
  loglevel = DEBUG
}
```

This config option is very good if you want to know what config settings are loaded by Akka:

```
akka {
  # Log the complete configuration at INFO level when the actor system is started.
  # This is useful when you are uncertain of what configuration is used.
  log-config-on-start = on
}
```

If you want very detailed logging of all automatically received messages that are processed by Actors:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of all AutoReceiveMessages (Kill, PoisonPill and the like)
      autoreceive = on
    }
  }
}
```

If you want very detailed logging of all lifecycle changes of Actors (restarts, deaths etc):

```
akka {
  actor {
    debug {
      # enable DEBUG logging of actor lifecycle changes
      lifecycle = on
    }
  }
}
```

If you want very detailed logging of all events, transitions and timers of FSM Actors that extend LoggingFSM:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of all LoggingFSMs for events, transitions and timers
      fsm = on
    }
  }
}
```

If you want to monitor subscriptions (subscribe/unsubscribe) on the ActorSystem.eventStream:

```
akka {
  actor {
    debug {
      # enable DEBUG logging of subscription changes on the eventStream
      event-stream = on
    }
  }
}
```

Auxiliary remote logging options

If you want to see all messages that are sent through remoting at DEBUG log level: (This is logged as they are sent by the transport layer, not by the Actor)

```
akka {
  remote {
    # If this is "on", Akka will log all outbound messages at DEBUG level, if off then they are no
    log-sent-messages = on
  }
}
```

If you want to see all messages that are received through remoting at DEBUG log level: (This is logged as they are received by the transport layer, not by any Actor)

```
akka {
  remote {
    # If this is "on", Akka will log all inbound messages at DEBUG level, if off then they are not
    log-received-messages = on
  }
}
```

Also see the logging options for TestKit: *Tracing Actor Invocations*.

5.3.2 Event Handler

Logging is performed asynchronously through an event bus. You can configure which event handlers that should subscribe to the logging events. That is done using the 'event-handlers' element in the *Configuration*. Here you can also define the log level.

```
akka {
  # Event handlers to register at boot time (Logging$DefaultLogger logs to STDOUT)
  event-handlers = ["akka.event.Logging$DefaultLogger"]
  # Options: ERROR, WARNING, INFO, DEBUG
  loglevel = "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:

```
import akka.event.Logging;
import akka.event.LoggingAdapter;

import akka.event.Logging.InitializeLogger;
import akka.event.Logging.Error;
import akka.event.Logging.Warning;
import akka.event.Logging.Info;
import akka.event.Logging.Debug;

  class MyEventListener extends UntypedActor {
    public void onReceive(Object message) {
      if (message instanceof InitializeLogger) {
        getSender().tell(Logging.loggerInitialized());
      } else if (message instanceof Error) {
        // ...
      } else if (message instanceof Warning) {
        // ...
      } else if (message instanceof Info) {
        // ...
      } else if (message instanceof Debug) {
        // ...
      }
    }
  }
```

5.3.3 SLF4J

Akka provides an event handler for SL4FJ. 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 Logback:

```
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>1.0.0</version>
      <scope>runtime</scope>
    </dependency>
```

You need to enable the Slf4jEventHandler in the 'event-handlers' element in the *Configuration*. Here you can also define the log level of the event bus. More fine grained log levels can be defined in the configuration of the SLF4J backend (e.g. logback.xml).

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

The SLF4J logger selected for each log event is chosen based on the `Class` of the log source specified when creating the `LoggingAdapter`, unless that was given directly as a string in which case that string is used (i.e. `LoggerFactory.getLogger(Class c)` is used in the first case and `LoggerFactory.getLogger(String s)` in the second).

Note: Beware that the the actor system's name is appended to a `String` log source if the LoggingAdapter was created giving an `ActorSystem` to the factory. If this is not intended, give a `LoggingBus` instead as shown below:

```
final LoggingAdapter log = Logging.getLogger(system.eventStream(), "my.nice.string");
```

Logging Thread and Akka Source in MDC

Since the logging is done asynchronously the thread in which the logging was performed is captured in Mapped Diagnostic Context (MDC) with attribute name `sourceThread`. With Logback the thread name is available with `%X{sourceThread}` specifier within the pattern layout configuration:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%date{ISO8601} %-5level %logger{36} %X{sourceThread} - %msg%n</pattern>
  </encoder>
</appender>
```

Note: It will probably be a good idea to use the `sourceThread` MDC value also in non-Akka parts of the application in order to have this property consistently available in the logs.

Another helpful facility is that Akka captures the actor's address when instantiating a logger within it, meaning that the full instance identification is available for associating log messages e.g. with members of a router. This information is available in the MDC with attribute name `akkaSource`:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%date{ISO8601} %-5level %logger{36} %X{akkaSource} - %msg%n</pattern>
  </encoder>
</appender>
```

For more details on what this attribute contains—also for non-actors—please see How to Log.

5.4 Event Bus (Java)

Originally conceived as a way to send messages to groups of actors, the `EventBus` has been generalized into a set of composable traits implementing a simple interface:

- `public boolean subscribe(S subscriber, C classifier)` subscribes the given subscriber to events with the given classifier
- `public boolean unsubscribe(S subscriber, C classifier)` undoes a specific subscription
- `public void unsubscribe(S subscriber)` undoes all subscriptions for the given subscriber
- `public void publish(E event)` publishes an event, which first is classified according to the specific bus (see Classifiers) and then published to all subscribers for the obtained classifier

This mechanism is used in different places within Akka, e.g. the *DeathWatch* and the Event Stream. Implementations can make use of the specific building blocks presented below.

An event bus must define the following three abstract types:

- `E` is the type of all events published on that bus
- `S` is the type of subscribers allowed to register on that event bus
- `C` defines the classifier to be used in selecting subscribers for dispatching events

The traits below are still generic in these types, but they need to be defined for any concrete implementation.

5.4.1 Classifiers

The classifiers presented here are part of the Akka distribution, but rolling your own in case you do not find a perfect match is not difficult, check the implementation of the existing ones on github.

Lookup Classification

The simplest classification is just to extract an arbitrary classifier from each event and maintaining a set of subscribers for each possible classifier. This can be compared to tuning in on a radio station. The abstract class `LookupEventBus` is still generic in that it abstracts over how to compare subscribers and how exactly to classify. The necessary methods to be implemented are the following:

- `public C classify(E event)` is used for extracting the classifier from the incoming events.
- `public int compareSubscribers(S a, S b)` must define a partial order over the subscribers, expressed as expected from `java.lang.Comparable.compare`.
- `public void publish(E event, S subscriber)` will be invoked for each event for all subscribers which registered themselves for the event's classifier.
- `public int mapSize` determines the initial size of the index data structure used internally (i.e. the expected number of different classifiers).

This classifier is efficient in case no subscribers exist for a particular event.

Subchannel Classification

If classifiers form a hierarchy and it is desired that subscription be possible not only at the leaf nodes, this classification may be just the right one. It can be compared to tuning in on (possibly multiple) radio channels by genre. This classification has been developed for the case where the classifier is just the JVM class of the event and subscribers may be interested in subscribing to all subclasses of a certain class, but it may be used with any classifier hierarchy. The abstract members needed by this classifier are

- `public Subclassification[C] subclassification` provides an object providing `isEqual(a: Classifier, b: Classifier)` and `isSubclass(a: Classifier, b: Classifier)` to be consumed by the other methods of this classifier; this method is called on various occasions, it should be implemented so that it always returns the same object for performance reasons.
- `public C classify(E event)` is used for extracting the classifier from the incoming events.
- `public void publish(E event, S subscriber)` will be invoked for each event for all subscribers which registered themselves for the event's classifier.

This classifier is also efficient in case no subscribers are found for an event, but it uses conventional locking to synchronize an internal classifier cache, hence it is not well-suited to use cases in which subscriptions change with very high frequency (keep in mind that "opening" a classifier by sending the first message will also have to re-check all previous subscriptions).

Scanning Classification

The previous classifier was built for multi-classifier subscriptions which are strictly hierarchical, this classifier is useful if there are overlapping classifiers which cover various parts of the event space without forming a hierarchy. It can be compared to tuning in on (possibly multiple) radio stations by geographical reachability (for old-school radio-wave transmission). The abstract members for this classifier are:

- `public int compareClassifiers(C a, C b)` is needed for determining matching classifiers and storing them in an ordered collection.
- `public int compareSubscribers(S a, S b)` is needed for storing subscribers in an ordered collection.
- `public boolean matches(C classifier, E event)` determines whether a given classifier shall match a given event; it is invoked for each subscription for all received events, hence the name of the classifier.
- `public void publish(E event, S subscriber)` will be invoked for each event for all subscribers which registered themselves for a classifier matching this event.

This classifier takes always a time which is proportional to the number of subscriptions, independent of how many actually match.

Actor Classification

This classification has been developed specifically for implementing *DeathWatch*: subscribers as well as classifiers are of type `ActorRef`. The abstract members are

- `public ActorRef classify(E event)` is used for extracting the classifier from the incoming events.
- `public int mapSize` determines the initial size of the index data structure used internally (i.e. the expected number of different classifiers).

This classifier is still is generic in the event type, and it is efficient for all use cases.

5.4.2 Event Stream

The event stream is the main event bus of each actor system: it is used for carrying *log messages* and Dead Letters and may be used by the user code for other purposes as well. It uses Subchannel Classification which enables registering to related sets of channels (as is used for `RemoteLifeCycleMessage`). The following example demonstrates how a simple subscription works. Given a simple actor:

```
import akka.actor.Props;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.UntypedActor;
import akka.actor.DeadLetter;
```

```
public static class DeadLetterActor extends UntypedActor {
  public void onReceive(Object message) {
    if (message instanceof DeadLetter) {
      System.out.println(message);
    }
  }
}
```

it can be subscribed like this:

```
final ActorSystem system = ActorSystem.create("DeadLetters");
final ActorRef actor = system.actorOf(new Props(DeadLetterActor.class));
system.eventStream().subscribe(actor, DeadLetter.class);
```

Default Handlers

Upon start-up the actor system creates and subscribes actors to the event stream for logging: these are the handlers which are configured for example in `application.conf`:

```
akka {
  event-handlers = ["akka.event.Logging$DefaultLogger"]
}
```

The handlers listed here by fully-qualified class name will be subscribed to all log event classes with priority higher than or equal to the configured log-level and their subscriptions are kept in sync when changing the log-level at runtime:

```
system.eventStream.setLogLevel(Logging.DebugLevel());
```

This means that log events for a level which will not be logged are typically not dispatched at all (unless manual subscriptions to the respective event class have been done)

Dead Letters

As described at *Stopping actors*, messages queued when an actor terminates or sent after its death are re-routed to the dead letter mailbox, which by default will publish the messages wrapped in `DeadLetter`. This wrapper holds the original sender, receiver and message of the envelope which was redirected.

Other Uses

The event stream is always there and ready to be used, just publish your own events (it accepts `Object`) and subscribe listeners to the corresponding JVM classes.

5.5 Scheduler (Java)

Sometimes the need for making things happen in the future arises, and where do you go look then? Look no further than `ActorSystem`! There you find the `scheduler` method that returns an instance of akka.actor.Scheduler, this instance is unique per ActorSystem and is used internally for scheduling things to happen at specific points in time. Please note that the scheduled tasks are executed by the default `MessageDispatcher` of the `ActorSystem`.

You can schedule sending of messages to actors and execution of tasks (functions or Runnable). You will get a `Cancellable` back that you can call `cancel` on to cancel the execution of the scheduled operation.

Warning: The default implementation of `Scheduler` used by Akka is based on the Netty `HashedWheelTimer`. It does not execute tasks at the exact time, but on every tick, it will run everything that is overdue. The accuracy of the default Scheduler can be modified by the "ticks-per-wheel" and "tick-duration" configuration properties. For more information, see: HashedWheelTimers.

5.5.1 Some examples

```
import akka.actor.Props;
import akka.util.Duration;
import java.util.concurrent.TimeUnit;

    //Schedules to send the "foo"-message to the testActor after 50ms
    system.scheduler().scheduleOnce(Duration.create(50, TimeUnit.MILLISECONDS), testActor, "foo");
```

```
//Schedules a Runnable to be executed (send the current time) to the testActor after 50ms
system.scheduler().scheduleOnce(Duration.create(50, TimeUnit.MILLISECONDS), new Runnable() {
  @Override
  public void run() {
    testActor.tell(System.currentTimeMillis());
  }
});
```

```
import akka.actor.Props;
import akka.util.Duration;
import java.util.concurrent.TimeUnit;

import akka.actor.UntypedActor;
import akka.actor.UntypedActorFactory;
import akka.actor.Cancellable;

    ActorRef tickActor = system.actorOf(new Props().withCreator(new UntypedActorFactory() {
      public UntypedActor create() {
        return new UntypedActor() {
          public void onReceive(Object message) {
            if (message.equals("Tick")) {
              // Do someting
            } else {
              unhandled(message);
            }
          }
        };
      }
    }));

    //This will schedule to send the Tick-message
    //to the tickActor after 0ms repeating every 50ms
    Cancellable cancellable = system.scheduler().schedule(Duration.Zero(), Duration.create(50, Ti
        tickActor, "Tick");

    //This cancels further Ticks to be sent
    cancellable.cancel();
```

5.5.2 From `akka.actor.ActorSystem`

```
/**
 * Light-weight scheduler for running asynchronous tasks after some deadline
 * in the future. Not terribly precise but cheap.
 */
def scheduler: Scheduler
```

5.5.3 The Scheduler interface

```
/**
 * An Akka scheduler service. This one needs one special behavior: if
 * Closeable, it MUST execute all outstanding tasks upon .close() in order
 * to properly shutdown all dispatchers.
 *
 * Furthermore, this timer service MUST throw IllegalStateException if it
 * cannot schedule a task. Once scheduled, the task MUST be executed. If
 * executed upon close(), the task may execute before its timeout.
 */
trait Scheduler {
  /**
```

```
   * Schedules a message to be sent repeatedly with an initial delay and
   * frequency. E.g. if you would like a message to be sent immediately and
   * thereafter every 500ms you would set delay=Duration.Zero and
   * frequency=Duration(500, TimeUnit.MILLISECONDS)
   *
   * Java & Scala API
   */
  def schedule(
    initialDelay: Duration,
    frequency: Duration,
    receiver: ActorRef,
    message: Any): Cancellable

  /**
   * Schedules a function to be run repeatedly with an initial delay and a
   * frequency. E.g. if you would like the function to be run after 2 seconds
   * and thereafter every 100ms you would set delay = Duration(2, TimeUnit.SECONDS)
   * and frequency = Duration(100, TimeUnit.MILLISECONDS)
   *
   * Scala API
   */
  def schedule(
    initialDelay: Duration, frequency: Duration)(f: ⇒ Unit): Cancellable

  /**
   * Schedules a function to be run repeatedly with an initial delay and
   * a frequency. E.g. if you would like the function to be run after 2
   * seconds and thereafter every 100ms you would set delay = Duration(2,
   * TimeUnit.SECONDS) and frequency = Duration(100, TimeUnit.MILLISECONDS)
   *
   * Java API
   */
  def schedule(
    initialDelay: Duration, frequency: Duration, runnable: Runnable): Cancellable

  /**
   * Schedules a Runnable to be run once with a delay, i.e. a time period that
   * has to pass before the runnable is executed.
   *
   * Java & Scala API
   */
  def scheduleOnce(delay: Duration, runnable: Runnable): Cancellable

  /**
   * Schedules a message to be sent once with a delay, i.e. a time period that has
   * to pass before the message is sent.
   *
   * Java & Scala API
   */
  def scheduleOnce(delay: Duration, receiver: ActorRef, message: Any): Cancellable

  /**
   * Schedules a function to be run once with a delay, i.e. a time period that has
   * to pass before the function is run.
   *
   * Scala API
   */
  def scheduleOnce(delay: Duration)(f: ⇒ Unit): Cancellable
}
```

5.5.4 The Cancellable interface

This allows you to `cancel` something that has been scheduled for execution.

Warning: This does not abort the execution of the task, if it had already been started.

```
/**
 * Signifies something that can be cancelled
 * There is no strict guarantee that the implementation is thread-safe,
 * but it should be good practice to make it so.
 */
trait Cancellable {
  /**
   * Cancels this Cancellable
   *
   * Java & Scala API
   */
  def cancel(): Unit

  /**
   * Returns whether this Cancellable has been cancelled
   *
   * Java & Scala API
   */
  def isCancelled: Boolean
}
```

5.6 Futures (Java)

5.6.1 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).

5.6.2 Execution Contexts

In order to execute callbacks and operations, Futures need something called an `ExecutionContext`, which is very similar to a `java.util.concurrent.Executor`. if you have an `ActorSystem` in scope, it will use its default dispatcher as the `ExecutionContext`, or you can use the factory methods provided by the `ExecutionContexts` class to wrap `Executors` and `ExecutorServices`, or even create your own.

```
import akka.dispatch.*;
import akka.util.Timeout;

import akka.dispatch.ExecutionContexts;
import akka.dispatch.ExecutionContextExecutorService;

      ExecutionContextExecutorService ec =
        ExecutionContexts.fromExecutorService(yourExecutorServiceGoesHere);

      //Use ec with your Futures
      Future<String> f1 = Futures.successful("foo", ec);

      // Then you shut the ec down somewhere at the end of your program/application.
      ec.shutdown();
```

5.6.3 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 `ask` method to send a message will return a Future. To wait for and retrieve the actual result the simplest method is:

```
import akka.dispatch.*;
import akka.util.Timeout;

    Timeout timeout = new Timeout(Duration.parse("5 seconds"));
    Future<Object> future = Patterns.ask(actor, msg, timeout);
    String result = (String) Await.result(future, timeout.duration());
```

This will cause the current thread to block and wait for the `UntypedActor` to 'complete' the `Future` with it's reply. Blocking is discouraged though as it can cause performance problem. The blocking operations are located in `Await.result` and `Await.ready` to make it easy to spot where blocking occurs. Alternatives to blocking are discussed further within this documentation. Also note that the `Future` returned by an `UntypedActor` is a `Future<Object>` since an `UntypedActor` is dynamic. That is why the cast to `String` is used in the above sample.

5.6.4 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.util.Duration;
import akka.japi.Function;
import java.util.concurrent.Callable;
import static akka.dispatch.Futures.future;
import static java.util.concurrent.TimeUnit.SECONDS;

    Future<String> f = future(new Callable<String>() {
      public String call() {
        return "Hello" + "World";
      }
    }, system.dispatcher());
    String result = (String) Await.result(f, Duration.create(1, SECONDS));
```

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`.

You can also create already completed Futures using the `Futures` class, which can be either successes:

```
Future<String> future = Futures.successful("Yay!", system.dispatcher());
```

Or failures:

```
Future<String> otherFuture = Futures.failed(new IllegalArgumentException("Bang!"), system.dispatch
```

5.6.5 Functional Futures

Akka's `Future` has 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 `Mapper` 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.util.Duration;
import akka.japi.Function;
import java.util.concurrent.Callable;
import static akka.dispatch.Futures.future;
import static java.util.concurrent.TimeUnit.SECONDS;

    Future<String> f1 = future(new Callable<String>() {
      public String call() {
        return "Hello" + "World";
      }
    }, system.dispatcher());

    Future<Integer> f2 = f1.map(new Mapper<String, Integer>() {
      public Integer apply(String s) {
        return s.length();
      }
    });

    int result = Await.result(f2, Duration.create(1, SECONDS));
    assertEquals(10, result);
```

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 its 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:

```
Future<String> f1 = future(new Callable<String>() {
  public String call() throws Exception {
    Thread.sleep(100);
    return "Hello" + "World";
  }
}, system.dispatcher());

Future<Integer> f2 = f1.map(new Mapper<String, Integer>() {
  public Integer apply(String s) {
    return s.length();
  }
});
```

The original `Future` will take at least 0.1 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:

```
Future<String> f1 = future(new Callable<String>() {
  public String call() {
    return "Hello" + "World";
  }
}, system.dispatcher());

Thread.sleep(100);
```

```
Future<Integer> f2 = f1.map(new Mapper<String, Integer>() {
  public Integer apply(String s) {
    return s.length();
  }
});
```

Our little string has been processed long before our 0.1 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`:

```
Future<String> f1 = future(new Callable<String>() {
  public String call() {
    return "Hello" + "World";
  }
}, system.dispatcher());

Future<Integer> f2 = f1.flatMap(new Mapper<String, Future<Integer>>() {
  public Future<Integer> apply(final String s) {
    return future(new Callable<Integer>() {
      public Integer call() {
        return s.length();
      }
    }, system.dispatcher());
  }
});
```

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.

If you need to do conditional propagation, you can use `filter`:

```
Future<Integer> future1 = Futures.successful(4, system.dispatcher());
Future<Integer> successfulFilter = future1.filter(Filter.filterOf(new Function<Integer, Boolean>(
  public Boolean apply(Integer i) {
    return i % 2 == 0;
  }
}));

Future<Integer> failedFilter = future1.filter(Filter.filterOf(new Function<Integer, Boolean>() {
  public Boolean apply(Integer i) {
    return i % 2 != 0;
  }
}));
//When filter fails, the returned Future will be failed with a scala.MatchError
```

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 static akka.dispatch.Futures.sequence;

    //Some source generating a sequence of Future<Integer>:s
    Iterable<Future<Integer>> listOfFutureInts = source;

    // now we have a Future[Iterable[Integer]]
    Future<Iterable<Integer>> futureListOfInts = sequence(listOfFutureInts, system.dispatcher());
```

```
    // Find the sum of the odd numbers
    Future<Long> futureSum = futureListOfInts.map(new Mapper<Iterable<Integer>, Long>() {
      public Long apply(Iterable<Integer> ints) {
        long sum = 0;
        for (Integer i : ints)
          sum += i;
        return sum;
      }
    });

    long result = Await.result(futureSum, Duration.create(1, SECONDS));
```

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<B>` and returns a `Future<Iterable<B>>`, enabling parallel `map` over the sequence, if you use `Futures.future` to create the `Future`.

```
import static akka.dispatch.Futures.traverse;

    //Just a sequence of Strings
    Iterable<String> listStrings = Arrays.asList("a", "b", "c");

    Future<Iterable<String>> futureResult = traverse(listStrings, new Function<String, Future<Str
      public Future<String> apply(final String r) {
        return future(new Callable<String>() {
          public String call() {
            return r.toUpperCase();
          }
        }, system.dispatcher());
      }
    }, system.dispatcher());

    //Returns the sequence of strings as upper case
    Iterable<String> result = Await.result(futureResult, Duration.create(1, SECONDS));
    assertEquals(Arrays.asList("A", "B", "C"), result);
```

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 be started when the last of the Futures is completed.

```
import akka.japi.Function2;
import static akka.dispatch.Futures.fold;


    //A sequence of Futures, in this case Strings
    Iterable<Future<String>> futures = source;

    //Start value is the empty string
    Future<String> resultFuture = fold("", futures, new Function2<String, String, String>() {
      public String apply(String r, String t) {
        return r + t; //Just concatenate
      }
    }, system.dispatcher());
    String result = Await.result(resultFuture, Duration.create(1, SECONDS));
```

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 empty String.

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 static akka.dispatch.Futures.reduce;


    //A sequence of Futures, in this case Strings
    Iterable<Future<String>> futures = source;

    Future<Object> resultFuture = reduce(futures, new Function2<Object, String, Object>() {
      public Object apply(Object r, String t) {
        return r + t; //Just concatenate
      }
    }, system.dispatcher());

    Object result = Await.result(resultFuture, Duration.create(1, SECONDS));
```

Same as with `fold`, the execution will be started when the last of the Futures is completed, 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.

5.6.6 Callbacks

Sometimes you just want to listen to a `Future` being completed, and react to that not by creating a new Future, but by side-effecting. For this Akka supports `onComplete`, `onSuccess` and `onFailure`, of which the latter two are specializations of the first.

```
future.onSuccess(new OnSuccess<String>() {
  public void onSuccess(String result) {
    if ("bar" == result) {
      //Do something if it resulted in "bar"
    } else {
      //Do something if it was some other String
    }
  }
});
```

```
future.onFailure(new OnFailure() {
  public void onFailure(Throwable failure) {
    if (failure instanceof IllegalStateException) {
      //Do something if it was this particular failure
    } else {
      //Do something if it was some other failure
    }
  }
});
```

```
future.onComplete(new OnComplete<String>() {
    public void onComplete(Throwable failure, String result) {
        if (failure != null) {
            //We got a failure, handle it here
        } else {
            // We got a result, do something with it
        }
    }
});
```

5.6.7 Ordering

Since callbacks are executed in any order and potentially in parallel, it can be tricky at the times when you need sequential ordering of operations. But there's a solution! And it's name is `andThen`, and it creates a new Future with the specified callback, a Future that will have the same result as the Future it's called on, which allows for ordering like in the following sample:

```
Future<String> future1 = Futures.successful("value", system.dispatcher()).andThen(new OnComplete<S
    public void onComplete(Throwable failure, String result) {
        if (failure != null)
            sendToIssueTracker(failure);
    }
}).andThen(new OnComplete<String>() {
  public void onComplete(Throwable failure, String result) {
    if (result != null)
      sendToTheInternetz(result);
  }
});
```

5.6.8 Auxiliary methods

`Future fallbackTo` combines 2 Futures into a new `Future`, and will hold the successful value of the second `Future` if the first `Future` fails.

```
Future<String> future1 = Futures.failed(new IllegalStateException("OHNOES1"), system.dispatcher()
Future<String> future2 = Futures.failed(new IllegalStateException("OHNOES2"), system.dispatcher()
Future<String> future3 = Futures.successful("bar", system.dispatcher());
Future<String> future4 = future1.fallbackTo(future2).fallbackTo(future3); // Will have "bar" in th
String result = Await.result(future4, Duration.create(1, SECONDS));
assertEquals("bar", result);
```

You can also combine two Futures into a new `Future` that will hold a tuple of the two Futures successful results, using the `zip` operation.

```
Future<String> future1 = Futures.successful("foo", system.dispatcher());
Future<String> future2 = Futures.successful("bar", system.dispatcher());
Future<String> future3 = future1.zip(future2).map(new Mapper<scala.Tuple2<String, String>, String>
  public String apply(scala.Tuple2<String, String> zipped) {
    return zipped._1() + " " + zipped._2();
  }
});

String result = Await.result(future3, Duration.create(1, SECONDS));
assertEquals("foo bar", result);
```

5.6.9 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 `Await.result` 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:

```
Future<Integer> future = future(new Callable<Integer>() {
  public Integer call() {
    return 1 / 0;
  }
}, system.dispatcher()).recover(new Recover<Integer>() {
```

```
  public Integer recover(Throwable problem) throws Throwable {
    if (problem instanceof ArithmeticException)
      return 0;
    else
      throw problem;
  }
});
int result = Await.result(future, Duration.create(1, SECONDS));
assertEquals(result, 0);
```

In this example, if the actor replied with a `akka.actor.Status.Failure` containing the `ArithmeticException`, our `Future` would have a result of 0. The `recover` method works very similarly to the standard try/catch blocks, so multiple `Exception`s 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 `recover` method.

You can also use the `recoverWith` method, which has the same relationship to `recover` as `flatMap` has to `map`, and is use like this:

```
Future<Integer> future = future(new Callable<Integer>() {
  public Integer call() {
    return 1 / 0;
  }
}, system.dispatcher()).recoverWith(new Recover<Future<Integer>>() {
  public Future<Integer> recover(Throwable problem) throws Throwable {
    if (problem instanceof ArithmeticException) {
      return future(new Callable<Integer>() {
        public Integer call() {
          return 0;
        }
      }, system.dispatcher());
    } else
      throw problem;
  }
});
int result = Await.result(future, Duration.create(1, SECONDS));
assertEquals(result, 0);
```

5.7 Fault Tolerance (Java)

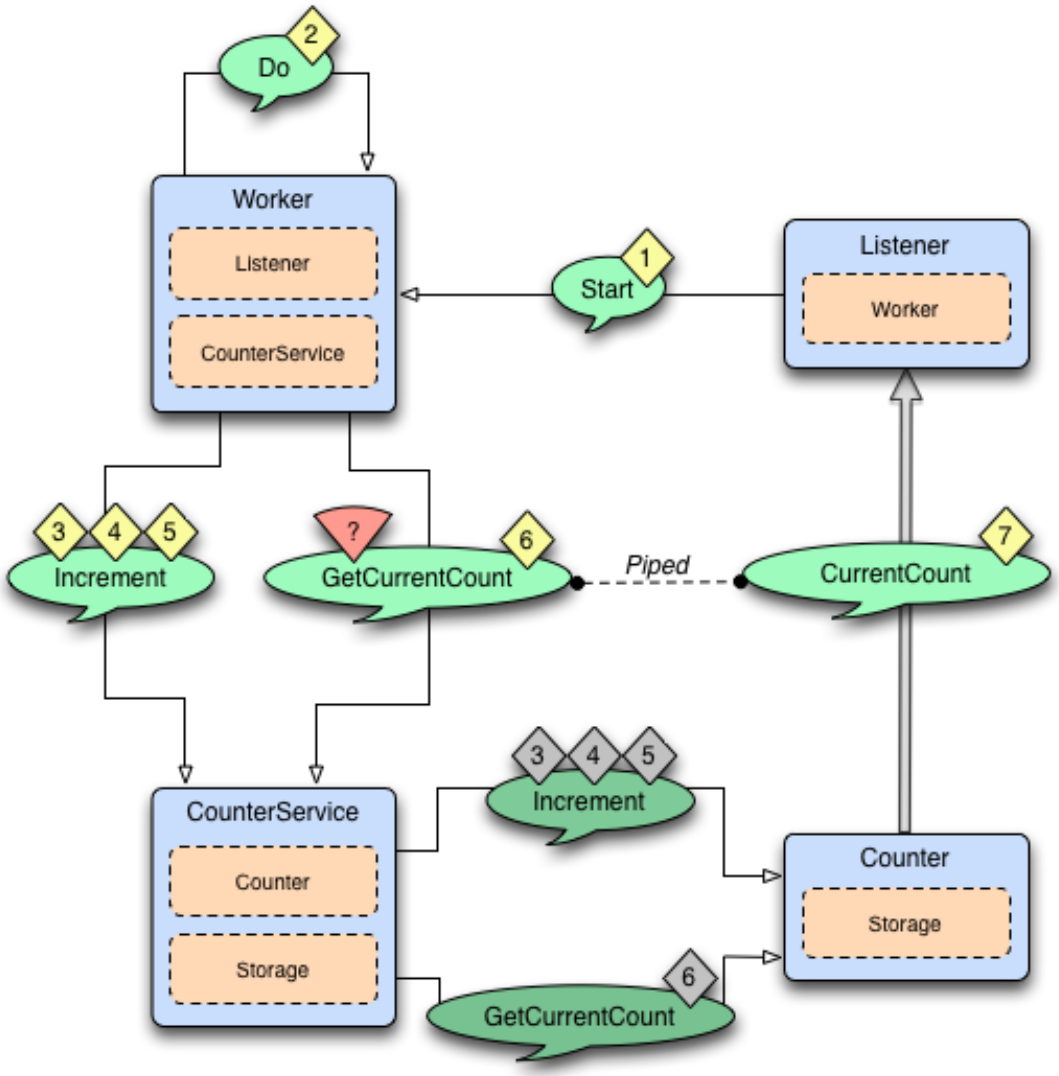
As explained in *Actor Systems* each actor is the supervisor of its children, and as such each actor defines fault handling supervisor strategy. This strategy cannot be changed afterwards as it is an integral part of the actor system's structure.

5.7.1 Fault Handling in Practice

First, let us look at a sample that illustrates one way to handle data store errors, which is a typical source of failure in real world applications. Of course it depends on the actual application what is possible to do when the data store is unavailable, but in this sample we use a best effort re-connect approach.

Read the following source code. The inlined comments explain the different pieces of the fault handling and why they are added. It is also highly recommended to run this sample as it is easy to follow the log output to understand what is happening in runtime.

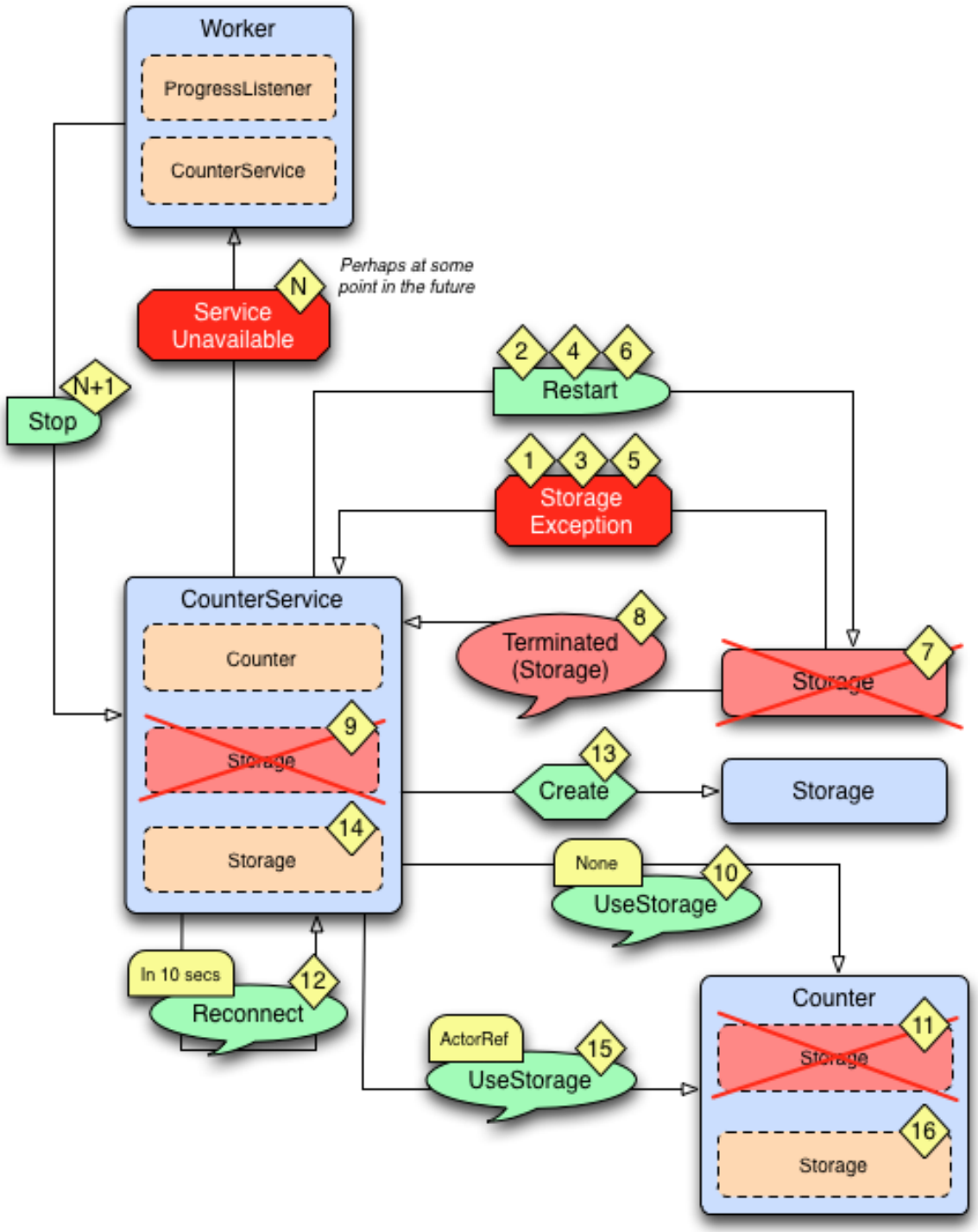
Diagrams of the Fault Tolerance Sample (Java)



The above diagram illustrates the normal message flow.

Normal flow:

Step	Description
1	The progress `Listener` starts the work.
2	The `Worker` schedules work by sending `Do` messages periodically to itself
3, 4, 5	When receiving `Do` the `Worker` tells the `CounterService` to increment the counter, three times. The `Increment` message is forwarded to the `Counter`, which updates its counter variable and sends current value to the `Storage`.
6, 7	The `Worker` asks the `CounterService` of current value of the counter and pipes the result back to the `Listener`.



The above diagram illustrates what happens in case of storage failure.

Failure flow:

Step	Description
1	The `Storage` throws `StorageException`.
2	The `CounterService` is supervisor of the `Storage` and restarts the `Storage` when `StorageException` is thrown.
3, 4, 5, 6	The `Storage` continues to fail and is restarted.
7	After 3 failures and restarts within 5 seconds the `Storage` is stopped by its supervisor, i.e. the `CounterService`.
8	The `CounterService` is also watching the `Storage` for termination and receives the `Terminated` message when the `Storage` has been stopped ...
9, 10, 11	and tells the `Counter` that there is no `Storage`.
12	The `CounterService` schedules a `Reconnect` message to itself.
13, 14	When it receives the `Reconnect` message it creates a new `Storage` ...
15, 16	and tells the the `Counter` to use the new `Storage`

Full Source Code of the Fault Tolerance Sample (Java)

```
// imports ...

public class FaultHandlingDocSample {

  /**
   * Runs the sample
   */
  public static void main(String[] args) {
    Config config = ConfigFactory.parseString("akka.loglevel = DEBUG \n" + "akka.actor.debug.lifec

    ActorSystem system = ActorSystem.create("FaultToleranceSample", config);
    ActorRef worker = system.actorOf(new Props(Worker.class), "worker");
    ActorRef listener = system.actorOf(new Props(Listener.class), "listener");
    // start the work and listen on progress
    // note that the listener is used as sender of the tell,
    // i.e. it will receive replies from the worker
    worker.tell(Start, listener);
  }

  /**
   * Listens on progress from the worker and shuts down the system when enough
   * work has been done.
   */
  public static class Listener extends UntypedActor {
    final LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    @Override
    public void preStart() {
      // If we don't get any progress within 15 seconds then the service is unavailable
      getContext().setReceiveTimeout(Duration.parse("15 seconds"));
    }

    public void onReceive(Object msg) {
      log.debug("received message {}", msg);
      if (msg instanceof Progress) {
        Progress progress = (Progress) msg;
        log.info("Current progress: {} %", progress.percent);
        if (progress.percent >= 100.0) {
          log.info("That's all, shutting down");
          getContext().system().shutdown();
```

```
      }
    } else if (msg == Actors.receiveTimeout()) {
      // No progress within 15 seconds, ServiceUnavailable
      log.error("Shutting down due to unavailable service");
      getContext().system().shutdown();
    } else {
      unhandled(msg);
    }
  }
}

// messages ...

/**
 * Worker performs some work when it receives the Start message. It will
 * continuously notify the sender of the Start message of current Progress.
 * The Worker supervise the CounterService.
 */
public static class Worker extends UntypedActor {
  final LoggingAdapter log = Logging.getLogger(getContext().system(), this);
  final Timeout askTimeout = new Timeout(Duration.parse("5 seconds"));

  // The sender of the initial Start message will continuously be notified about progress
  ActorRef progressListener;
  final ActorRef counterService = getContext().actorOf(new Props(CounterService.class), "counte
  final int totalCount = 51;

  // Stop the CounterService child if it throws ServiceUnavailable
  private static SupervisorStrategy strategy = new OneForOneStrategy(-1, Duration.Inf(),
      new Function<Throwable, Directive>() {
        @Override
        public Directive apply(Throwable t) {
          if (t instanceof ServiceUnavailable) {
            return stop();
          } else {
            return escalate();
          }
        }
      });

  @Override
  public SupervisorStrategy supervisorStrategy() {
    return strategy;
  }

  public void onReceive(Object msg) {
    log.debug("received message {}", msg);
    if (msg.equals(Start) && progressListener == null) {
      progressListener = getSender();
      getContext().system().scheduler().schedule(Duration.Zero(), Duration.parse("1 second"), ge
    } else if (msg.equals(Do)) {
      counterService.tell(new Increment(1), getSelf());
      counterService.tell(new Increment(1), getSelf());
      counterService.tell(new Increment(1), getSelf());

      // Send current progress to the initial sender
      pipe(ask(counterService, GetCurrentCount, askTimeout)
             .mapTo(manifest(CurrentCount.class))
             .map(new Mapper<CurrentCount, Progress>() {
          public Progress apply(CurrentCount c) {
              return new Progress(100.0 * c.count / totalCount);
          }
      }))
```

```
        .to(progressListener);
    } else {
      unhandled(msg);
    }
  }
}

// messages ...

/**
 * Adds the value received in Increment message to a persistent counter.
 * Replies with CurrentCount when it is asked for CurrentCount. CounterService
 * supervise Storage and Counter.
 */
public static class CounterService extends UntypedActor {

  // Reconnect message
  static final Object Reconnect = "Reconnect";

  private static class SenderMsgPair {
    final ActorRef sender;
    final Object msg;

    SenderMsgPair(ActorRef sender, Object msg) {
      this.msg = msg;
      this.sender = sender;
    }
  }

  final LoggingAdapter log = Logging.getLogger(getContext().system(), this);
  final String key = getSelf().path().name();
  ActorRef storage;
  ActorRef counter;
  final List<SenderMsgPair> backlog = new ArrayList<SenderMsgPair>();
  final int MAX_BACKLOG = 10000;

  // Restart the storage child when StorageException is thrown.
  // After 3 restarts within 5 seconds it will be stopped.
  private static SupervisorStrategy strategy = new OneForOneStrategy(3, Duration.parse("5 secon
      new Function<Throwable, Directive>() {
        @Override
        public Directive apply(Throwable t) {
          if (t instanceof StorageException) {
            return restart();
          } else {
            return escalate();
          }
        }
      });

  @Override
  public SupervisorStrategy supervisorStrategy() {
    return strategy;
  }

  @Override
  public void preStart() {
    initStorage();
  }

  /**
   * The child storage is restarted in case of failure, but after 3 restarts,
   * and still failing it will be stopped. Better to back-off than
```

```
     * continuously failing. When it has been stopped we will schedule a
     * Reconnect after a delay. Watch the child so we receive Terminated message
     * when it has been terminated.
     */
    void initStorage() {
      storage = getContext().watch(getContext().actorOf(new Props(Storage.class), "storage"));
      // Tell the counter, if any, to use the new storage
      if (counter != null)
        counter.tell(new UseStorage(storage), getSelf());
      // We need the initial value to be able to operate
      storage.tell(new Get(key), getSelf());
    }

    @Override
    public void onReceive(Object msg) {
      log.debug("received message {}", msg);
      if (msg instanceof Entry && ((Entry) msg).key.equals(key) && counter == null) {
        // Reply from Storage of the initial value, now we can create the Counter
        final long value = ((Entry) msg).value;
        counter = getContext().actorOf(new Props().withCreator(new UntypedActorFactory() {
          public Actor create() {
            return new Counter(key, value);
          }
        }));
        // Tell the counter to use current storage
        counter.tell(new UseStorage(storage), getSelf());
        // and send the buffered backlog to the counter
        for (SenderMsgPair each : backlog) {
          counter.tell(each.msg, each.sender);
        }
        backlog.clear();
      } else if (msg instanceof Increment) {
        forwardOrPlaceInBacklog(msg);
      } else if (msg.equals(GetCurrentCount)) {
        forwardOrPlaceInBacklog(msg);
      } else if (msg instanceof Terminated) {
        // After 3 restarts the storage child is stopped.
        // We receive Terminated because we watch the child, see initStorage.
        storage = null;
        // Tell the counter that there is no storage for the moment
        counter.tell(new UseStorage(null), getSelf());
        // Try to re-establish storage after while
        getContext().system().scheduler().scheduleOnce(Duration.parse("10 seconds"), getSelf(), Re
      } else if (msg.equals(Reconnect)) {
        // Re-establish storage after the scheduled delay
        initStorage();
      } else {
        unhandled(msg);
      }
    }

    void forwardOrPlaceInBacklog(Object msg) {
      // We need the initial value from storage before we can start delegate to the counter.
      // Before that we place the messages in a backlog, to be sent to the counter when
      // it is initialized.
      if (counter == null) {
        if (backlog.size() >= MAX_BACKLOG)
          throw new ServiceUnavailable("CounterService not available, lack of initial value");
        backlog.add(new SenderMsgPair(getSender(), msg));
      } else {
        counter.forward(msg, getContext());
      }
    }
```

```
}

// messages ...

/**
 * The in memory count variable that will send current value to the Storage,
 * if there is any storage available at the moment.
 */
public static class Counter extends UntypedActor {
  final LoggingAdapter log = Logging.getLogger(getContext().system(), this);
  final String key;
  long count;
  ActorRef storage;

  public Counter(String key, long initialValue) {
    this.key = key;
    this.count = initialValue;
  }

  @Override
  public void onReceive(Object msg) {
    log.debug("received message {}", msg);
    if (msg instanceof UseStorage) {
      storage = ((UseStorage) msg).storage;
      storeCount();
    } else if (msg instanceof Increment) {
      count += ((Increment) msg).n;
      storeCount();
    } else if (msg.equals(GetCurrentCount)) {
      getSender().tell(new CurrentCount(key, count), getSelf());
    } else {
      unhandled(msg);
    }
  }

  void storeCount() {
    // Delegate dangerous work, to protect our valuable state.
    // We can continue without storage.
    if (storage != null) {
      storage.tell(new Store(new Entry(key, count)), getSelf());
    }
  }
}

// messages ...

/**
 * Saves key/value pairs to persistent storage when receiving Store message.
 * Replies with current value when receiving Get message. Will throw
 * StorageException if the underlying data store is out of order.
 */
public static class Storage extends UntypedActor {

  final LoggingAdapter log = Logging.getLogger(getContext().system(), this);
  final DummyDB db = DummyDB.instance;

  @Override
  public void onReceive(Object msg) {
    log.debug("received message {}", msg);
    if (msg instanceof Store) {
      Store store = (Store) msg;
      db.save(store.entry.key, store.entry.value);
    } else if (msg instanceof Get) {
```

```
        Get get = (Get) msg;
        Long value = db.load(get.key);
        getSender().tell(new Entry(get.key, value == null ? Long.valueOf(0L) : value), getSelf());
      } else {
        unhandled(msg);
      }
    }
  }

  // dummydb ...
}
```

5.7.2 Creating a Supervisor Strategy

The following sections explain the fault handling mechanism and alternatives in more depth.

For the sake of demonstration let us consider the following strategy:

```
private static SupervisorStrategy strategy = new OneForOneStrategy(10, Duration.parse("1 minute"),
    new Function<Throwable, Directive>() {
      @Override
      public Directive apply(Throwable t) {
        if (t instanceof ArithmeticException) {
          return resume();
        } else if (t instanceof NullPointerException) {
          return restart();
        } else if (t instanceof IllegalArgumentException) {
          return stop();
        } else {
          return escalate();
        }
      }
    });

@Override
public SupervisorStrategy supervisorStrategy() {
  return strategy;
}
```

I have chosen a few well-known exception types in order to demonstrate the application of the fault handling directives described in *Supervision and Monitoring*. First off, it is a one-for-one strategy, meaning that each child is treated separately (an all-for-one strategy works very similarly, the only difference is that any decision is applied to all children of the supervisor, not only the failing one). There are limits set on the restart frequency, namely maximum 10 restarts per minute. `-1` and `Duration.Inf()` means that the respective limit does not apply, leaving the possibility to specify an absolute upper limit on the restarts or to make the restarts work infinitely.

5.7.3 Default Supervisor Strategy

`Escalate` is used if the defined strategy doesn't cover the exception that was thrown.

When the supervisor strategy is not defined for an actor the following exceptions are handled by default:

- `ActorInitializationException` will stop the failing child actor
- `ActorKilledException` will stop the failing child actor
- `Exception` will restart the failing child actor
- Other types of `Throwable` will be escalated to parent actor

If the exception escalate all the way up to the root guardian it will handle it in the same way as the default strategy defined above.

5.7.4 Test Application

The following section shows the effects of the different directives in practice, wherefor a test setup is needed. First off, we need a suitable supervisor:

```
static public class Supervisor extends UntypedActor {

  private static SupervisorStrategy strategy = new OneForOneStrategy(10, Duration.parse("1 minute
      new Function<Throwable, Directive>() {
        @Override
        public Directive apply(Throwable t) {
          if (t instanceof ArithmeticException) {
            return resume();
          } else if (t instanceof NullPointerException) {
            return restart();
          } else if (t instanceof IllegalArgumentException) {
            return stop();
          } else {
            return escalate();
          }
        }
      });

  @Override
  public SupervisorStrategy supervisorStrategy() {
    return strategy;
  }


  public void onReceive(Object o) {
    if (o instanceof Props) {
      getSender().tell(getContext().actorOf((Props) o));
    } else {
      unhandled(o);
    }
  }
}
```

This supervisor will be used to create a child, with which we can experiment:

```
static public class Child extends UntypedActor {
  int state = 0;

  public void onReceive(Object o) throws Exception {
    if (o instanceof Exception) {
      throw (Exception) o;
    } else if (o instanceof Integer) {
      state = (Integer) o;
    } else if (o.equals("get")) {
      getSender().tell(state);
    } else {
      unhandled(o);
    }
  }
}
```

The test is easier by using the utilities described in *Testing Actor Systems (Scala)*, where `TestProbe` provides an actor ref useful for receiving and inspecting replies.

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.SupervisorStrategy;
import static akka.actor.SupervisorStrategy.*;
import akka.actor.OneForOneStrategy;
```

```
import akka.actor.Props;
import akka.actor.Terminated;
import akka.actor.UntypedActor;
import akka.dispatch.Await;
import static akka.pattern.Patterns.ask;
import akka.util.Duration;
import akka.testkit.AkkaSpec;
import akka.testkit.TestProbe;

public class FaultHandlingTestBase {
  static ActorSystem system;
  Duration timeout = Duration.create(5, SECONDS);

  @BeforeClass
  public static void start() {
    system = ActorSystem.create("test", AkkaSpec.testConf());
  }

  @AfterClass
  public static void cleanup() {
    system.shutdown();
  }

  @Test
  public void mustEmploySupervisorStrategy() throws Exception {
    // code here
  }

}
```

Let us create actors:

```
Props superprops = new Props(Supervisor.class);
ActorRef supervisor = system.actorOf(superprops, "supervisor");
ActorRef child = (ActorRef) Await.result(ask(supervisor, new Props(Child.class), 5000), timeout);
```

The first test shall demonstrate the `Resume` directive, so we try it out by setting some non-initial state in the actor and have it fail:

```
child.tell(42);
assert Await.result(ask(child, "get", 5000), timeout).equals(42);
child.tell(new ArithmeticException());
assert Await.result(ask(child, "get", 5000), timeout).equals(42);
```

As you can see the value 42 survives the fault handling directive. Now, if we change the failure to a more serious `NullPointerException`, that will no longer be the case:

```
child.tell(new NullPointerException());
assert Await.result(ask(child, "get", 5000), timeout).equals(0);
```

And finally in case of the fatal `IllegalArgumentException` the child will be terminated by the supervisor:

```
final TestProbe probe = new TestProbe(system);
probe.watch(child);
child.tell(new IllegalArgumentException());
probe.expectMsg(new Terminated(child));
```

Up to now the supervisor was completely unaffected by the child's failure, because the directives set did handle it. In case of an `Exception`, this is not true anymore and the supervisor escalates the failure.

```
child = (ActorRef) Await.result(ask(supervisor, new Props(Child.class), 5000), timeout);
probe.watch(child);
assert Await.result(ask(child, "get", 5000), timeout).equals(0);
```

```
child.tell(new Exception());
probe.expectMsg(new Terminated(child));
```

The supervisor itself is supervised by the top-level actor provided by the `ActorSystem`, which has the default policy to restart in case of all `Exception` cases (with the notable exceptions of `ActorInitializationException` and `ActorKilledException`). Since the default directive in case of a restart is to kill all children, we expected our poor child not to survive this failure.

In case this is not desired (which depends on the use case), we need to use a different supervisor which overrides this behavior.

```
static public class Supervisor2 extends UntypedActor {

  private static SupervisorStrategy strategy = new OneForOneStrategy(10, Duration.parse("1 minute
      new Function<Throwable, Directive>() {
        @Override
        public Directive apply(Throwable t) {
          if (t instanceof ArithmeticException) {
            return resume();
          } else if (t instanceof NullPointerException) {
            return restart();
          } else if (t instanceof IllegalArgumentException) {
            return stop();
          } else {
            return escalate();
          }
        }
      });

  @Override
  public SupervisorStrategy supervisorStrategy() {
    return strategy;
  }


  public void onReceive(Object o) {
    if (o instanceof Props) {
      getSender().tell(getContext().actorOf((Props) o));
    } else {
      unhandled(o);
    }
  }

  @Override
  public void preRestart(Throwable cause, Option<Object> msg) {
    // do not kill all children, which is the default here
  }
}
```

With this parent, the child survives the escalated restart, as demonstrated in the last test:

```
superprops = new Props(Supervisor2.class);
supervisor = system.actorOf(superprops, "supervisor2");
child = (ActorRef) Await.result(ask(supervisor, new Props(Child.class), 5000), timeout);
child.tell(23);
assert Await.result(ask(child, "get", 5000), timeout).equals(23);
child.tell(new Exception());
assert Await.result(ask(child, "get", 5000), timeout).equals(0);
```

5.8 Dispatchers (Java)

An Akka `MessageDispatcher` is what makes Akka Actors "tick", it is the engine of the machine so to speak. All `MessageDispatcher` implementations are also an `ExecutionContext`, which means that they can be used to execute arbitrary code, for instance *Futures (Java)*.

5.8.1 Default dispatcher

Every `ActorSystem` will have a default dispatcher that will be used in case nothing else is configured for an `Actor`. The default dispatcher can be configured, and is by default a `Dispatcher` with a "fork-join-executor", which gives excellent performance in most cases.

5.8.2 Setting the dispatcher for an Actor

So in case you want to give your `Actor` a different dispatcher than the default, you need to do two things, of which the first is:

```
ActorRef myActor =
  system.actorOf(new Props(MyUntypedActor.class).withDispatcher("my-dispatcher"),
    "myactor3");
```

Note: The "dispatcherId" you specify in withDispatcher is in fact a path into your configuration. So in this example it's a top-level section, but you could for instance put it as a sub-section, where you'd use periods to denote sub-sections, like this: `"foo.bar.my-dispatcher"`

And then you just need to configure that dispatcher in your configuration:

```
my-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "fork-join-executor"
  # Configuration for the fork join pool
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 2
    # Parallelism (threads) ... ceil(available processors * factor)
    parallelism-factor = 2.0
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

And here's another example that uses the "thread-pool-executor":

```
my-thread-pool-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "thread-pool-executor"
  # Configuration for the thread pool
  thread-pool-executor {
    # minimum number of threads to cap factor-based core number to
    core-pool-size-min = 2
```

```
    # No of core threads ... ceil(available processors * factor)
    core-pool-size-factor = 2.0
    # maximum number of threads to cap factor-based number to
    core-pool-size-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

For more options, see the default-dispatcher section of the *Configuration*.

5.8.3 Types of dispatchers

There are 4 different types of message dispatchers:

- Dispatcher
  - This is an event-based dispatcher that binds a set of Actors to a thread pool. It is the default dispatcher used if one is not specified.
  - Sharability: Unlimited
  - Mailboxes: Any, creates one per Actor
  - Use cases: Default dispatcher, Bulkheading
  - **Driven by:** `java.util.concurrent.ExecutorService` specify using "executor" using "fork-join-executor", "thread-pool-executor" or the FQCN of an `akka.dispatcher.ExecutorServiceConfigurator`
- PinnedDispatcher
  - This dispatcher dedicates a unique thread for each actor using it; i.e. each actor will have its own thread pool with only one thread in the pool.
  - Sharability: None
  - Mailboxes: Any, creates one per Actor
  - Use cases: Bulkheading
  - **Driven by:** Any `akka.dispatch.ThreadPoolExecutorConfigurator` by default a "thread-pool-executor"
- BalancingDispatcher
  - This is an executor based event driven dispatcher that will try to redistribute work from busy actors to idle actors.
  - All the actors share a single Mailbox that they get their messages from.
  - It is assumed that all actors using the same instance of this dispatcher can process all messages that have been sent to one of the actors; i.e. the actors belong to a pool of actors, and to the client there is no guarantee about which actor instance actually processes a given message.
  - Sharability: Actors of the same type only
  - Mailboxes: Any, creates one for all Actors
  - Use cases: Work-sharing
  - **Driven by:** `java.util.concurrent.ExecutorService` specify using "executor" using "fork-join-executor", "thread-pool-executor" or the FQCN of an `akka.dispatcher.ExecutorServiceConfigurator`

  - Note that you can **not** use a `BalancingDispatcher` as a **Router Dispatcher**. (You can however use it for the **Routees**)

- CallingThreadDispatcher
  - This dispatcher runs invocations on the current thread only. This dispatcher does not create any new threads, but it can be used from different threads concurrently for the same actor. See *CallingThreadDispatcher* for details and restrictions.
  - Sharability: Unlimited
  - Mailboxes: Any, creates one per Actor per Thread (on demand)
  - Use cases: Testing
  - Driven by: The calling thread (duh)

More dispatcher configuration examples

Configuring a `PinnedDispatcher`:

```
my-pinned-dispatcher {
  executor = "thread-pool-executor"
  type = PinnedDispatcher
}
```

And then using it:

```
ActorRef myActor = system.actorOf(new Props(MyUntypedActor.class)
    .withDispatcher("my-pinned-dispatcher"));
```

5.8.4 Mailboxes

An Akka `Mailbox` holds the messages that are destined for an `Actor`. Normally each `Actor` has its own mailbox, but with example a `BalancingDispatcher` all actors with the same `BalancingDispatcher` will share a single instance.

Builtin implementations

Akka comes shipped with a number of default mailbox implementations:

- UnboundedMailbox
  - Backed by a `java.util.concurrent.ConcurrentLinkedQueue`
  - Blocking: No
  - Bounded: No
- BoundedMailbox
  - Backed by a `java.util.concurrent.LinkedBlockingQueue`
  - Blocking: Yes
  - Bounded: Yes
- UnboundedPriorityMailbox
  - Backed by a `java.util.concurrent.PriorityBlockingQueue`
  - Blocking: Yes
  - Bounded: No
- BoundedPriorityMailbox

  - Backed by a `java.util.PriorityBlockingQueue` wrapped in an `akka.util.BoundedBlockingQueue`
  - Blocking: Yes
  - Bounded: Yes

- Durable mailboxes, see *Durable Mailboxes*.

Mailbox configuration examples

How to create a PriorityMailbox:

```
public static class MyPrioMailbox extends UnboundedPriorityMailbox {
  public MyPrioMailbox(ActorSystem.Settings settings, Config config) { // needed for reflective in
    // Create a new PriorityGenerator, lower prio means more important
    super(new PriorityGenerator() {
      @Override
      public int gen(Object message) {
        if (message.equals("highpriority"))
          return 0; // 'highpriority messages should be treated first if possible
        else if (message.equals("lowpriority"))
          return 2; // 'lowpriority messages should be treated last if possible
        else if (message.equals(Actors.poisonPill()))
          return 3; // PoisonPill when no other left
        else
          return 1; // By default they go between high and low prio
      }
    });
  }
}
```

And then add it to the configuration:

```
prio-dispatcher {
  mailbox-type = "akka.docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
}
```

And then an example on how you would use it:

```
  // We create a new Actor that just prints out what it processes
ActorRef myActor = system.actorOf(
    new Props().withCreator(new UntypedActorFactory() {
      public UntypedActor create() {
        return new UntypedActor() {
          LoggingAdapter log =
                  Logging.getLogger(getContext().system(), this);
          {
            getSelf().tell("lowpriority");
            getSelf().tell("lowpriority");
            getSelf().tell("highpriority");
            getSelf().tell("pigdog");
            getSelf().tell("pigdog2");
            getSelf().tell("pigdog3");
            getSelf().tell("highpriority");
            getSelf().tell(Actors.poisonPill());
          }

          public void onReceive(Object message) {
            log.info(message.toString());
          }
        };
      }
    }).withDispatcher("prio-dispatcher"));
```

```
/*
Logs:
  'highpriority
  'highpriority
  'pigdog
  'pigdog2
  'pigdog3
  'lowpriority
  'lowpriority
*/
```

Note: Make sure to include a constructor which takes `akka.actor.ActorSystem.Settings` and `com.typesafe.config.Config` arguments, as this constructor is invoked reflectively to construct your mailbox type. The config passed in as second argument is that section from the configuration which describes the dispatcher using this mailbox type; the mailbox type will be instantiated once for each dispatcher using it.

5.9 Routing (Java)

A Router is an actor that routes incoming messages to outbound actors. The router routes the messages sent to it to its underlying actors called 'routees'.

Akka comes with some defined routers out of the box, but as you will see in this chapter it is really easy to create your own. The routers shipped with Akka are:

- `akka.routing.RoundRobinRouter`
- `akka.routing.RandomRouter`
- `akka.routing.SmallestMailboxRouter`
- `akka.routing.BroadcastRouter`
- `akka.routing.ScatterGatherFirstCompletedRouter`

5.9.1 Routers In Action

This is an example of how to create a router that is defined in configuration:

```
akka.actor.deployment {
  /router {
    router = round-robin
    nr-of-instances = 5
  }
}
```

```
ActorRef router = system.actorOf(new Props(ExampleActor.class).withRouter(new FromConfig()), "rout
```

This is an example of how to programmatically create a router and set the number of routees it should create:

```
int nrOfInstances = 5;
ActorRef router1 = system.actorOf(new Props(ExampleActor.class).withRouter(new RoundRobinRouter(n
```

You can also give the router already created routees as in:

```
ActorRef actor1 = system.actorOf(new Props(ExampleActor.class));
ActorRef actor2 = system.actorOf(new Props(ExampleActor.class));
ActorRef actor3 = system.actorOf(new Props(ExampleActor.class));
Iterable<ActorRef> routees = Arrays.asList(new ActorRef[] { actor1, actor2, actor3 });
ActorRef router2 = system.actorOf(new Props(ExampleActor.class).withRouter(RoundRobinRouter.create
```

When you create a router programmatically you define the number of routees *or* you pass already created routees to it. If you send both parameters to the router *only* the latter will be used, i.e. `nrOfInstances` is disregarded.

It is also worth pointing out that if you define the ‘‘router‘‘ in the configuration file then this value will be used instead of any programmatically sent parameters. The decision whether to create a router at all, on the other hand, must be taken within the code, i.e. you cannot make something a router by external configuration alone (see below for details).

Once you have the router actor it is just to send messages to it as you would to any actor:

```
router.tell(new MyMsg());
```

The router will apply its behavior to the message it receives and forward it to the routees.

Remotely Deploying Routees

In addition to being able to supply looked-up remote actors as routees, you can make the router deploy its created children on a set of remote hosts; this will be done in round-robin fashion. In order to do that, wrap the router configuration in a `RemoteRouterConfig`, attaching the remote addresses of the nodes to deploy to. Naturally, this requires you to include the `akka-remote` module on your classpath:

```
Address addr1 = new Address("akka", "remotesys", "otherhost", 1234);
Address addr2 = AddressFromURIString.parse("akka://othersys@anotherhost:1234");
Address[] addresses = new Address[] { addr1, addr2 };
ActorRef routerRemote = system.actorOf(new Props(ExampleActor.class)
  .withRouter(new RemoteRouterConfig(new RoundRobinRouter(5), addresses)));
```

5.9.2 How Routing is Designed within Akka

Routers behave like single actors, but they should also not hinder scalability. This apparent contradiction is solved by making routers be represented by a special `RoutedActorRef`, which dispatches incoming messages destined for the routees without actually invoking the router actor’s behavior (and thus avoiding its mailbox; the single router actor’s task is to manage all aspects related to the lifecycle of the routees). This means that the code which decides which route to take is invoked concurrently from all possible senders and hence must be thread-safe, it cannot live the simple and happy life of code within an actor.

There is one part in the above paragraph which warrants some more background explanation: Why does a router need a “head” which is actual parent to all the routees? The initial design tried to side-step this issue, but location transparency as well as mandatory parental supervision required a redesign. Each of the actors which the router spawns must have its unique identity, which translates into a unique actor path. Since the router has only one given name in its parent’s context, another level in the name space is needed, which according to the addressing semantics implies the existence of an actor with the router’s name. This is not only necessary for the internal messaging involved in creating, restarting and terminating actors, it is also needed when the pooled actors need to converse with other actors and receive replies in a deterministic fashion. Since each actor knows its own external representation as well as that of its parent, the routees decide where replies should be sent when reacting to a message:

```
getSender().tell("reply", getContext().parent()); // replies go to router
```

```
getSender().tell("reply", getSelf()); // replies go to this actor
```

It is apparent now why routing needs to be enabled in code rather than being possible to “bolt on” later: whether or not an actor is routed means a change to the actor hierarchy, changing the actor paths of all children of the router. The routees especially do need to know that they are routed to in order to choose the sender reference for any messages they dispatch as shown above.

5.9.3 Routers vs. Supervision

As explained in the previous section, routers create new actor instances as children of the "head" router, who therefore also is their supervisor. The supervisor strategy of this actor can be configured by means of the `RouterConfig.supervisorStrategy` property, which is supported for all built-in router types. It defaults to "always escalate", which leads to the application of the router's parent's supervision directive to all children of the router uniformly (i.e. not only the one which failed). It should be mentioned that the router overrides the default behavior of terminating all children upon restart, which means that a restart—while re-creating them—does not have an effect on the number of actors in the pool.

Setting the strategy is easily done:

```
final SupervisorStrategy strategy = new OneForOneStrategy(5, Duration.parse("1 minute"),
    new Class<?>[] { Exception.class });
final ActorRef router = system.actorOf(new Props(MyActor.class)
    .withRouter(new RoundRobinRouter(5).withSupervisorStrategy(strategy)));
```

Another potentially useful approach is to give the router the same strategy as its parent, which effectively treats all actors in the pool as if they were direct children of their grand-parent instead.

5.9.4 Router usage

In this section we will describe how to use the different router types. First we need to create some actors that will be used in the examples:

```
public class PrintlnActor extends UntypedActor {
  public void onReceive(Object msg) {
    System.out.println(String.format("Received message '%s' in actor %s", msg, getSelf().path().na
  }
}
```

and

```
public class FibonacciActor extends UntypedActor {
  public void onReceive(Object msg) {
    if (msg instanceof FibonacciNumber) {
      FibonacciNumber fibonacciNumber = (FibonacciNumber) msg;
      getSender().tell(fibonacci(fibonacciNumber.getNbr()));
    } else {
      unhandled(msg);
    }
  }

  private int fibonacci(int n) {
    return fib(n, 1, 0);
  }

  private int fib(int n, int b, int a) {
    if (n == 0)
      return a;
    // recursion
    return fib(n - 1, a + b, b);
  }

  public static class FibonacciNumber implements Serializable {
    private static final long serialVersionUID = 1L;
    private final int nbr;

    public FibonacciNumber(int nbr) {
      this.nbr = nbr;
    }
```

```
    public int getNbr() {
      return nbr;
    }

  }
}
```

RoundRobinRouter

Routes in a round-robin fashion to its routees. Code example:

```
ActorRef roundRobinRouter = getContext().actorOf(
    new Props(PrintlnActor.class).withRouter(new RoundRobinRouter(5)), "router");
for (int i = 1; i <= 10; i++) {
  roundRobinRouter.tell(i, getSelf());
}
```

When run you should see a similar output to this:

```
Received message '1' in actor $b
Received message '2' in actor $c
Received message '3' in actor $d
Received message '6' in actor $b
Received message '4' in actor $e
Received message '8' in actor $d
Received message '5' in actor $f
Received message '9' in actor $e
Received message '10' in actor $f
Received message '7' in actor $c
```

If you look closely to the output you can see that each of the routees received two messages which is exactly what you would expect from a round-robin router to happen. (The name of an actor is automatically created in the format `$letter` unless you specify it - hence the names printed above.)

RandomRouter

As the name implies this router type selects one of its routees randomly and forwards the message it receives to this routee. This procedure will happen each time it receives a message. Code example:

```
ActorRef randomRouter = getContext().actorOf(new Props(PrintlnActor.class).withRouter(new RandomR
    "router");
for (int i = 1; i <= 10; i++) {
  randomRouter.tell(i, getSelf());
}
```

When run you should see a similar output to this:

```
Received message '1' in actor $e
Received message '2' in actor $c
Received message '4' in actor $b
Received message '5' in actor $d
Received message '3' in actor $e
Received message '6' in actor $c
Received message '7' in actor $d
Received message '8' in actor $e
Received message '9' in actor $d
Received message '10' in actor $d
```

The result from running the random router should be different, or at least random, every time you run it. Try to run it a couple of times to verify its behavior if you don't trust us.

SmallestMailboxRouter

A Router that tries to send to the non-suspended routee with fewest messages in mailbox. The selection is done in this order:

- pick any idle routee (not processing message) with empty mailbox
- pick any routee with empty mailbox
- pick routee with fewest pending messages in mailbox
- pick any remote routee, remote actors are consider lowest priority, since their mailbox size is unknown

Code example:

```
ActorRef smallestMailboxRouter = getContext().actorOf(
    new Props(PrintlnActor.class).withRouter(new SmallestMailboxRouter(5)), "router");
for (int i = 1; i <= 10; i++) {
  smallestMailboxRouter.tell(i, getSelf());
}
```

BroadcastRouter

A broadcast router forwards the message it receives to *all* its routees. Code example:

```
ActorRef broadcastRouter = getContext().actorOf(new Props(PrintlnActor.class).withRouter(new Broa
    "router");
broadcastRouter.tell("this is a broadcast message", getSelf());
```

When run you should see a similar output to this:

```
Received message 'this is a broadcast message' in actor $f
Received message 'this is a broadcast message' in actor $d
Received message 'this is a broadcast message' in actor $e
Received message 'this is a broadcast message' in actor $c
Received message 'this is a broadcast message' in actor $b
```

As you can see here above each of the routees, five in total, received the broadcast message.

ScatterGatherFirstCompletedRouter

The ScatterGatherFirstCompletedRouter will send the message on to all its routees as a future. It then waits for first result it gets back. This result will be sent back to original sender. Code example:

```
ActorRef scatterGatherFirstCompletedRouter = getContext().actorOf(
    new Props(FibonacciActor.class).withRouter(new ScatterGatherFirstCompletedRouter(5, Duration
        .parse("2 seconds"))), "router");
Timeout timeout = new Timeout(Duration.parse("5 seconds"));
Future<Object> futureResult = akka.pattern.Patterns.ask(scatterGatherFirstCompletedRouter,
    new FibonacciActor.FibonacciNumber(10), timeout);
int result = (Integer) Await.result(futureResult, timeout.duration());
```

When run you should see this:

```
The result of calculating Fibonacci for 10 is 55
```

From the output above you can't really see that all the routees performed the calculation, but they did! The result you see is from the first routee that returned its calculation to the router.

5.9.5 Broadcast Messages

There is a special type of message that will be sent to all routees regardless of the router. This message is called `Broadcast` and is used in the following manner:

```
router.tell(new Broadcast("Watch out for Davy Jones' locker"));
```

Only the actual message is forwarded to the routees, i.e. "Watch out for Davy Jones' locker" in the example above. It is up to the routee implementation whether to handle the broadcast message or not.

5.9.6 Dynamically Resizable Routers

All routers can be used with a fixed number of routees or with a resize strategy to adjust the number of routees dynamically.

This is an example of how to create a resizable router that is defined in configuration:

```
akka.actor.deployment {
  /router2 {
    router = round-robin
    resizer {
      lower-bound = 2
      upper-bound = 15
    }
  }
}
```

```
ActorRef router2 = system.actorOf(new Props(ExampleActor.class).withRouter(new FromConfig()), "ro
```

Several more configuration options are available and described in `akka.actor.deployment.default.resizer` section of the reference *Configuration*.

This is an example of how to programmatically create a resizable router:

```
int lowerBound = 2;
int upperBound = 15;
DefaultResizer resizer = new DefaultResizer(lowerBound, upperBound);
ActorRef router3 = system.actorOf(new Props(ExampleActor.class).withRouter(new RoundRobinRouter(n
```

It is also worth pointing out that if you define the ''router'' in the configuration file then this value will be used instead of any programmatically sent parameters.

Note: Resizing is triggered by sending messages to the actor pool, but it is not completed synchronously; instead a message is sent to the "head" `Router` to perform the size change. Thus you cannot rely on resizing to instantaneously create new workers when all others are busy, because the message just sent will be queued to the mailbox of a busy actor. To remedy this, configure the pool to use a balancing dispatcher, see Configuring Dispatchers for more information.

5.9.7 Custom Router

You can also create your own router should you not find any of the ones provided by Akka sufficient for your needs. In order to roll your own router you have to fulfill certain criteria which are explained in this section.

The router created in this example is a simple vote counter. It will route the votes to specific vote counter actors. In this case we only have two parties the Republicans and the Democrats. We would like a router that forwards all democrat related messages to the Democrat actor and all republican related messages to the Republican actor.

We begin with defining the class:

```
public static class VoteCountRouter extends CustomRouterConfig {

  @Override public String routerDispatcher() {
    return Dispatchers.DefaultDispatcherId();
  }

  @Override public SupervisorStrategy supervisorStrategy() {
    return SupervisorStrategy.defaultStrategy();
  }

  // crRoute ...

}
```

The next step is to implement the `createCustomRoute` method in the class just defined:

```
@Override
public CustomRoute createCustomRoute(Props props, RouteeProvider routeeProvider) {
  final ActorRef democratActor = routeeProvider.context().actorOf(new Props(DemocratActor.class),
  final ActorRef republicanActor = routeeProvider.context().actorOf(new Props(RepublicanActor.cla
  List<ActorRef> routees = Arrays.asList(new ActorRef[] { democratActor, republicanActor });

  routeeProvider.registerRoutees(routees);

  return new CustomRoute() {
    @Override
    public Iterable<Destination> destinationsFor(ActorRef sender, Object msg) {
      switch ((Message) msg) {
      case DemocratVote:
      case DemocratCountResult:
        return Arrays.asList(new Destination[] { new Destination(sender, democratActor) });
      case RepublicanVote:
      case RepublicanCountResult:
        return Arrays.asList(new Destination[] { new Destination(sender, republicanActor) });
      default:
        throw new IllegalArgumentException("Unknown message: " + msg);
      }
    }
  };
}
```

As you can see above we start off by creating the routees and put them in a collection.

Make sure that you don't miss to implement the line below as it is *really* important. It registers the routees internally and failing to call this method will cause a `ActorInitializationException` to be thrown when the router is used. Therefore always make sure to do the following in your custom router:

```
routeeProvider.registerRoutees(routees);
```

The routing logic is where your magic sauce is applied. In our example it inspects the message types and forwards to the correct routee based on this:

```
return new CustomRoute() {
  @Override
  public Iterable<Destination> destinationsFor(ActorRef sender, Object msg) {
    switch ((Message) msg) {
    case DemocratVote:
    case DemocratCountResult:
      return Arrays.asList(new Destination[] { new Destination(sender, democratActor) });
    case RepublicanVote:
    case RepublicanCountResult:
      return Arrays.asList(new Destination[] { new Destination(sender, republicanActor) });
    default:
      throw new IllegalArgumentException("Unknown message: " + msg);
```

```
    }
  }
};
```

As you can see above what's returned in the `CustomRoute` function, which defines the mapping from incoming sender/message to a `List` of `Destination(sender, routee)`. The sender is what "parent" the routee should see - changing this could be useful if you for example want another actor than the original sender to intermediate the result of the routee (if there is a result). For more information about how to alter the original sender we refer to the source code of ScatterGatherFirstCompletedRouter

All in all the custom router looks like this:

```
enum Message {
  DemocratVote, DemocratCountResult, RepublicanVote, RepublicanCountResult
}


public static class DemocratActor extends UntypedActor {
  int counter = 0;

  public void onReceive(Object msg) {
    switch ((Message) msg) {
    case DemocratVote:
      counter++;
      break;
    case DemocratCountResult:
      getSender().tell(counter, getSelf());
      break;
    default:
      unhandled(msg);
    }
  }
}

public static class RepublicanActor extends UntypedActor {
  int counter = 0;

  public void onReceive(Object msg) {
    switch ((Message) msg) {
    case RepublicanVote:
      counter++;
      break;
    case RepublicanCountResult:
      getSender().tell(counter, getSelf());
      break;
    default:
      unhandled(msg);
    }
  }
}


public static class VoteCountRouter extends CustomRouterConfig {

  @Override public String routerDispatcher() {
    return Dispatchers.DefaultDispatcherId();
  }

  @Override public SupervisorStrategy supervisorStrategy() {
    return SupervisorStrategy.defaultStrategy();
  }

  @Override
```

```
  public CustomRoute createCustomRoute(Props props, RouteeProvider routeeProvider) {
    final ActorRef democratActor = routeeProvider.context().actorOf(new Props(DemocratActor.class)
    final ActorRef republicanActor = routeeProvider.context().actorOf(new Props(RepublicanActor.cl
    List<ActorRef> routees = Arrays.asList(new ActorRef[] { democratActor, republicanActor });

    routeeProvider.registerRoutees(routees);

    return new CustomRoute() {
      @Override
      public Iterable<Destination> destinationsFor(ActorRef sender, Object msg) {
        switch ((Message) msg) {
        case DemocratVote:
        case DemocratCountResult:
          return Arrays.asList(new Destination[] { new Destination(sender, democratActor) });
        case RepublicanVote:
        case RepublicanCountResult:
          return Arrays.asList(new Destination[] { new Destination(sender, republicanActor) });
        default:
          throw new IllegalArgumentException("Unknown message: " + msg);
        }
      }
    };
  }

}
```

If you are interested in how to use the VoteCountRouter it looks like this:

```
@Test
public void countVotesAsIntendedNotAsInFlorida() throws Exception {
  ActorRef routedActor = system.actorOf(new Props().withRouter(new VoteCountRouter()));
  routedActor.tell(DemocratVote);
  routedActor.tell(DemocratVote);
  routedActor.tell(RepublicanVote);
  routedActor.tell(DemocratVote);
  routedActor.tell(RepublicanVote);
  Timeout timeout = new Timeout(Duration.parse("1 seconds"));
  Future<Object> democratsResult = ask(routedActor, DemocratCountResult, timeout);
  Future<Object> republicansResult = ask(routedActor, RepublicanCountResult, timeout);

  assertEquals(3, Await.result(democratsResult, timeout.duration()));
  assertEquals(2, Await.result(republicansResult, timeout.duration()));
}
```

Configured Custom Router

It is possible to define configuration properties for custom routers. In the `router` property of the deployment configuration you define the fully qualified class name of the router class. The router class must extend `akka.routing.CustomRouterConfig` and have a constructor with one `com.typesafe.config.Config` parameter. The deployment section of the configuration is passed to the constructor.

Custom Resizer

A router with dynamically resizable number of routees is implemented by providing a `akka.routing.Resizer` in `resizer` method of the `RouterConfig`. See `akka.routing.DefaultResizer` for inspiration of how to write your own resize strategy.

5.9.8 Configuring Dispatchers

The dispatcher for created children of the router will be taken from `Props` as described in *Dispatchers (Java)*. For a dynamic pool it makes sense to configure the `BalancingDispatcher` if the precise routing is not so important (i.e. no consistent hashing or round-robin is required); this enables newly created routees to pick up work immediately by stealing it from their siblings. Note that you can **not** use a `BalancingDispatcher` as a **Router Dispatcher**. (You can however use it for the **Routees**)

The "head" router cannot always run on the same dispatcher, because it does not process the same type of messages, hence this special actor does not use the dispatcher configured in `Props`, but takes the `routerDispatcher` from the `RouterConfig` instead, which defaults to the actor system's default dispatcher. All standard routers allow setting this property in their constructor or factory method, custom routers have to implement the method in a suitable way.

```
final ActorRef router = system.actorOf(new Props(MyActor.class)
  .withRouter(new RoundRobinRouter(5).withDispatcher("head")) // "head" router runs on "head" disp
  .withDispatcher("workers")); // MyActor "workers" run on "workers" dispatcher
```

Note: It is not allowed to configure the `routerDispatcher` to be a `BalancingDispatcher` since the messages meant for the special router actor cannot be processed by any other actor.

At first glance there seems to be an overlap between the `BalancingDispatcher` and Routers, but they complement each other. The balancing dispatcher is in charge of running the actors while the routers are in charge of deciding which message goes where. A router can also have children that span multiple actor systems, even remote ones, but a dispatcher lives inside a single actor system.

When using a `RoundRobinRouter` with a `BalancingDispatcher` there are some configuration settings to take into account.

- There can only be `nr-of-instances` messages being processed at the same time no matter how many threads are configured for the `BalancingDispatcher`.
- Having `throughput` set to a low number makes no sense since you will only be handing off to another actor that processes the same `MailBox` as yourself, which can be costly. Either the message just got into the mailbox and you can receive it as well as anybody else, or everybody else is busy and you are the only one available to receive the message.
- Resizing the number of routees only introduce inertia, since resizing is performed at specified intervals, but work stealing is instantaneous.

5.10 Remoting (Java)

For an introduction of remoting capabilities of Akka please see *Location Transparency*.

5.10.1 Preparing your ActorSystem for Remoting

The Akka remoting is a separate jar file. Make sure that you have the following dependency in your project:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-remote</artifactId>
  <version>2.0.4</version>
</dependency>
```

To enable remote capabilities in your Akka project you should, at a minimum, add the following changes to your `application.conf` file:

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    transport = "akka.remote.netty.NettyRemoteTransport"
    netty {
      hostname = "127.0.0.1"
      port = 2552
    }
 }
}
```

As you can see in the example above there are four things you need to add to get started:

- Change provider from `akka.actor.LocalActorRefProvider` to `akka.remote.RemoteActorRefProvider`
- Add host name - the machine you want to run the actor system on; this host name is exactly what is passed to remote systems in order to identify this system and consequently used for connecting back to this system if need be, hence set it to a reachable IP address or resolvable name in case you want to communicate across the network.
- Add port number - the port the actor system should listen on, set to 0 to have it chosen automatically

---

Note: The port number needs to be unique for each actor system on the same machine even if the actor systems have different names. This is because each actor system has its own network subsystem listening for connections and handling messages as not to interfere with other actor systems.

The example above only illustrates the bare minimum of properties you have to add to enable remoting. There are lots of more properties that are related to remoting in Akka. We refer to the following reference file for more information:

```
####################################
# Akka Remote Reference Config File #
####################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

# comments about akka.actor settings left out where they are already in akka-
# actor.jar, because otherwise they would be repeated in config rendering.

akka {

  actor {

    serializers {
      proto = "akka.serialization.ProtobufSerializer"
    }


    serialization-bindings {
      # Since com.google.protobuf.Message does not extend Serializable but GeneratedMessage
      # does, need to use the more specific one here in order to avoid ambiguity
      "com.google.protobuf.GeneratedMessage" = proto
    }

    deployment {

      default {
```

```
      # if this is set to a valid remote address, the named actor will be deployed
      # at that node e.g. "akka://sys@host:port"
      remote = ""

      target {

        # A list of hostnames and ports for instantiating the children of a
        # non-direct router
        #   The format should be on "akka://sys@host:port", where:
        #    - sys is the remote actor system name
        #    - hostname can be either hostname or IP address the remote actor
        #      should connect to
        #    - port should be the port for the remote server on the other node
        # The number of actor instances to be spawned is still taken from the
        # nr-of-instances setting as for local routers; the instances will be
        # distributed round-robin among the given nodes.
        nodes = []

      }
    }
  }
}

remote {

  # Which implementation of akka.remote.RemoteTransport to use
  # default is a TCP-based remote transport based on Netty
  transport = "akka.remote.netty.NettyRemoteTransport"

  # Enable untrusted mode for full security of server managed actors, allows
  # untrusted clients to connect.
  untrusted-mode = off

  # Timeout for ACK of cluster operations, like checking actor out etc.
  remote-daemon-ack-timeout = 30s

  # If this is "on", Akka will log all inbound messages at DEBUG level, if off then they are not
  log-received-messages = off

  # If this is "on", Akka will log all outbound messages at DEBUG level, if off then they are no
  log-sent-messages = off

  # If this is "on", Akka will log all RemoteLifeCycleEvents at the level defined for each, if o
  log-remote-lifecycle-events = off

  # Each property is annotated with (I) or (O) or (I&O), where I stands for "inbound" and O for
  # The NettyRemoteTransport always starts the server role to allow inbound connections, and it
  # active client connections whenever sending to a destination which is not yet connected; if o
  # it reuses inbound connections for replies, which is called a passive client connection (i.e
  # to client).
  netty {

    # (O) In case of increased latency / overflow how long should we wait (blocking the sender)
    # until we deem the send to be cancelled?
    # 0 means "never backoff", any positive number will indicate time to block at most.
    backoff-timeout = 0ms

    # (I&O) Generate your own with '$AKKA_HOME/scripts/generate_config_with_secure_cookie.sh'
    # or using 'akka.util.Crypt.generateSecureCookie'
    secure-cookie = ""

    # (I) Should the remote server require that its peers share the same secure-cookie
    # (defined in the 'remote' section)?
```

```
    require-cookie = off

    # (I) Reuse inbound connections for outbound messages
    use-passive-connections = on

    # (I) The hostname or ip to bind the remoting to,
    # InetAddress.getLocalHost.getHostAddress is used if empty
    hostname = ""

    # (I) The default remote server port clients should connect to.
    # Default is 2552 (AKKA), use 0 if you want a random available port
    # This port needs to be unique for each actor system on the same machine.
    port = 2552

    # (O) The address of a local network interface (IP Address) to bind to when creating
    # outbound connections. Set to "" or "auto" for automatic selection of local address.
    outbound-local-address = "auto"

    # (I&O) Increase this if you want to be able to send messages with large payloads
    message-frame-size = 1 MiB

    # (O) Timeout duration
    connection-timeout = 120s

    # (I) Sets the size of the connection backlog
    backlog = 4096

    # (I) Length in akka.time-unit how long core threads will be kept alive if idling
    execution-pool-keepalive = 60s

    # (I) Size of the core pool of the remote execution unit
    execution-pool-size = 4

    # (I) Maximum channel size, 0 for off
    max-channel-memory-size = 0b

    # (I) Maximum total size of all channels, 0 for off
    max-total-memory-size = 0b

    # (O) Time between reconnect attempts for active clients
    reconnect-delay = 5s

    # (O) Read inactivity period (lowest resolution is seconds)
    # after which active client connection is shutdown;
    # will be re-established in case of new communication requests.
    # A value of 0 will turn this feature off
    read-timeout = 0s

    # (O) Write inactivity period (lowest resolution is seconds)
    # after which a heartbeat is sent across the wire.
    # A value of 0 will turn this feature off
    write-timeout = 10s

    # (O) Inactivity period of both reads and writes (lowest resolution is seconds)
    # after which active client connection is shutdown;
    # will be re-established in case of new communication requests
    # A value of 0 will turn this feature off
    all-timeout = 0s

    # (O) Maximum time window that a client should try to reconnect for
    reconnection-time-window = 600s

    # (I&O) Used to configure the number of I/O worker threads on server sockets
```

```
      server-socket-worker-pool {
        # Min number of threads to cap factor-based number to
        pool-size-min = 2

        # The pool size factor is used to determine thread pool size
        # using the following formula: ceil(available processors * factor).
        # Resulting size is then bounded by the pool-size-min and
        # pool-size-max values.
        pool-size-factor = 2.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 128
      }

      # (I&O) Used to configure the number of I/O worker threads on client sockets
      client-socket-worker-pool {
        # Min number of threads to cap factor-based number to
        pool-size-min = 2

        # The pool size factor is used to determine thread pool size
        # using the following formula: ceil(available processors * factor).
        # Resulting size is then bounded by the pool-size-min and
        # pool-size-max values.
        pool-size-factor = 2.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 128
      }
    }

    # The dispatcher used for the system actor "network-event-sender"
    network-event-sender-dispatcher {
      executor = thread-pool-executor
      type = PinnedDispatcher
    }
  }
}
```

5.10.2 Looking up Remote Actors

`actorFor(path)` will obtain an `ActorRef` to an Actor on a remote node:

```
ActorRef actor = context.actorFor("akka://app@10.0.0.1:2552/user/serviceA/retrieval");
```

As you can see from the example above the following pattern is used to find an `ActorRef` on a remote node:

```
akka://<actorsystemname>@<hostname>:<port>/<actor path>
```

For more details on how actor addresses and paths are formed and used, please refer to *Actor References, Paths and Addresses*.

5.10.3 Creating Actors Remotely

The configuration below instructs the system to deploy the actor "retrieval" on the specific host "app@10.0.0.1". The "app" in this case refers to the name of the `ActorSystem` (only showing deployment section):

```
akka {
  actor {
    deployment {
      /serviceA/retrieval {
        remote = "akka://app@10.0.0.1:2552"
```

```
      }
    }
  }
}
```

Logical path lookup is supported on the node you are on, i.e. to use the actor created above you would do the following:

```
ActorRef a1 = getContext().actorFor("/serviceA/retrieval");
```

This will obtain an `ActorRef` on a remote node:

```
ActorRef a2 = getContext().actorFor("akka://app@10.0.0.1:2552/user/serviceA/retrieval");
```

As you can see from the example above the following pattern is used to find an `ActorRef` on a remote node:

```
akka://<actorsystemname>@<hostname>:<port>/<actor path>
```

Programmatic Remote Deployment

To allow dynamically deployed systems, it is also possible to include deployment configuration in the `Props` which are used to create an actor: this information is the equivalent of a deployment section from the configuration file, and if both are given, the external configuration takes precedence.

With these imports:

```
import akka.actor.ActorRef;
import akka.actor.Address;
import akka.actor.AddressFromURIString;
import akka.actor.Deploy;
import akka.actor.Props;
import akka.actor.ActorSystem;
import akka.remote.RemoteScope;
```

and a remote address like this:

```
Address addr = new Address("akka", "sys", "host", 1234);
addr = AddressFromURIString.parse("akka://sys@host:1234"); // the same
```

you can advise the system to create a child on that remote node like so:

```
ActorRef ref = system.actorOf(new Props(RemoteDeploymentDocSpec.Echo.class).withDeploy(new Deploy
```

5.10.4 Serialization

When using remoting for actors you must ensure that the `props` and `messages` used for those actors are serializable. Failing to do so will cause the system to behave in an unintended way.

For more information please see *Serialization (Java)*

5.10.5 Routers with Remote Destinations

It is absolutely feasible to combine remoting with *Routing (Java)*. This is also done via configuration:

```
akka {
  actor {
    deployment {
      /serviceA/aggregation {
        router = "round-robin"
        nr-of-instances = 10
        target {
```

```
          nodes = ["akka://app@10.0.0.2:2552", "akka://app@10.0.0.3:2552"]
        }
      }
    }
  }
}
```

This configuration setting will clone the actor "aggregation" 10 times and deploy it evenly distributed across the two given target nodes.

5.10.6 Description of the Remoting Sample

There is a more extensive remote example that comes with the Akka distribution. Please have a look here for more information: Remote Sample This sample demonstrates both, remote deployment and look-up of remote actors. First, let us have a look at the common setup for both scenarios (this is `common.conf`):

```
akka {

  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }

  remote {
    netty {
      hostname = "127.0.0.1"
    }
  }
}
```

This enables the remoting by installing the `RemoteActorRefProvider` and chooses the default remote transport. All other options will be set specifically for each show case.

Note: Be sure to replace the default IP 127.0.0.1 with the real address the system is reachable by if you deploy onto multiple machines!

Remote Lookup

In order to look up a remote actor, that one must be created first. For this purpose, we configure an actor system to listen on port 2552 (this is a snippet from `application.conf`):

```
calculator {
  include "common"

  akka {
    remote.netty.port = 2552
  }
}
```

Then the actor must be created. For all code which follows, assume these imports:

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.kernel.Bootable;
import com.typesafe.config.ConfigFactory;
```

The actor doing the work will be this one:

```
public class JSimpleCalculatorActor extends UntypedActor {
    @Override
    public void onReceive(Object message) {

        if (message instanceof Op.Add) {
            Op.Add add = (Op.Add) message;
            System.out.println("Calculating " + add.getN1() + " + " + add.getN2());
            getSender().tell(new Op.AddResult(add.getN1(), add.getN2(), add.getN1() + add.getN2()

        } else if (message instanceof Op.Subtract) {
            Op.Subtract subtract = (Op.Subtract) message;
            System.out.println("Calculating " + subtract.getN1() + " - " + subtract.getN2());
            getSender().tell(new Op.SubtractResult(subtract.getN1(), subtract.getN2(), subtract.ge

        } else {
            unhandled(message);
        }
    }
}
```

and we start it within an actor system using the above configuration

```
public class JCalculatorApplication implements Bootable {
    private ActorSystem system;

    public JCalculatorApplication() {
        system = ActorSystem.create("CalculatorApplication", ConfigFactory.load().getConfig("calcu
        ActorRef actor = system.actorOf(new Props(JSimpleCalculatorActor.class), "simpleCalculato

    }

    @Override
    public void startup() {
    }

    @Override
    public void shutdown() {
        system.shutdown();
    }
}
```

With the service actor up and running, we may look it up from another actor system, which will be configured to use port 2553 (this is a snippet from `application.conf`).

```
remotelookup {
  include "common"

  akka {
    remote.netty.port = 2553
  }
}
```

The actor which will query the calculator is a quite simple one for demonstration purposes

```
public class JLookupActor extends UntypedActor {

    @Override
    public void onReceive(Object message) throws Exception {

        if (message instanceof InternalMsg.MathOpMsg) {

            // send message to server actor
            InternalMsg.MathOpMsg msg = (InternalMsg.MathOpMsg) message;
            msg.getActor().tell(msg.getMathOp(), getSelf());
```

```
        } else if (message instanceof Op.MathResult) {

            // receive reply from server actor

            if (message instanceof Op.AddResult) {
                Op.AddResult result = (Op.AddResult) message;
                System.out.println("Add result: " + result.getN1() + " + " +
                        result.getN2() + " = " + result.getResult());

            } else if (message instanceof Op.SubtractResult) {
                Op.SubtractResult result = (Op.SubtractResult) message;
                System.out.println("Sub result: " + result.getN1() + " - " +
                        result.getN2() + " = " + result.getResult());
            }
        } else {
          unhandled(message);
        }
    }
}
```

and it is created from an actor system using the aforementioned client's config.

```
public class JLookupApplication implements Bootable {
    private ActorSystem system;
    private ActorRef actor;
    private ActorRef remoteActor;

    public JLookupApplication() {
        system = ActorSystem.create("LookupApplication", ConfigFactory.load().getConfig("remotelo
        actor = system.actorOf(new Props(JLookupActor.class));
        remoteActor = system.actorFor("akka://CalculatorApplication@127.0.0.1:2552/user/simpleCal
    }

    public void doSomething(Op.MathOp mathOp) {
        actor.tell(new InternalMsg.MathOpMsg(remoteActor, mathOp));
    }

    @Override
    public void startup() {
    }

    @Override
    public void shutdown() {
        system.shutdown();
    }
}
```

Requests which come in via `doSomething` will be sent to the client actor along with the reference which was looked up earlier. Observe how the actor system name using in `actorFor` matches the remote system's name, as do IP and port number. Top-level actors are always created below the `"/user"` guardian, which supervises them.

Remote Deployment

Creating remote actors instead of looking them up is not visible in the source code, only in the configuration file. This section is used in this scenario (this is a snippet from `application.conf`):

```
remotecreation {
  include "common"

  akka {
    actor {
```

```
      deployment {
        /advancedCalculator {
          remote = "akka://CalculatorApplication@127.0.0.1:2552"
        }
      }
    }

    remote.netty.port = 2554
  }
}
```

For all code which follows, assume these imports:

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.kernel.Bootable;
import com.typesafe.config.ConfigFactory;
```

The server actor can multiply or divide numbers:

```
public class JAdvancedCalculatorActor extends UntypedActor {
    @Override
    public void onReceive(Object message) throws Exception {

        if (message instanceof Op.Multiply) {
            Op.Multiply multiply = (Op.Multiply) message;
            System.out.println("Calculating " + multiply.getN1() + " * " + multiply.getN2());
            getSender().tell(new Op.MultiplicationResult(multiply.getN1(), multiply.getN2(), mult

        } else if (message instanceof Op.Divide) {
            Op.Divide divide = (Op.Divide) message;
            System.out.println("Calculating " + divide.getN1() + " / " + divide.getN2());
            getSender().tell(new Op.DivisionResult(divide.getN1(), divide.getN2(), divide.getN1()

        } else {
            unhandled(message);
        }
    }
}
```

The client actor looks like in the previous example

```
public class JCreationActor extends UntypedActor {
    private static final NumberFormat formatter = new DecimalFormat("#0.00");

    @Override
    public void onReceive(Object message) throws Exception {

        if (message instanceof InternalMsg.MathOpMsg) {
            // forward math op to server actor
            InternalMsg.MathOpMsg msg = (InternalMsg.MathOpMsg) message;
            msg.getActor().tell(msg.getMathOp(), getSelf());

        } else if (message instanceof Op.MathResult) {

            // receive reply from server actor

            if (message instanceof Op.MultiplicationResult) {
                Op.MultiplicationResult result = (Op.MultiplicationResult) message;
                System.out.println("Mul result: " + result.getN1() + " * " +
                        result.getN2() + " = " + result.getResult());
```

```
            } else if (message instanceof Op.DivisionResult) {
                Op.DivisionResult result = (Op.DivisionResult) message;
                System.out.println("Div result: " + result.getN1() + " / " +
                        result.getN2() + " = " + formatter.format(result.getResult()));
            }
        } else {
          unhandled(message);
        }
    }
}
```

but the setup uses only `actorOf`:

```
public class JCreationApplication implements Bootable {
    private ActorSystem system;
    private ActorRef actor;
    private ActorRef remoteActor;

    public JCreationApplication() {
        system = ActorSystem.create("CreationApplication", ConfigFactory.load().getConfig("remote
        actor = system.actorOf(new Props(JCreationActor.class));
        remoteActor = system.actorOf(new Props(JAdvancedCalculatorActor.class), "advancedCalculat
    }

    public void doSomething(Op.MathOp mathOp) {
        actor.tell(new InternalMsg.MathOpMsg(remoteActor, mathOp));
    }

    @Override
    public void startup() {
    }

    @Override
    public void shutdown() {
        system.shutdown();
    }
}
```

Observe how the name of the server actor matches the deployment given in the configuration file, which will transparently delegate the actor creation to the remote node.

Remote Events

It is possible to listen to events that occur in Akka Remote, and to subscribe/unsubscribe to there events, you simply register as listener to the below described types in on the `ActorSystem.eventStream`.

Note: To subscribe to any outbound-related events, subscribe to `RemoteClientLifeCycleEvent` To subscribe to any inbound-related events, subscribe to `RemoteServerLifeCycleEvent` To subscribe to any remote events, subscribe to `RemoteLifeCycleEvent`

To intercept when an outbound connection is disconnected, you listen to `RemoteClientDisconnected` which holds the transport used (RemoteTransport) and the outbound address that was disconnected (Address).

To intercept when an outbound connection is connected, you listen to `RemoteClientConnected` which holds the transport used (RemoteTransport) and the outbound address that was connected to (Address).

To intercept when an outbound client is started you listen to `RemoteClientStarted` which holds the transport used (RemoteTransport) and the outbound address that it is connected to (Address).

To intercept when an outbound client is shut down you listen to `RemoteClientShutdown` which holds the transport used (RemoteTransport) and the outbound address that it was connected to (Address).

To intercept when an outbound message cannot be sent, you listen to `RemoteClientWriteFailed` which holds the payload that was not written (AnyRef), the cause of the failed send (Throwable), the transport used (RemoteTransport) and the outbound address that was the destination (Address).

For general outbound-related errors, that do not classify as any of the others, you can listen to `RemoteClientError`, which holds the cause (Throwable), the transport used (RemoteTransport) and the outbound address (Address).

To intercept when an inbound server is started (typically only once) you listen to `RemoteServerStarted` which holds the transport that it will use (RemoteTransport).

To intercept when an inbound server is shut down (typically only once) you listen to `RemoteServerShutdown` which holds the transport that it used (RemoteTransport).

To intercept when an inbound connection has been established you listen to `RemoteServerClientConnected` which holds the transport used (RemoteTransport) and optionally the address that connected (Option<Address>).

To intercept when an inbound connection has been disconnected you listen to `RemoteServerClientDisconnected` which holds the transport used (RemoteTransport) and optionally the address that disconnected (Option<Address>).

To intercept when an inbound remote client has been closed you listen to `RemoteServerClientClosed` which holds the transport used (RemoteTransport) and optionally the address of the remote client that was closed (Option<Address>).

5.11 Serialization (Java)

Akka has a built-in Extension for serialization, and it is both possible to use the built-in serializers and to write your own.

The serialization mechanism is both used by Akka internally to serialize messages, and available for ad-hoc serialization of whatever you might need it for.

5.11.1 Usage

Configuration

For Akka to know which `Serializer` to use for what, you need edit your *Configuration*, in the "akka.actor.serializers"-section you bind names to implementations of the `akka.serialization.Serializer` you wish to use, like this:

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
      serializers {
        java = "akka.serialization.JavaSerializer"
        proto = "akka.serialization.ProtobufSerializer"
        myown = "akka.docs.serialization.MyOwnSerializer"
      }
    }
  }
""")
```

After you've bound names to different implementations of `Serializer` you need to wire which classes should be serialized using which `Serializer`, this is done in the "akka.actor.serialization-bindings"-section:

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
      serializers {
```

```
        java = "akka.serialization.JavaSerializer"
        proto = "akka.serialization.ProtobufSerializer"
        myown = "akka.docs.serialization.MyOwnSerializer"
      }

      serialization-bindings {
        "java.lang.String" = java
        "akka.docs.serialization.Customer" = java
        "com.google.protobuf.Message" = proto
        "akka.docs.serialization.MyOwnSerializable" = myown
        "java.lang.Boolean" = myown
      }
    }
  }
""")
```

You only need to specify the name of an interface or abstract base class of the messages. In case of ambiguity, i.e. the message implements several of the configured classes, the most specific configured class will be used, i.e. the one of which all other candidates are superclasses. If this condition cannot be met, because e.g. `java.io.Serializable` and `MyOwnSerializable` both apply and neither is a subtype of the other, a warning will be issued.

Akka provides serializers for `java.io.Serializable` and protobuf `com.google.protobuf.GeneratedMessage` by default (the latter only if depending on the akka-remote module), so normally you don't need to add configuration for that; since `com.google.protobuf.GeneratedMessage` implements `java.io.Serializable`, protobuf messages will always by serialized using the protobuf protocol unless specifically overridden. In order to disable a default serializer, map its marker type to "none":

```
akka.actor.serialization-bindings {
  "java.io.Serializable" = none
}
```

Verification

If you want to verify that your messages are serializable you can enable the following config option:

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
      serialize-messages = on
    }
  }
""")
```

Warning: We only recommend using the config option turned on when you're running tests. It is completely pointless to have it turned on in other scenarios.

If you want to verify that your `Props` are serializable you can enable the following config option:

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
      serialize-creators = on
    }
  }
""")
```

Warning: We only recommend using the config option turned on when you're running tests. It is completely pointless to have it turned on in other scenarios.

Programmatic

If you want to programmatically serialize/deserialize using Akka Serialization, here's some examples:

```
import akka.actor.*;
import akka.remote.RemoteActorRefProvider;
import akka.serialization.*;
import com.typesafe.config.*;

    ActorSystem system = ActorSystem.create("example");

    // Get the Serialization Extension
    Serialization serialization = SerializationExtension.get(system);

    // Have something to serialize
    String original = "woohoo";

    // Find the Serializer for it
    Serializer serializer = serialization.findSerializerFor(original);

    // Turn it into bytes
    byte[] bytes = serializer.toBinary(original);

    // Turn it back into an object,
    // the nulls are for the class manifest and for the classloader
    String back = (String) serializer.fromBinary(bytes);

    // Voilá!
    assertEquals(original, back);
```

For more information, have a look at the `ScalaDoc` for `akka.serialization._`

5.11.2 Customization

So, lets say that you want to create your own `Serializer`, you saw the `akka.docs.serialization.MyOwnSerializer` in the config example above?

Creating new Serializers

First you need to create a class definition of your `Serializer`, which is done by extending `akka.serialization.JSerializer`, like this:

```
import akka.actor.*;
import akka.remote.RemoteActorRefProvider;
import akka.serialization.*;
import com.typesafe.config.*;

    public static class MyOwnSerializer extends JSerializer {

      // This is whether "fromBinary" requires a "clazz" or not
      @Override public boolean includeManifest() {
          return false;
      }

      // Pick a unique identifier for your Serializer,
      // you've got a couple of billions to choose from,
      // 0 - 16 is reserved by Akka itself
      @Override public int identifier() {
          return 1234567;
      }
```

```
      // "toBinary" serializes the given object to an Array of Bytes
      @Override public byte[] toBinary(Object obj) {
        // Put the code that serializes the object here
        // ... ...
      }

      // "fromBinary" deserializes the given array,
      // using the type hint (if any, see "includeManifest" above)
      @Override public Object fromBinaryJava(byte[] bytes,
                    Class<?> clazz) {
        // Put your code that deserializes here
        // ... ...
      }
    }
```

Then you only need to fill in the blanks, bind it to a name in your *Configuration* and then list which classes that should be serialized using it.

Serializing ActorRefs

All ActorRefs are serializable using JavaSerializer, but in case you are writing your own serializer, you might want to know how to serialize and deserialize them properly, here's the magic incantation:

```
import akka.actor.*;
import akka.remote.RemoteActorRefProvider;
import akka.serialization.*;
import com.typesafe.config.*;

        // Serialize
        // (beneath toBinary)
        final Address transportAddress =
                Serialization.currentTransportAddress().value();
        String identifier;

        // If there is no transportAddress,
        // it means that either this Serializer isn't called
        // within a piece of code that sets it,
        // so either you need to supply your own,
        // or simply use the local path.
        if (transportAddress == null) identifier = theActorRef.path().toString();
        else identifier = theActorRef.path().toStringWithAddress(transportAddress);
        // Then just serialize the identifier however you like


        // Deserialize
        // (beneath fromBinary)
        final ActorRef deserializedActorRef = theActorSystem.actorFor(identifier);
        // Then just use the ActorRef
```

Note: `ActorPath.toStringWithAddress` only differs from `toString` if the address does not already have `host` and `port` components, i.e. it only inserts address information for local addresses.

This assumes that serialization happens in the context of sending a message through the remote transport. There are other uses of serialization, though, e.g. storing actor references outside of an actor application (database, durable mailbox, etc.). In this case, it is important to keep in mind that the address part of an actor's path determines how that actor is communicated with. Storing a local actor path might be the right choice if the retrieval happens in the same logical context, but it is not enough when deserializing it on a different network host: for that it would need to include the system's remote transport address. An actor system is not limited to having just one remote transport per se, which makes this question a bit more interesting.

In the general case, the local address to be used depends on the type of remote address which shall be the recipient of the serialized information. Use `ActorRefProvider.getExternalAddressFor(remoteAddr)` to query the system for the appropriate address to use when sending to `remoteAddr`:

```
public static class ExternalAddressExt implements Extension {
  private final ExtendedActorSystem system;

  public ExternalAddressExt(ExtendedActorSystem system) {
    this.system = system;
  }

  public Address getAddressFor(Address remoteAddress) {
    final scala.Option<Address> optAddr = system.provider()
        .getExternalAddressFor(remoteAddress);
    if (optAddr.isDefined()) {
      return optAddr.get();
    } else {
      throw new UnsupportedOperationException(
          "cannot send to remote address " + remoteAddress);
    }
  }
}

public static class ExternalAddress extends
    AbstractExtensionId<ExternalAddressExt> implements ExtensionIdProvider {
  public static final ExternalAddress ID = new ExternalAddress();

  public ExternalAddress lookup() {
    return ID;
  }

  public ExternalAddressExt createExtension(ExtendedActorSystem system) {
    return new ExternalAddressExt(system);
  }
}
```

This requires that you know at least which type of address will be supported by the system which will deserialize the resulting actor reference; if you have no concrete address handy you can create a dummy one for the right protocol using `new Address(protocol, "", "", 0)` (assuming that the actual transport used is as lenient as Akka's RemoteActorRefProvider).

There is a possible simplification available if you are just using the default `NettyRemoteTransport` with the `RemoteActorRefProvider`, which is enabled by the fact that this combination has just a single remote address:

```
public static class DefaultAddressExt implements Extension {
  private final ExtendedActorSystem system;

  public DefaultAddressExt(ExtendedActorSystem system) {
    this.system = system;
  }

  public Address getAddress() {
    final ActorRefProvider provider = system.provider();
    if (provider instanceof RemoteActorRefProvider) {
      return ((RemoteActorRefProvider) provider).transport().address();
    } else {
      throw new UnsupportedOperationException("need RemoteActorRefProvider");
    }
  }
}

public static class DefaultAddress extends
    AbstractExtensionId<DefaultAddressExt> implements ExtensionIdProvider {
```

```
  public static final DefaultAddress ID = new DefaultAddress();

  public DefaultAddress lookup() {
    return ID;
  }

  public DefaultAddressExt createExtension(ExtendedActorSystem system) {
    return new DefaultAddressExt(system);
  }
}
```

This solution has to be adapted once other providers are used (like the planned extensions for clustering).

Deep serialization of Actors

The current recommended approach to do deep serialization of internal actor state is to use Event Sourcing, for more reading on the topic, see these examples:

Martin Krasser on EventSourcing Part1

Martin Krasser on EventSourcing Part2

Note: Built-in API support for persisting Actors will come in a later release, see the roadmap for more info:

Akka 2.0 roadmap

5.11.3 A Word About Java Serialization

When using Java serialization without employing the `JavaSerializer` for the task, you must make sure to supply a valid `ExtendedActorSystem` in the dynamic variable `JavaSerializer.currentSystem`. This is used when reading in the representation of an `ActorRef` for turning the string representation into a real reference. `DynamicVariable` is a thread-local variable, so be sure to have it set while deserializing anything which might contain actor references.

5.12 Software Transactional Memory (Java)

5.12.1 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.

The use of STM in Akka is inspired by the concepts and views in Clojure's STM. Please take the time to read this excellent document about state in clojure and view this presentation by Rich Hickey (the genius behind Clojure).

5.12.2 Scala STM

The STM supported in Akka is ScalaSTM which will be soon included in the Scala standard library.

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.

5.12.3 Integration with Actors

In Akka we've also integrated Actors and STM in *Agents (Java)* and *Transactors (Java)*.

5.13 Agents (Java)

Agents in Akka are 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`) 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.

5.13.1 Creating and stopping Agents

Agents are created by invoking `new Agent(value, system)` passing in the Agent's initial value and a reference to the `ActorSystem` for your application. An `ActorSystem` is required to create the underlying Actors. See *Actor Systems* for more information about actor systems.

Here is an example of creating an Agent:

```
import akka.actor.ActorSystem;
import akka.agent.Agent;
```

```
ActorSystem system = ActorSystem.create("app");

Agent<Integer> agent = new Agent<Integer>(5, system);
```

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();
```

5.13.2 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.

```
import akka.japi.Function;
```

```
// send a value
agent.send(7);

// send a function
agent.send(new Function<Integer, Integer>() {
  public Integer apply(Integer i) {
    return i * 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);
```

5.13.3 Reading an Agent's value

Agents can be dereferenced (you can get an Agent's value) by calling the get method:

```
Integer 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.

5.13.4 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`:

```
import akka.util.Timeout;
import static java.util.concurrent.TimeUnit.SECONDS;
```

```
Integer result = agent.await(new Timeout(5, SECONDS));
```

5.14 Transactors (Java)

5.14.1 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.

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.

5.14.2 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.

5.14.3 Coordinated transactions

Akka provides an explicit mechanism for coordinating transactions across actors. Under the hood it uses a `CommitBarrier`, 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 ActorRef friend = null;

    public Increment() {}

    public Increment(ActorRef friend) {
        this.friend = friend;
    }

    public boolean hasFriend() {
        return friend != null;
    }

    public ActorRef getFriend() {
        return friend;
    }
}
```

```
import akka.actor.*;
import akka.transactor.*;
import scala.concurrent.stm.Ref;
import scala.concurrent.stm.japi.STM;

public class CoordinatedCounter extends UntypedActor {
    private Ref.View<Integer> count = STM.newRef(0);

    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 Runnable() {
                    public void run() {
                        STM.increment(count, 1);
                    }
                });
            }
        } else if ("GetCount".equals(incoming)) {
            getSender().tell(count.get());
        } else {
          unhandled(incoming);
        }
    }
}
```

```
import akka.actor.*;
import akka.dispatch.Await;
import static akka.pattern.Patterns.ask;
import akka.transactor.Coordinated;
import akka.util.Duration;
import akka.util.Timeout;
import static java.util.concurrent.TimeUnit.SECONDS;
```

```
ActorSystem system = ActorSystem.create("CoordinatedExample");

ActorRef counter1 = system.actorOf(new Props(CoordinatedCounter.class));
ActorRef counter2 = system.actorOf(new Props(CoordinatedCounter.class));

Timeout timeout = new Timeout(5, SECONDS);

counter1.tell(new Coordinated(new Increment(counter2), timeout));

Integer count = (Integer) Await.result(ask(counter1, "GetCount", timeout), timeout.duration());
```

To start a new coordinated transaction that you will also participate in, create a `Coordinated` object, passing in a `Timeout`:

```
Timeout timeout = new Timeout(5, SECONDS);
Coordinated coordinated = new Coordinated(timeout);
```

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(), timeout));
```

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, passing in a `java.lang.Runnable`.

```
coordinated.atomic(new Runnable() {
    public void run() {
        // do something in the coordinated transaction ...
    }
});
```

The coordinated transaction will wait for the other transactions before committing. If any of the coordinated transactions fail then they all fail.

Note: The same actor should not be added to a coordinated transaction more than once. The transaction will not be able to complete as an actor only processes a single message at a time. When processing the first message the coordinated transaction will wait for the commit barrier, which in turn needs the second message to be received to proceed.

5.14.4 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.*;
import scala.concurrent.stm.Ref;
import scala.concurrent.stm.japi.STM;

public class Counter extends UntypedTransactor {
    Ref.View<Integer> count = STM.newRef(0);

    public void atomically(Object message) {
        if (message instanceof Increment) {
            STM.increment(count, 1);
        }
    }

    @Override public boolean normally(Object message) {
        if ("GetCount".equals(message)) {
            getSender().tell(count.get());
            return true;
        } else return false;
    }
}
```

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.

Here's an example of coordinating an increment, using an untyped transactor, similar to the explicitly coordinated example above.

```
import akka.actor.*;
import akka.transactor.*;
import java.util.Set;
import scala.concurrent.stm.Ref;
import scala.concurrent.stm.japi.STM;

public class FriendlyCounter extends UntypedTransactor {
    Ref.View<Integer> count = STM.newRef(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();
```

```
    }

    public void atomically(Object message) {
        if (message instanceof Increment) {
            STM.increment(count, 1);
        }
    }

    @Override public boolean normally(Object message) {
        if ("GetCount".equals(message)) {
            getSender().tell(count.get());
            return true;
        } else return false;
    }
}
```

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.

5.15 Building Finite State Machine Actors (Java)

5.15.1 Overview

The FSM (Finite State Machine) pattern is best described in the Erlang design principles. In short, it can be seen 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'.*

While the Scala programming language enables the formulation of a nice internal DSL (domain specific language) for formulating finite state machines (see *FSM*), Java's verbosity does not lend itself well to the same approach. This chapter describes ways to effectively achieve the same separation of concerns through self-discipline.

5.15.2 How State should be Handled

All mutable fields (or transitively mutable data structures) referenced by the FSM actor's implementation should be collected in one place and only mutated using a small well-defined set of methods. One way to achieve this is to assemble all mutable state in a superclass which keeps it private and offers protected methods for mutating it.

```
import java.util.ArrayList;
import java.util.List;
import akka.actor.ActorRef;
```

```
static abstract class MyFSMBase extends UntypedActor {

  /*
   * This is the mutable state of this state machine.
   */
  protected enum State {
    IDLE, ACTIVE;
  }
```

```
  private State state = State.IDLE;
  private ActorRef target;
  private List<Object> queue;

  /*
   * Then come all the mutator methods:
   */
  protected void init(ActorRef target) {
    this.target = target;
    queue = new ArrayList<Object>();
  }

  protected void setState(State s) {
    if (state != s) {
      transition(state, s);
      state = s;
    }
  }

  protected void enqueue(Object o) {
    if (queue != null)
      queue.add(o);
  }

  protected List<Object> drainQueue() {
    final List<Object> q = queue;
    if (q == null)
      throw new IllegalStateException("drainQueue(): not yet initialized");
    queue = new ArrayList<Object>();
    return q;
  }

  /*
   * Here are the interrogation methods:
   */
  protected boolean isInitialized() {
    return target != null;
  }

  protected State getState() {
    return state;
  }

  protected ActorRef getTarget() {
    if (target == null)
      throw new IllegalStateException("getTarget(): not yet initialized");
    return target;
  }

  /*
   * And finally the callbacks (only one in this example: react to state change)
   */
  abstract protected void transition(State old, State next);
}
```

The benefit of this approach is that state changes can be acted upon in one central place, which makes it impossible to forget inserting code for reacting to state transitions when adding to the FSM's machinery.

5.15.3 Message Buncher Example

The base class shown above is designed to support a similar example as for the Scala FSM documentation: an actor which receives and queues messages, to be delivered in batches to a configurable target actor. The messages

involved are:

```
public static final class SetTarget {
  final ActorRef ref;

  public SetTarget(ActorRef ref) {
    this.ref = ref;
  }
}

public static final class Queue {
  final Object o;

  public Queue(Object o) {
    this.o = o;
  }
}

public static final Object flush = new Object();

public static final class Batch {
  final List<Object> objects;

  public Batch(List<Object> objects) {
    this.objects = objects;
  }
}
```

This actor has only the two states `IDLE` and `ACTIVE`, making their handling quite straight-forward in the concrete actor derived from the base class:

```
import akka.event.LoggingAdapter;
import akka.event.Logging;
import akka.actor.UntypedActor;
```

```
static public class MyFSM extends MyFSMBase {

  private final LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  @Override
  public void onReceive(Object o) {

    if (getState() == State.IDLE) {

      if (o instanceof SetTarget)
        init(((SetTarget) o).ref);

      else
        whenUnhandled(o);

    } else if (getState() == State.ACTIVE) {

      if (o == flush)
        setState(State.IDLE);

      else
        whenUnhandled(o);
    }
  }

  @Override
  public void transition(State old, State next) {
    if (old == State.ACTIVE) {
      getTarget().tell(new Batch(drainQueue()));
```

```
    }
  }

  private void whenUnhandled(Object o) {
    if (o instanceof Queue && isInitialized()) {
      enqueue(((Queue) o).o);
      setState(State.ACTIVE);

    } else {
      log.warning("received unknown message {} in state {}", o, getState());
    }
  }
}
```

The trick here is to factor out common functionality like `whenUnhandled` and `transition` in order to obtain a few well-defined points for reacting to change or insert logging.

5.15.4 State-Centric vs. Event-Centric

In the example above, the subjective complexity of state and events was roughly equal, making it a matter of taste whether to choose primary dispatch on either; in the example a state-based dispatch was chosen. Depending on how evenly the matrix of possible states and events is populated, it may be more practical to handle different events first and distinguish the states in the second tier. An example would be a state machine which has a multitude of internal states but handles only very few distinct events.

5.16 Akka Extensions (Java)

If you want to add features to Akka, there is a very elegant, but powerful mechanism for doing so. It's called Akka Extensions and is comprised of 2 basic components: an `Extension` and an `ExtensionId`.

Extensions will only be loaded once per `ActorSystem`, which will be managed by Akka. You can choose to have your Extension loaded on-demand or at `ActorSystem` creation time through the Akka configuration. Details on how to make that happens are below, in the "Loading from Configuration" section.

> **Warning:** Since an extension is a way to hook into Akka itself, the implementor of the extension needs to ensure the thread safety of his/her extension.

5.16.1 Building an Extension

So let's create a sample extension that just lets us count the number of times something has happened.

First, we define what our `Extension` should do:

```
import akka.actor.*;
import java.util.concurrent.atomic.AtomicLong;

  public static class CountExtensionImpl implements Extension {
    //Since this Extension is a shared instance
    // per ActorSystem we need to be threadsafe
    private final AtomicLong counter = new AtomicLong(0);

    //This is the operation this Extension provides
    public long increment() {
      return counter.incrementAndGet();
    }
  }
```

Then we need to create an `ExtensionId` for our extension so we can grab ahold of it.

```
import akka.actor.*;
import java.util.concurrent.atomic.AtomicLong;

  public static class CountExtension extends AbstractExtensionId<CountExtensionImpl> implements Ex
    //This will be the identifier of our CountExtension
    public final static CountExtension CountExtensionProvider = new CountExtension();

    //The lookup method is required by ExtensionIdProvider,
    // so we return ourselves here, this allows us
    // to configure our extension to be loaded when
    // the ActorSystem starts up
    public CountExtension lookup() {
      return CountExtension.CountExtensionProvider; //The public static final
    }

    //This method will be called by Akka
    // to instantiate our Extension
    public CountExtensionImpl createExtension(ExtendedActorSystem system) {
      return new CountExtensionImpl();
    }
  }
```

Wicked! Now all we need to do is to actually use it:

```
// typically you would use static import of CountExtension.CountExtensionProvider field
CountExtension.CountExtensionProvider.get(system).increment();
```

Or from inside of an Akka Actor:

```
public static class MyActor extends UntypedActor {
  public void onReceive(Object msg) {
    // typically you would use static import of CountExtension.CountExtensionProvider field
    CountExtension.CountExtensionProvider.get(getContext().system()).increment();
  }
}
```

That's all there is to it!

5.16.2 Loading from Configuration

To be able to load extensions from your Akka configuration you must add FQCNs of implementations of either `ExtensionId` or `ExtensionIdProvider` in the "akka.extensions" section of the config you provide to your `ActorSystem`.

```
akka {
  extensions = ["akka.docs.extension.ExtensionDocTestBase.CountExtension"]
}
```

5.16.3 Applicability

The sky is the limit! By the way, did you know that Akka's `Typed Actors`, `Serialization` and other features are implemented as Akka Extensions?

Application specific settings

The *Configuration* can be used for application specific settings. A good practice is to place those settings in an Extension.

Sample configuration:

```
myapp {
  db {
    uri = "mongodb://example1.com:27017,example2.com:27017"
  }
  circuit-breaker {
    timeout = 30 seconds
  }
}
```

The `Extension`:

```
import akka.actor.Extension;
import akka.actor.AbstractExtensionId;
import akka.actor.ExtensionIdProvider;
import akka.actor.ActorSystem;
import akka.actor.ExtendedActorSystem;
import akka.util.Duration;
import com.typesafe.config.Config;
import java.util.concurrent.TimeUnit;

  public static class SettingsImpl implements Extension {

    public final String DB_URI;
    public final Duration CIRCUIT_BREAKER_TIMEOUT;

    public SettingsImpl(Config config) {
      DB_URI = config.getString(config.getString("myapp.db.uri"));
      CIRCUIT_BREAKER_TIMEOUT = Duration.create(config.getMilliseconds("myapp.circuit-breaker.time
          TimeUnit.MILLISECONDS);
    }

  }

  public static class Settings extends AbstractExtensionId<SettingsImpl> implements ExtensionIdPro
    public final static Settings SettingsProvider = new Settings();

    public Settings lookup() {
      return Settings.SettingsProvider;
    }

    public SettingsImpl createExtension(ExtendedActorSystem system) {
      return new SettingsImpl(system.settings().config());
    }
  }
```

Use it:

```
public static class MyActor extends UntypedActor {
  // typically you would use static import of CountExtension.CountExtensionProvider field
  final SettingsImpl settings = Settings.SettingsProvider.get(getContext().system());
  Connection connection = connect(settings.DB_URI, settings.CIRCUIT_BREAKER_TIMEOUT);
```

5.17 ZeroMQ (Java)

Akka provides a ZeroMQ module which abstracts a ZeroMQ connection and therefore allows interaction between Akka actors to take place over ZeroMQ connections. The messages can be of a proprietary format or they can be defined using Protobuf. The socket actor is fault-tolerant by default and when you use the newSocket method to create new sockets it will properly reinitialize the socket.

ZeroMQ is very opinionated when it comes to multi-threading so configuration option *akka.zeromq.socket-dispatcher* always needs to be configured to a PinnedDispatcher, because the actual ZeroMQ socket can only

be accessed by the thread that created it.

The ZeroMQ module for Akka is written against an API introduced in JZMQ, which uses JNI to interact with the native ZeroMQ library. Instead of using JZMQ, the module uses ZeroMQ binding for Scala that uses the native ZeroMQ library through JNA. In other words, the only native library that this module requires is the native ZeroMQ library. The benefit of the scala library is that you don't need to compile and manage native dependencies at the cost of some runtime performance. The scala-bindings are compatible with the JNI bindings so they are a drop-in replacement, in case you really need to get that extra bit of performance out.

5.17.1 Connection

ZeroMQ supports multiple connectivity patterns, each aimed to meet a different set of requirements. Currently, this module supports publisher-subscriber connections and connections based on dealers and routers. For connecting or accepting connections, a socket must be created. Sockets are always created using the `akka.zeromq.ZeroMQExtension`, for example:

```
import akka.zeromq.Bind;
import akka.zeromq.ZeroMQExtension;

    ActorRef pubSocket = ZeroMQExtension.get(system).newPubSocket(new Bind("tcp://127.0.0.1:1233"
```

Above examples will create a ZeroMQ Publisher socket that is Bound to the port 1233 on localhost.

Similarly you can create a subscription socket, with a listener, that subscribes to all messages from the publisher using:

```
import akka.zeromq.Connect;
import akka.zeromq.Listener;
import akka.zeromq.Subscribe;

    ActorRef listener = system.actorOf(new Props(ListenerActor.class));
    ActorRef subSocket = ZeroMQExtension.get(system).newSubSocket(new Connect("tcp://127.0.0.1:12
        new Listener(listener), Subscribe.all());
```

```
public static class ListenerActor extends UntypedActor {
  public void onReceive(Object message) throws Exception {
    //...
  }
}
```

The following sub-sections describe the supported connection patterns and how they can be used in an Akka environment. However, for a comprehensive discussion of connection patterns, please refer to ZeroMQ – The Guide.

Publisher-Subscriber Connection

In a publisher-subscriber (pub-sub) connection, the publisher accepts one or more subscribers. Each subscriber shall subscribe to one or more topics, whereas the publisher publishes messages to a set of topics. Also, a subscriber can subscribe to all available topics. In an Akka environment, pub-sub connections shall be used when an actor sends messages to one or more actors that do not interact with the actor that sent the message.

When you're using zeromq pub/sub you should be aware that it needs multicast - check your cloud - to work properly and that the filtering of events for topics happens client side, so all events are always broadcasted to every subscriber.

An actor is subscribed to a topic as follows:

```
ActorRef subTopicSocket = ZeroMQExtension.get(system).newSubSocket(new Connect("tcp://127.0.0.1:1
    new Listener(listener), new Subscribe("foo.bar"));
```

It is a prefix match so it is subscribed to all topics starting with `foo.bar`. Note that if the given string is empty or `Subscribe.all()` is used, the actor is subscribed to all topics.

To unsubscribe from a topic you do the following:

```
import akka.zeromq.Unsubscribe;

    subTopicSocket.tell(new Unsubscribe("foo.bar"));
```

To publish messages to a topic you must use two Frames with the topic in the first frame.

```
import akka.zeromq.Frame;
import akka.zeromq.ZMQMessage;

    pubSocket.tell(new ZMQMessage(new Frame("foo.bar"), new Frame(payload)));
```

Pub-Sub in Action

The following example illustrates one publisher with two subscribers.

The publisher monitors current heap usage and system load and periodically publishes `Heap` events on the `"health.heap"` topic and `Load` events on the `"health.load"` topic.

```
import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.actor.Props;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.util.Duration;
import akka.serialization.SerializationExtension;
import akka.serialization.Serialization;
import java.io.Serializable;
import java.lang.management.ManagementFactory;

  public static final Object TICK = "TICK";

  public static class Heap implements Serializable {
    public final long timestamp;
    public final long used;
    public final long max;

    public Heap(long timestamp, long used, long max) {
      this.timestamp = timestamp;
      this.used = used;
      this.max = max;
    }
  }

  public static class Load implements Serializable {
    public final long timestamp;
    public final double loadAverage;

    public Load(long timestamp, double loadAverage) {
      this.timestamp = timestamp;
      this.loadAverage = loadAverage;
    }
  }

  public static class HealthProbe extends UntypedActor {

    ActorRef pubSocket = ZeroMQExtension.get(getContext().system()).newPubSocket(new Bind("tcp://
    MemoryMXBean memory = ManagementFactory.getMemoryMXBean();
    OperatingSystemMXBean os = ManagementFactory.getOperatingSystemMXBean();
    Serialization ser = SerializationExtension.get(getContext().system());

    @Override
```

```
    public void preStart() {
      getContext().system().scheduler()
          .schedule(Duration.parse("1 second"), Duration.parse("1 second"), getSelf(), TICK);
    }

    @Override
    public void postRestart(Throwable reason) {
      // don't call preStart, only schedule once
    }

    @Override
    public void onReceive(Object message) {
      if (message.equals(TICK)) {
        MemoryUsage currentHeap = memory.getHeapMemoryUsage();
        long timestamp = System.currentTimeMillis();

        // use akka SerializationExtension to convert to bytes
        byte[] heapPayload = ser.serializerFor(Heap.class).toBinary(
            new Heap(timestamp, currentHeap.getUsed(), currentHeap.getMax()));
        // the first frame is the topic, second is the message
        pubSocket.tell(new ZMQMessage(new Frame("health.heap"), new Frame(heapPayload)));

        // use akka SerializationExtension to convert to bytes
        byte[] loadPayload = ser.serializerFor(Load.class).toBinary(new Load(timestamp, os.getSyst
        // the first frame is the topic, second is the message
        pubSocket.tell(new ZMQMessage(new Frame("health.load"), new Frame(loadPayload)));
      } else {
        unhandled(message);
      }
    }

  }
```

```
system.actorOf(new Props(HealthProbe.class), "health");
```

Let's add one subscriber that logs the information. It subscribes to all topics starting with `"health"`, i.e. both `Heap` and `Load` events.

```
public static class Logger extends UntypedActor {

  ActorRef subSocket = ZeroMQExtension.get(getContext().system()).newSubSocket(new Connect("tcp:/
      new Listener(getSelf()), new Subscribe("health"));
  Serialization ser = SerializationExtension.get(getContext().system());
  SimpleDateFormat timestampFormat = new SimpleDateFormat("HH:mm:ss.SSS");
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  @Override
  public void onReceive(Object message) {
    if (message instanceof ZMQMessage) {
      ZMQMessage m = (ZMQMessage) message;
      // the first frame is the topic, second is the message
      if (m.firstFrameAsString().equals("health.heap")) {
        Heap heap = (Heap) ser.serializerFor(Heap.class).fromBinary(m.payload(1));
        log.info("Used heap {} bytes, at {}", heap.used, timestampFormat.format(new Date(heap.time
      } else if (m.firstFrameAsString().equals("health.load")) {
        Load load = (Load) ser.serializerFor(Load.class).fromBinary(m.payload(1));
        log.info("Load average {}, at {}", load.loadAverage, timestampFormat.format(new Date(load
      }
    } else {
      unhandled(message);
    }
  }
```

```
}
```

```
system.actorOf(new Props(Logger.class), "logger");
```

Another subscriber keep track of used heap and warns if too much heap is used. It only subscribes to `Heap` events.

```
public static class HeapAlerter extends UntypedActor {

  ActorRef subSocket = ZeroMQExtension.get(getContext().system()).newSubSocket(new Connect("tcp:/
      new Listener(getSelf()), new Subscribe("health.heap"));
  Serialization ser = SerializationExtension.get(getContext().system());
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);
  int count = 0;

  @Override
  public void onReceive(Object message) {
    if (message instanceof ZMQMessage) {
      ZMQMessage m = (ZMQMessage) message;
      // the first frame is the topic, second is the message
      if (m.firstFrameAsString().equals("health.heap")) {
        Heap heap = (Heap) ser.serializerFor(Heap.class).fromBinary(m.payload(1));
        if (((double) heap.used / heap.max) > 0.9) {
          count += 1;
        } else {
          count = 0;
        }
        if (count > 10) {
          log.warning("Need more memory, using {} %", (100.0 * heap.used / heap.max));
        }
      }
    } else {
      unhandled(message);
    }
  }

}
```

```
system.actorOf(new Props(HeapAlerter.class), "alerter");
```

Router-Dealer Connection

While Pub/Sub is nice the real advantage of zeromq is that it is a "lego-box" for reliable messaging. And because there are so many integrations the multi-language support is fantastic. When you're using ZeroMQ to integrate many systems you'll probably need to build your own ZeroMQ devices. This is where the router and dealer socket types come in handy. With those socket types you can build your own reliable pub sub broker that uses TCP/IP and does publisher side filtering of events.

To create a Router socket that has a high watermark configured, you would do:

```
ActorRef highWatermarkSocket = ZeroMQExtension.get(system).newRouterSocket(
    new SocketOption[] { new Listener(listener), new Bind("tcp://127.0.0.1:1233"), new HighWaterma
```

The akka-zeromq module accepts most if not all the available configuration options for a zeromq socket.

Push-Pull Connection

Akka ZeroMQ module supports `Push-Pull` connections.

You can create a `Push` connection through the:

```
ActorRef newPushSocket(SocketOption[] socketParameters);
```

You can create a `Pull` connection through the:

```
ActorRef newPullSocket(SocketOption[] socketParameters);
```

More documentation and examples will follow soon.

Rep-Req Connection

Akka ZeroMQ module supports `Rep-Req` connections.

You can create a `Rep` connection through the:

```
ActorRef newRepSocket(SocketOption[] socketParameters);
```

You can create a `Req` connection through the:

```
ActorRef newReqSocket(SocketOption[] socketParameters);
```

More documentation and examples will follow soon.

5.18 Microkernel (Java)

The Akka Microkernel is included in the Akka download found at downloads.

To run an application with the microkernel you need to create a Bootable class that handles the startup and shutdown the application. An example is included below.

Put your application jar in the `deploy` directory to have it automatically loaded.

To start the kernel use the scripts in the `bin` directory, passing the boot classes for your application.

There is a simple example of an application setup for running with the microkernel included in the akka download. This can be run with the following command (on a unix-based system):

```
bin/akka sample.kernel.hello.HelloKernel
```

Use `Ctrl-C` to interrupt and exit the microkernel.

On a Windows machine you can also use the bin/akka.bat script.

The code for the Hello Kernel example (see the `HelloKernel` class for an example of creating a Bootable):

```
/**
 *  Copyright (C) 2009-2012 Typesafe Inc. <http://www.typesafe.com>
 */
package sample.kernel.hello.java;

import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.actor.ActorSystem;
import akka.actor.Props;
import akka.kernel.Bootable;

public class HelloKernel implements Bootable {
  final ActorSystem system = ActorSystem.create("hellokernel");

  public static class HelloActor extends UntypedActor {
    final ActorRef worldActor =
      getContext().actorOf(new Props(WorldActor.class));

  public void onReceive(Object message) {
```

```
    if (message == "start")
      worldActor.tell("Hello");
    else if (message instanceof String)
      System.out.println("Received message '%s'".format((String)message));
    else unhandled(message);
  }
}

public static class WorldActor extends UntypedActor {
  public void onReceive(Object message) {
    if (message instanceof String)
      getSender().tell(((String)message).toUpperCase() + " world!");
    else unhandled(message);
  }
}

  public void startup() {
    system.actorOf(new Props(HelloActor.class)).tell("start");
  }

  public void shutdown() {
    system.shutdown();
  }
}
```

5.18.1 Distribution of microkernel application

To make a distribution package of the microkernel and your application the `akka-sbt-plugin` provides `AkkaKernelPlugin`. It creates the directory structure, with jar files, configuration files and start scripts.

To use the sbt plugin you define it in your `project/plugins.sbt`:

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

addSbtPlugin("com.typesafe.akka" % "akka-sbt-plugin" % "2.0.4")
```

Then you add it to the settings of your `project/Build.scala`. It is also important that you add the `akka-kernel` dependency. This is an example of a complete sbt build file:

```
import sbt._
import Keys._
import akka.sbt.AkkaKernelPlugin
import akka.sbt.AkkaKernelPlugin.{ Dist, outputDirectory, distJvmOptions}

object HelloKernelBuild extends Build {
  val Organization = "akka.sample"
  val Version      = "2.0.4"
  val ScalaVersion = "2.9.1"

  lazy val HelloKernel = Project(
    id = "hello-kernel",
    base = file("."),
    settings = defaultSettings ++ AkkaKernelPlugin.distSettings ++ Seq(
      libraryDependencies ++= Dependencies.helloKernel,
      distJvmOptions in Dist := "-Xms256M -Xmx1024M",
      outputDirectory in Dist := file("target/hello-dist")
    )
  )

  lazy val buildSettings = Defaults.defaultSettings ++ Seq(
    organization := Organization,
    version      := Version,
```

```
    scalaVersion := ScalaVersion,
    crossPaths   := false,
    organizationName := "Typesafe Inc.",
    organizationHomepage := Some(url("http://www.typesafe.com"))
  )

  lazy val defaultSettings = buildSettings ++ Seq(
    resolvers += "Typesafe Repo" at "http://repo.typesafe.com/typesafe/releases/",

    // compile options
    scalacOptions ++= Seq("-encoding", "UTF-8", "-deprecation", "-unchecked"),
    javacOptions  ++= Seq("-Xlint:unchecked", "-Xlint:deprecation")

  )
}

object Dependencies {
  import Dependency._

  val helloKernel = Seq(
    akkaKernel, akkaSlf4j, logback
  )
}

object Dependency {
  // Versions
  object V {
    val Akka      = "2.0.4"
  }

  val akkaKernel        = "com.typesafe.akka" % "akka-kernel"         % V.Akka
  val akkaSlf4j         = "com.typesafe.akka" % "akka-slf4j"          % V.Akka
  val logback           = "ch.qos.logback"    % "logback-classic"     % "1.0.0"
}
```

Run the plugin with sbt:

```
> dist
> dist:clean
```

There are several settings that can be defined:

- `outputDirectory` - destination directory of the package, default `target/dist`
- `distJvmOptions` - JVM parameters to be used in the start script
- `configSourceDirs` - Configuration files are copied from these directories, default `src/config`, `src/main/config`, `src/main/resources`
- `distMainClass` - Kernel main class to use in start script
- `libFilter` - Filter of dependency jar files
- `additionalLibs` - Additional dependency jar files

CHAPTER

SIX

CLUSTER

6.1 Cluster Specification

Note: *This document describes the new clustering coming in Akka 2.1 (not 2.0)*

6.1.1 Intro

Akka Cluster provides a fault-tolerant, elastic, decentralized peer-to-peer cluster with no single point of failure (SPOF) or single point of bottleneck (SPOB). It implements a Dynamo-style system using gossip protocols, automatic failure detection, automatic partitioning, handoff, and cluster rebalancing. But with some differences due to the fact that it is not just managing passive data, but actors - active, sometimes stateful, components that also have requirements on message ordering, the number of active instances in the cluster, etc.

6.1.2 Terms

These terms are used throughout the documentation.

node A logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a *hostname:port* tuple.

cluster A set of nodes. Contains distributed Akka applications.

partition An actor or subtree of actors in the Akka application that is distributed within the cluster.

partition point The actor at the head of a partition. The point around which a partition is formed.

partition path Also referred to as the actor address. Has the format *actor1/actor2/actor3*

instance count The number of instances of a partition in the cluster. Also referred to as the `N-value` of the partition.

instance node A node that an actor instance is assigned to.

partition table A mapping from partition path to a set of instance nodes (where the nodes are referred to by the ordinal position given the nodes in sorted order).

6.1.3 Membership

A cluster is made up of a set of member nodes. The identifier for each node is a *hostname:port* pair. An Akka application is distributed over a cluster with each node hosting some part of the application. Cluster membership and partitioning of the application are decoupled. A node could be a member of a cluster without hosting any actors.

Gossip

The cluster membership used in Akka is based on Amazon's Dynamo system and particularly the approach taken in Basho's' Riak distributed database. Cluster membership is communicated using a Gossip Protocol, where the current state of the cluster is gossiped randomly through the cluster. Joining a cluster is initiated by specifying a set of `seed` nodes with which to begin gossiping.

Vector Clocks

Vector clocks are an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations.

We use vector clocks to to reconcile and merge differences in cluster state during gossiping. A vector clock is a set of (node, counter) pairs. Each update to the cluster state has an accompanying update to the vector clock.

One problem with vector clocks is that their history can over time be very long, which will both make comparisons take longer time as well as take up unnecessary memory. To solve that problem we do pruning of the vector clocks according to the pruning algorithm in Riak.

Gossip convergence

Information about the cluster converges at certain points of time. This is when all nodes have seen the same cluster state. Convergence is recognised by passing a map from node to current state version during gossip. This information is referred to as the gossip overview. When all versions in the overview are equal there is convergence. Gossip convergence cannot occur while any nodes are unreachable, either the nodes become reachable again, or the nodes need to be moved into the `down` or `removed` states (see section on Member states below).

Failure Detector

The failure detector is responsible for trying to detect if a node is unreachable from the rest of the cluster. For this we are using an implementation of The Phi Accrual Failure Detector by Hayashibara et al.

An accrual failure detector decouple monitoring and interpretation. That makes them applicable to a wider area of scenarios and more adequate to build generic failure detection services. The idea is that it is keeping a history of failure statistics, calculated from heartbeats received from the gossip protocol, and is trying to do educated guesses by taking multiple factors, and how they accumulate over time, into account in order to come up with a better guess if a specific node is up or down. Rather than just answering "yes" or "no" to the question "is the node down?" it returns a `phi` value representing the likelihood that the node is down.

The `threshold` that is the basis for the calculation is configurable by the user. A low `threshold` is prone to generate many wrong suspicions but ensures a quick detection in the event of a real crash. Conversely, a high `threshold` generates fewer mistakes but needs more time to detect actual crashes. The default `threshold` is 8 and is appropriate for most situations. However in cloud environments, such as Amazon EC2, the value could be increased to 12 in order to account for network issues that sometimes occur on such platforms.

Leader

After gossip convergence a leader for the cluster can be determined. There is no leader election process, the leader can always be recognised deterministically by any node whenever there is gossip convergence. The leader is simply the first node in sorted order that is able to take the leadership role, where the only allowed member states for a leader are `up` or `leaving` (see below for more information about member states).

The role of the leader is to shift members in and out of the cluster, changing `joining` members to the `up` state or `exiting` members to the `removed` state, and to schedule rebalancing across the cluster. Currently leader actions are only triggered by receiving a new cluster state with gossip convergence but it may also be possible for the user to explicitly rebalance the cluster by specifying migrations, or to rebalance the cluster automatically based on metrics from member nodes. Metrics may be spread using the gossip protocol or possibly more efficiently using a

random chord method, where the leader contacts several random nodes around the cluster ring and each contacted node gathers information from their immediate neighbours, giving a random sampling of load information.

The leader also has the power, if configured so, to "auto-down" a node that according to the Failure Detector is considered unreachable. This means setting the unreachable node status to `down` automatically.

Gossip protocol

A variation of *push-pull gossip* is used to reduce the amount of gossip information sent around the cluster. In push-pull gossip a digest is sent representing current versions but not actual values; the recipient of the gossip can then send back any values for which it has newer versions and also request values for which it has outdated versions. Akka uses a single shared state with a vector clock for versioning, so the variant of push-pull gossip used in Akka makes use of the gossip overview (containing the current state versions for all nodes) to only push the actual state as needed. This also allows any node to easily determine which other nodes have newer or older information, not just the nodes involved in a gossip exchange.

Periodically, the default is every 1 second, each node chooses another random node to initiate a round of gossip with. The choice of node is random but can also include extra gossiping for unreachable nodes, seed nodes, and nodes with either newer or older state versions.

The gossip overview contains the current state version for all nodes and also a list of unreachable nodes. Whenever a node receives a gossip overview it updates the Failure Detector with the liveness information.

The nodes defined as `seed` nodes are just regular member nodes whose only "special role" is to function as contact points in the cluster and to help breaking logical partitions as seen in the gossip algorithm defined below.

During each round of gossip exchange the following process is used:

1. Gossip to random live node (if any)
2. Gossip to random unreachable node with certain probability depending on the number of unreachable and live nodes
3. If the node gossiped to at (1) was not a `seed` node, or the number of live nodes is less than number of seeds, gossip to random `seed` node with certain probability depending on number of unreachable, seed, and live nodes.
4. Gossip to random node with newer or older state information, based on the current gossip overview, with some probability (?)

The gossiper only sends the gossip overview to the chosen node. The recipient of the gossip can use the gossip overview to determine whether:

1. it has a newer version of the gossip state, in which case it sends that back to the gossiper, or
2. it has an outdated version of the state, in which case the recipient requests the current state from the gossiper

If the recipient and the gossip have the same version then the gossip state is not sent or requested.

The main structures used in gossiping are the gossip overview and the gossip state:

```
 GossipOverview {
   versions: Map[Node, VectorClock],
   unreachable: Set[Node]
 }

GossipState {
   version: VectorClock,
   members: SortedSet[Member],
   partitions: Tree[PartitionPath, Node],
   pending: Set[PartitionChange],
   meta: Option[Map[String, Array[Byte]]]
 }
```

Some of the other structures used are:

```
Node = InetSocketAddress

Member {
  node: Node,
  state: MemberState
}

MemberState = Joining | Up | Leaving | Exiting | Down | Removed

PartitionChange {
  from: Node,
  to: Node,
  path: PartitionPath,
  status: PartitionChangeStatus
}

PartitionChangeStatus = Awaiting | Complete
```

Membership lifecycle

A node begins in the `joining` state. Once all nodes have seen that the new node is joining (through gossip convergence) the leader will set the member state to `up` and can start assigning partitions to the new node.

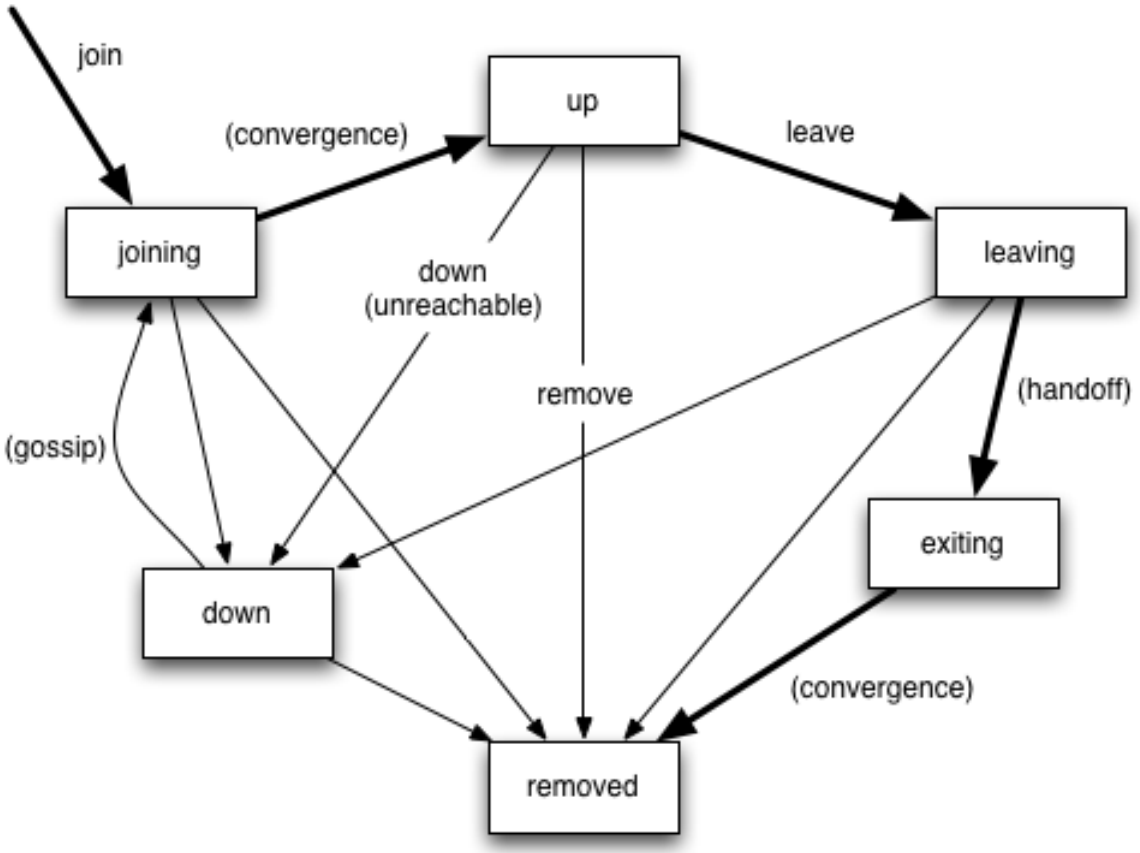
If a node is leaving the cluster in a safe, expected manner then it switches to the `leaving` state. The leader will reassign partitions across the cluster (it is possible for a leaving node to itself be the leader). When all partition handoff has completed then the node will change to the `exiting` state. Once all nodes have seen the exiting state (convergence) the leader will remove the node from the cluster, marking it as `removed`.

A node can also be removed forcefully by moving it directly to the `removed` state using the `remove` action. The cluster will rebalance based on the new cluster membership.

If a node is unreachable then gossip convergence is not possible and therefore any leader actions are also not possible (for instance, allowing a node to become a part of the cluster, or changing actor distribution). To be able to move forward the state of the unreachable nodes must be changed. If the unreachable node is experiencing only transient difficulties then it can be explicitly marked as `down` using the `down` user action. When this node comes back up and begins gossiping it will automatically go through the joining process again. If the unreachable node will be permanently down then it can be removed from the cluster directly with the `remove` user action. The cluster can also *auto-down* a node using the accrual failure detector.

This means that nodes can join and leave the cluster at any point in time, e.g. provide cluster elasticity.

State diagram for the member states



Member states

- **joining** transient state when joining a cluster
- **up** normal operating state
- **leaving / exiting** states during graceful removal
- **removed** tombstone state (no longer a member)
- **down** marked as down/offline/unreachable

User actions

- **join** join a single node to a cluster - can be explicit or automatic on startup if a list of seed nodes have been specified in the configuration
- **leave** tell a node to leave the cluster gracefully
- **down** mark a node as temporarily down
- **remove** remove a node from the cluster immediately

Leader actions

The leader has the following duties:

- shifting members in and out of the cluster

  - joining -> up
  - exiting -> removed
- partition distribution
  - scheduling handoffs (pending changes)
  - setting the partition table (partition path -> base node)
  - Automatic rebalancing based on runtime metrics in the system (such as CPU, RAM, Garbage Collection, mailbox depth etc.)

6.1.4 Partitioning

Each partition (an actor or actor subtree) in the actor system is assigned to a set of nodes in the cluster. The actor at the head of the partition is referred to as the partition point. The mapping from partition path (actor address of the format "a/b/c") to instance nodes is stored in the partition table and is maintained as part of the cluster state through the gossip protocol. The partition table is only updated by the leader node. Currently the only possible partition points are *routed* actors.

Routed actors can have an instance count greater than one. The instance count is also referred to as the `N-value`. If the `N-value` is greater than one then a set of instance nodes will be given in the partition table.

Note that in the first implementation there may be a restriction such that only top-level partitions are possible (the highest possible partition points are used and sub-partitioning is not allowed). Still to be explored in more detail.

The cluster leader determines the current instance count for a partition based on two axes: fault-tolerance and scaling.

Fault-tolerance determines a minimum number of instances for a routed actor (allowing N-1 nodes to crash while still maintaining at least one running actor instance). The user can specify a function from current number of nodes to the number of acceptable node failures: n: Int => f: Int where f < n.

Scaling reflects the number of instances needed to maintain good throughput and is influenced by metrics from the system, particularly a history of mailbox size, CPU load, and GC percentages. It may also be possible to accept scaling hints from the user that indicate expected load.

The balancing of partitions can be determined in a very simple way in the first implementation, where the overlap of partitions is minimized. Partitions are spread over the cluster ring in a circular fashion, with each instance node in the first available space. For example, given a cluster with ten nodes and three partitions, A, B, and C, having N-values of 4, 3, and 5; partition A would have instances on nodes 1-4; partition B would have instances on nodes 5-7; partition C would have instances on nodes 8-10 and 1-2. The only overlap is on nodes 1 and 2.

The distribution of partitions is not limited, however, to having instances on adjacent nodes in the sorted ring order. Each instance can be assigned to any node and the more advanced load balancing algorithms will make use of this. The partition table contains a mapping from path to instance nodes. The partitioning for the above example would be:

```
A -> { 1, 2, 3, 4 }
B -> { 5, 6, 7 }
C -> { 8, 9, 10, 1, 2 }
```

If 5 new nodes join the cluster and in sorted order these nodes appear after the current nodes 2, 4, 5, 7, and 8, then the partition table could be updated to the following, with all instances on the same physical nodes as before:

```
A -> { 1, 2, 4, 5 }
B -> { 7, 9, 10 }
C -> { 12, 14, 15, 1, 2 }
```

When rebalancing is required the leader will schedule handoffs, gossiping a set of pending changes, and when each change is complete the leader will update the partition table.

Handoff

Handoff for an actor-based system is different than for a data-based system. The most important point is that message ordering (from a given node to a given actor instance) may need to be maintained. If an actor is a singleton actor (only one instance possible throughout the cluster) then the cluster may also need to assure that there is only one such actor active at any one time. Both of these situations can be handled by forwarding and buffering messages during transitions.

A *graceful handoff* (one where the previous host node is up and running during the handoff), given a previous host node `N1`, a new host node `N2`, and an actor partition `A` to be migrated from `N1` to `N2`, has this general structure:

1. the leader sets a pending change for `N1` to handoff `A` to `N2`
2. `N1` notices the pending change and sends an initialization message to `N2`
3. in response `N2` creates `A` and sends back a ready message
4. after receiving the ready message `N1` marks the change as complete and shuts down `A`
5. the leader sees the migration is complete and updates the partition table
6. all nodes eventually see the new partitioning and use `N2`

Transitions

There are transition times in the handoff process where different approaches can be used to give different guarantees.

Migration transition The first transition starts when `N1` initiates the moving of `A` and ends when `N1` receives the ready message, and is referred to as the *migration transition*.

The first question is; during the migration transition, should:

- `N1` continue to process messages for `A`?
- Or is it important that no messages for `A` are processed on `N1` once migration begins?

If it is okay for the previous host node `N1` to process messages during migration then there is nothing that needs to be done at this point.

If no messages are to be processed on the previous host node during migration then there are two possibilities: the messages are forwarded to the new host and buffered until the actor is ready, or the messages are simply dropped by terminating the actor and allowing the normal dead letter process to be used.

Update transition The second transition begins when the migration is marked as complete and ends when all nodes have the updated partition table (when all nodes will use `N2` as the host for `A`, i.e. we have convergence) and is referred to as the *update transition*.

Once the update transition begins `N1` can forward any messages it receives for `A` to the new host `N2`. The question is whether or not message ordering needs to be preserved. If messages sent to the previous host node `N1` are being forwarded, then it is possible that a message sent to `N1` could be forwarded after a direct message to the new host `N2`, breaking message ordering from a client to actor `A`.

In this situation `N2` can keep a buffer for messages per sending node. Each buffer is flushed and removed when an acknowledgement (`ack`) message has been received. When each node in the cluster sees the partition update it first sends an `ack` message to the previous host node `N1` before beginning to use `N2` as the new host for `A`. Any messages sent from the client node directly to `N2` will be buffered. `N1` can count down the number of acks to determine when no more forwarding is needed. The `ack` message from any node will always follow any other messages sent to `N1`. When `N1` receives the `ack` message it also forwards it to `N2` and again this `ack` message will follow any other messages already forwarded for `A`. When `N2` receives an `ack` message, the buffer for the sending node can be flushed and removed. Any subsequent messages from this sending node can be queued normally. Once all nodes in the cluster have acknowledged the partition change and `N2` has cleared all buffers, the handoff is

complete and message ordering has been preserved. In practice the buffers should remain small as it is only those messages sent directly to `N2` before the acknowledgement has been forwarded that will be buffered.

Graceful handoff

A more complete process for graceful handoff would be:

1. the leader sets a pending change for `N1` to handoff `A` to `N2`
2. `N1` notices the pending change and sends an initialization message to `N2`. Options:
   (a) keep `A` on `N1` active and continuing processing messages as normal
   (b) `N1` forwards all messages for `A` to `N2`
   (c) `N1` drops all messages for `A` (terminate `A` with messages becoming dead letters)
3. in response `N2` creates `A` and sends back a ready message. Options:
   (a) `N2` simply processes messages for `A` as normal
   (b) `N2` creates a buffer per sending node for `A`. Each buffer is opened (flushed and removed) when an acknowledgement for the sending node has been received (via `N1`)
4. after receiving the ready message `N1` marks the change as complete. Options:
   (a) `N1` forwards all messages for `A` to `N2` during the update transition
   (b) `N1` drops all messages for `A` (terminate `A` with messages becoming dead letters)
5. the leader sees the migration is complete and updates the partition table
6. all nodes eventually see the new partitioning and use `N2`
   (a) each node sends an acknowledgement message to `N1`
   (b) when `N1` receives the acknowledgement it can count down the pending acknowledgements and remove forwarding when complete
   (c) when `N2` receives the acknowledgement it can open the buffer for the sending node (if buffers are used)

The default approach is to take options 2a, 3a, and 4a - allowing `A` on `N1` to continue processing messages during migration and then forwarding any messages during the update transition. This assumes stateless actors that do not have a dependency on message ordering from any given source.

- If an actor has a distributed durable mailbox then nothing needs to be done, other than migrating the actor.
- If message ordering needs to be maintained during the update transition then option 3b can be used, creating buffers per sending node.
- If the actors are robust to message send failures then the dropping messages approach can be used (with no forwarding or buffering needed).
- If an actor is a singleton (only one instance possible throughout the cluster) and state is transferred during the migration initialization, then options 2b and 3b would be required.

6.1.5 Stateful Actor Replication

Support for stateful singleton actors will come in future releases of Akka, and is scheduled for Akka 2.2. Having a Dynamo base for the clustering already we should use the same infrastructure to provide stateful actor clustering and datastore as well. The stateful actor clustering should be layered on top of the distributed datastore. See the next section for a rough outline on how the distributed datastore could be implemented.

Implementing a Dynamo-style distributed database on top of Akka Cluster

The missing pieces to implement a full Dynamo-style eventually consistent data storage on top of the Akka Cluster as described in this document are:

- Configuration of `READ` and `WRITE` consistency levels according to the `N/R/W` numbers defined in the Dynamo paper.
  - R = read replica count
  - W = write replica count
  - N = replication factor
  - Q = QUORUM = N / 2 + 1
  - W + R > N = full consistency
- Define a versioned data message wrapper:

```
Versioned[T](hash: Long, version: VectorClock, data: T)
```

- Define a single system data broker actor on each node that uses a `Consistent Hashing Router` and that have instances on all other nodes in the node ring.
- For `WRITE`:
  1. Wrap data in a `Versioned Message`
  2. Send a `Versioned Message` with the data is sent to a number of nodes matching the `W-value`.
- For `READ`:
  1. Read in the `Versioned Message` with the data from as many replicas as you need for the consistency level required by the `R-value`.
  2. Do comparison on the versions (using Vector Clocks)
  3. If the versions differ then do Read Repair to update the inconsistent nodes.
  4. Return the latest versioned data.

CHAPTER

SEVEN

MODULES

7.1 Durable Mailboxes

7.1.1 Overview

Akka supports a set of durable mailboxes. A durable mailbox is a replacement for the standard actor mailbox that is durable. What this means in practice is that if there are pending messages in the actor's mailbox when the node of the actor resides on crashes, then when you restart the node, the actor will be able to continue processing as if nothing had happened; with all pending messages still in its mailbox.

None of these mailboxes implements transactions for current message. It's possible if the actor crashes after receiving a message, but before completing processing of it, that the message could be lost.

Warning: IMPORTANT
None of these mailboxes work with blocking message send, i.e. the message send operations that are relying on futures; `?` or `ask`. If the node has crashed and then restarted, the thread that was blocked waiting for the reply is gone and there is no way we can deliver the message.

The durable mailboxes currently supported are:

- `FileBasedMailbox` – backed by a journaling transaction log on the local file system
- **DEPRECATED** `RedisBasedMailbox` – backed by Redis
- **DEPRECATED** `ZooKeeperBasedMailbox` – backed by ZooKeeper
- **DEPRECATED** `BeanstalkBasedMailbox` – backed by Beanstalkd
- **DEPRECATED** `MongoBasedMailbox` – backed by MongoDB

Warning: IMPORTANT
In order to streamline the distribution, we have deprecated all durable mailboxes except for the FileBasedMailbox, which will serve as a blueprint for how to implement new ones and to serve as the default durable mailbox implementation.

We'll walk through each one of these in detail in the sections below.

You can easily implement your own mailbox. Look at the existing implementations for inspiration.

Let us know if you have a wish for a certain priority order.

General Usage

The durable mailboxes and their configuration options reside in the `akka.actor.mailbox` package.

You configure durable mailboxes through the dispatcher. The actor is oblivious to which type of mailbox it is using.

In the configuration of the dispatcher you specify the fully qualified class name of the mailbox:

```
my-dispatcher {
  mailbox-type = akka.actor.mailbox.FileBasedMailboxType
}
```

Here is an example of how to create an actor with a durable dispatcher, in Scala:

```
import akka.actor.Props

    val myActor = system.actorOf(Props[MyActor].withDispatcher("my-dispatcher"), name = "myactor")
```

Corresponding example in Java:

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

    ActorRef myActor = system.actorOf(
        new Props().withDispatcher("my-dispatcher").withCreator(new UntypedActorFactory() {
          public UntypedActor create() {
            return new MyUntypedActor();
          }
        }), "myactor");
```

The actor is oblivious to which type of mailbox it is using.

This gives you an excellent way of creating bulkheads in your application, where groups of actors sharing the same dispatcher also share the same backing storage. Read more about that in the *Dispatchers (Scala)* documentation.

7.1.2 File-based durable mailbox

This mailbox is backed by a journaling transaction log on the local file system. It is the simplest to use since it does not require an extra infrastructure piece to administer, but it is usually sufficient and just what you need.

You configure durable mailboxes through the dispatcher, as described in *General Usage* with the following mailbox type.

Config:

```
my-dispatcher {
  mailbox-type = akka.actor.mailbox.FileBasedMailboxType
}
```

You can also configure and tune the file-based durable mailbox. This is done in the `akka.actor.mailbox.file-based` section in the *Configuration*.

```
##############################################
# Akka File Mailboxes Reference Config File #
##############################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.
#
# For more information see <https://github.com/robey/kestrel/>

akka {
  actor {
    mailbox {
      file-based {
        # directory below which this queue resides
        directory-path = "./_mb"

        # attempting to add an item after the queue reaches this size (in items) will fail.
```

```
        max-items = 2147483647

        # attempting to add an item after the queue reaches this size (in bytes) will fail.
        max-size = 2147483647 bytes

        # attempting to add an item larger than this size (in bytes) will fail.
        max-item-size = 2147483647 bytes

        # maximum expiration time for this queue (seconds).
        max-age = 0s

        # maximum journal size before the journal should be rotated.
        max-journal-size = 16 MiB

        # maximum size of a queue before it drops into read-behind mode.
        max-memory-size = 128 MiB

        # maximum overflow (multiplier) of a journal file before we re-create it.
        max-journal-overflow = 10

        # absolute maximum size of a journal file until we rebuild it, no matter what.
        max-journal-size-absolute = 9223372036854775807 bytes

        # whether to drop older items (instead of newer) when the queue is full
        discard-old-when-full = on

        # whether to keep a journal file at all
        keep-journal = on

        # whether to sync the journal after each transaction
        sync-journal = off
      }
    }
  }
}
```

7.1.3 Redis-based durable mailbox

Warning: IMPORTANT
In order to streamline the distribution, we have deprecated all durable mailboxes except for the FileBasedMailbox, which will serve as a blueprint for how to implement new ones and to serve as the default durable mailbox implementation.

This mailbox is backed by a Redis queue. Redis Is a very fast NOSQL database that has a wide range of data structure abstractions, one of them is a queue which is what we are using in this implementation. This means that you have to start up a Redis server that can host these durable mailboxes. Read more in the Redis documentation on how to do that.

You configure durable mailboxes through the dispatcher, as described in *General Usage* with the following mailbox type.

Config:

```
my-dispatcher {
  mailbox-type = akka.actor.mailbox.RedisBasedMailboxType
}
```

You also need to configure the IP and port for the Redis server. This is done in the `akka.actor.mailbox.redis` section in the *Configuration*.

```
#############################################
# Akka Redis Mailboxes Reference Config File #
#############################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.
#
# for more information see <http://redis.io/>

akka {
  actor {
    mailbox {
      redis {
        # hostname of where the redis queue resides
        hostname = "127.0.0.1"

        # port at which the redis queue resides
        port = 6379
      }
    }
  }
}
```

7.1.4 ZooKeeper-based durable mailbox

Warning: IMPORTANT
In order to streamline the distribution, we have deprecated all durable mailboxes except for the FileBasedMailbox, which will serve as a blueprint for how to implement new ones and to serve as the default durable mailbox implementation.

This mailbox is backed by ZooKeeper. ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services This means that you have to start up a ZooKeeper server (for production a ZooKeeper server ensemble) that can host these durable mailboxes. Read more in the ZooKeeper documentation on how to do that.

You configure durable mailboxes through the dispatcher, as described in *General Usage* with the following mailbox type.

Config:

```
my-dispatcher {
  mailbox-type = akka.actor.mailbox.ZooKeeperBasedMailboxType
}
```

You also need to configure ZooKeeper server addresses, timeouts, etc. This is done in the `akka.actor.mailbox.zookeeper` section in the *Configuration*.

```
################################################
# Akka ZooKepper Mailboxes Reference Config File #
################################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.
#
# For more information see <http://wiki.apache.org/hadoop/ZooKeeper>

akka {
  actor {
    mailbox {
      zookeeper {
        # host and port to connect to ZooKeeper
```

```
        server-addresses = "127.0.0.1:2181"

        # timeout after which an unreachable client is considered dead and its session is closed
        session-timeout = 60s

        # maximum wait period while connecting to ZooKeeper service
        connection-timeout = 60s
      }
    }
  }
}
```

7.1.5 Beanstalk-based durable mailbox

Warning: IMPORTANT
In order to streamline the distribution, we have deprecated all durable mailboxes except for the FileBasedMailbox, which will serve as a blueprint for how to implement new ones and to serve as the default durable mailbox implementation.

This mailbox is backed by Beanstalkd. Beanstalk is a simple, fast work queue. This means that you have to start up a Beanstalk server that can host these durable mailboxes. Read more in the Beanstalk documentation on how to do that.

You configure durable mailboxes through the dispatcher, as described in *General Usage* with the following mailbox type.

Config:

```
my-dispatcher {
  mailbox-type = akka.actor.mailbox.BeanstalkBasedMailboxType
}
```

You also need to configure the IP, and port, and so on, for the Beanstalk server. This is done in the `akka.actor.mailbox.beanstalk` section in the *Configuration*.

```
##################################################
# Akka Beanstalk Mailboxes Reference Config File #
##################################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.
#
# for more information see <https://github.com/kr/beanstalkd/blob/v1.3/doc/protocol.txt>

akka {
  actor {
    mailbox {
      beanstalk {
        # hostname to connect to
        hostname = "127.0.0.1"

        # port to connect to
        port = 11300

        # wait period in case of a connection failure before reconnect
        reconnect-window = 5s

        # integer number of seconds to wait before putting the job in
        # the ready queue. The job will be in the "delayed" state during this time.
        message-submit-delay = 0s
```

```
        # time to run -- is an integer number of seconds to allow a worker
        # to run this job. This time is counted from the moment a worker reserves
        # this job. If the worker does not delete, release, or bury the job within
        # <ttr> seconds, the job will time out and the server will release the job.
        message-time-to-live = 120s
      }
    }
  }

}
```

7.1.6 MongoDB-based Durable Mailboxes

Warning: IMPORTANT
In order to streamline the distribution, we have deprecated all durable mailboxes except for the FileBasedMailbox, which will serve as a blueprint for how to implement new ones and to serve as the default durable mailbox implementation.

This mailbox is backed by MongoDB. MongoDB is a fast, lightweight and scalable document-oriented database. It contains a number of features cohesive to a fast, reliable & durable queueing mechanism which the Akka Mailbox takes advantage of.

Akka's implementations of MongoDB mailboxes are built on top of the purely asynchronous MongoDB driver (often known as Hammersmith and `com.mongodb.async`) and as such are purely callback based with a Netty network layer. This makes them extremely fast & lightweight versus building on other MongoDB implementations such as mongo-java-driver and Casbah.

You configure durable mailboxes through the dispatcher, as described in *General Usage* with the following mailbox type.

Config:

```
my-dispatcher {
  mailbox-type = akka.actor.mailbox.MongoBasedMailboxType
}
```

You will need to configure the URI for the MongoDB server, using the URI Format specified in the MongoDB Documentation. This is done in the `akka.actor.mailbox.mongodb` section in the *Configuration*.

```
############################################
# Akka MongoDB Mailboxes Reference Config File #
############################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka {
  actor {
    mailbox {
      mongodb {

        # Any specified collection name will be used as a prefix for
        # collections that use durable mongo mailboxes.
        # Follow Mongo URI Spec - http://www.mongodb.org/display/DOCS/Connections
        uri = "mongodb://localhost/akka.mailbox"

        # Configurable timeouts for certain ops
        timeout {
          # time to wait for a read to succeed before timing out the future
          read = 3000ms
          # time to wait for a write to succeed before timing out the future
```

```
            write = 3000ms
          }
        }
      }
    }
  }
}
```

You must specify a hostname (and optionally port) and at *least* a Database name. If you specify a collection name, it will be used as a 'prefix' for the collections Akka creates to store mailbox messages. Otherwise, collections will be prefixed with `mailbox.`

It is also possible to configure the timeout thresholds for Read and Write operations in the `timeout` block.

7.2 HTTP

7.2.1 Play2-mini

The Akka team recommends the Play2-mini framework when building RESTful service applications that integrates with Akka. It provides a REST API on top of Play2.

7.2.2 Getting started

First you must make your application aware of play-mini. In SBT you just have to add the following to your _libraryDependencies_:

```
libraryDependencies += "com.typesafe" %% "play-mini" % "<version-number>"
```

7.3 Camel

Note: The Akka Camel module has not been migrated to Akka 2.0.4 yet.

It might not make it into Akka 2.0 final but will then hopefully be re-introduce in an upcoming release. It might also be backported to 2.0 final.

7.4 Spring Integration

Note: The Akka Spring module has not been migrated to Akka 2.0.4 yet.

It might not make it into Akka 2.0 final but will then hopefully be re-introduce in an upcoming release. It might also be backported to 2.0 final.

CHAPTER

EIGHT

INFORMATION FOR DEVELOPERS

8.1 Building Akka

This page describes how to build and run Akka from the latest source code.

8.1.1 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 repository from http://github.com/akka/akka.

For example:

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

If you have already cloned the repository previously then you can update the code with `git pull`:

```
git pull origin master
```

8.1.2 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/AkkaBuild.scala`.

8.1.3 Building Akka

First make sure that you are in the akka code directory:

```
cd akka
```

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.

Parallel Execution

By default the tests are executed sequentially. They can be executed in parallel to reduce build times, if hardware can handle the increased memory and cpu usage. Add the following system property to sbt launch script to activate parallel execution:

```
-Dakka.parallelExecution=true
```

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
```

sbt interactive mode

Note that in the examples above we are calling `sbt compile` and `sbt test` and so on, but 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] Set current project to default (in build file:/.../akka/project/plugins/)
[info] Set current project to akka (in build file:/.../akka/)
> compile
...
> test
...
```

sbt batch mode

It's also possible to combine commands in a single call. For example, testing, and publishing Akka to the local Ivy repository can be done with:

```
sbt test publish-local
```

8.1.4 Dependencies

You can look at the Ivy dependency resolution information that is created on `sbt update` and found in `~/.ivy2/cache`. For example, the `~/.ivy2/cache/com.typesafe.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.

8.2 Multi-JVM Testing

Support for running applications (objects with main methods) and ScalaTest tests in multiple JVMs.

8.2.1 Setup

The multi-JVM testing is an sbt plugin that you can find here:

http://github.com/typesafehub/sbt-multi-jvm

You can add it as a plugin by adding the following to your project/plugins.sbt:

```
resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.typesafe.sbtmultijvm" % "sbt-multi-jvm" % "0.1.9")
```

You can then add multi-JVM testing to `project/Build.scala` by including the `MultiJvm` settings and config. For example, here is how the akka-remote project adds multi-JVM testing:

```
import sbt._
import Keys._
import com.typesafe.sbtmultijvm.MultiJvmPlugin
import com.typesafe.sbtmultijvm.MultiJvmPlugin.{ MultiJvm, extraOptions }

object AkkaBuild extends Build {

  lazy val remote = Project(
    id = "akka-remote",
    base = file("akka-remote"),
    settings = defaultSettings ++ MultiJvmPlugin.settings ++ Seq(
      extraOptions in MultiJvm <<= (sourceDirectory in MultiJvm) { src =>
         (name: String) => (src ** (name + ".conf")).get.headOption.map("-Dconfig.file=" + _.abso
      },
      test in Test <<= (test in Test) dependsOn (test in MultiJvm)
    )
  ) configs (MultiJvm)

  lazy val buildSettings = Defaults.defaultSettings ++ Seq(
    organization := "com.typesafe.akka",
    version      := "2.0.4",
    scalaVersion := "2.9.1",
    crossPaths   := false
  )

  lazy val defaultSettings = buildSettings ++ Seq(
    resolvers += "Typesafe Repo" at "http://repo.typesafe.com/typesafe/releases/"
  )

}
```

You can specify JVM options for the forked JVMs:

```
jvmOptions in MultiJvm := Seq("-Xmx256M")
```

8.2.2 Running tests

The multi-jvm tasks are similar to the normal tasks: `test`, `test-only`, and `run`, but are under the `multi-jvm` configuration.

So in Akka, to run all the multi-JVM tests in the akka-remote project use (at the sbt prompt):

```
akka-remote/multi-jvm:test
```

Or one can change to the `akka-remote` project first, and then run the tests:

```
project akka-remote
multi-jvm:test
```

To run individual tests use `test-only`:

```
multi-jvm:test-only akka.remote.RandomRoutedRemoteActor
```

More than one test name can be listed to run multiple specific tests. Tab-completion in sbt makes it easy to complete the test names.

It's also possible to specify JVM options with `test-only` by including those options after the test names and `--`. For example:

```
multi-jvm:test-only akka.remote.RandomRoutedRemoteActor -- -Dsome.option=something
```

8.2.3 Creating application tests

The tests are discovered, and combined, through a naming convention. MultiJvm tests are located in `src/multi-jvm/scala` directory. A test is named with the following pattern:

```
{TestName}MultiJvm{NodeName}
```

That is, each test has `MultiJvm` in the middle of its name. The part before it groups together tests/applications under a single `TestName` that will run together. The part after, the `NodeName`, is a distinguishing name for each forked JVM.

So to create a 3-node test called `Sample`, you can create three applications like the following:

```
package sample

object SampleMultiJvmNode1 {
  def main(args: Array[String]) {
    println("Hello from node 1")
  }
}

object SampleMultiJvmNode2 {
  def main(args: Array[String]) {
    println("Hello from node 2")
  }
}

object SampleMultiJvmNode3 {
  def main(args: Array[String]) {
    println("Hello from node 3")
  }
}
```

When you call `multi-jvm:run sample.Sample` at the sbt prompt, three JVMs will be spawned, one for each node. It will look like this:

```
> multi-jvm:run sample.Sample
...
[info] Starting JVM-Node1 for sample.SampleMultiJvmNode1
[info] Starting JVM-Node2 for sample.SampleMultiJvmNode2
[info] Starting JVM-Node3 for sample.SampleMultiJvmNode3
[JVM-Node1] Hello from node 1
[JVM-Node2] Hello from node 2
[JVM-Node3] Hello from node 3
[success] Total time: ...
```

8.2.4 Naming

You can change what the `MultiJvm` identifier is. For example, to change it to `ClusterTest` use the `multiJvmMarker` setting:

```
multiJvmMarker in MultiJvm := "ClusterTest"
```

Your tests should now be named `{TestName}ClusterTest{NodeName}`.

8.2.5 Configuration of the JVM instances

Setting JVM options

You can define specific JVM options for each of the spawned JVMs. You do that by creating a file named after the node in the test with suffix `.opts` and put them in the same directory as the test.

For example, to feed the JVM options `-Dakka.remote.port=9991` to the `SampleMultiJvmNode1` let's create three `*.opts` files and add the options to them.

`SampleMultiJvmNode1.opts`:

```
-Dakka.remote.port=9991
```

`SampleMultiJvmNode2.opts`:

```
-Dakka.remote.port=9992
```

`SampleMultiJvmNode3.opts`:

```
-Dakka.remote.port=9993
```

Overriding configuration options

You can also override the options in the *Configuration* file with different options for each spawned JVM. You do that by creating a file named after the node in the test with suffix `.conf` and put them in the same directory as the test .

For example, to override the configuration option `akka.cluster.name` let's create three `*.conf` files and add the option to them.

`SampleMultiJvmNode1.conf`:

```
akka.remote.port = 9991
```

`SampleMultiJvmNode2.conf`:

```
akka.remote.port = 9992
```

`SampleMultiJvmNode3.conf`:

```
akka.remote.port = 9993
```

8.2.6 ScalaTest

There is also support for creating ScalaTest tests rather than applications. To do this use the same naming convention as above, but create ScalaTest suites rather than objects with main methods. You need to have ScalaTest on the classpath. Here is a similar example to the one above but using ScalaTest:

```
package sample

import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers

class SpecMultiJvmNode1 extends WordSpec with MustMatchers {
  "A node" should {
    "be able to say hello" in {
```

```
      val message = "Hello from node 1"
      message must be("Hello from node 1")
    }
  }
}

class SpecMultiJvmNode2 extends WordSpec with MustMatchers {
  "A node" should {
    "be able to say hello" in {
      val message = "Hello from node 2"
      message must be("Hello from node 2")
    }
  }
}
```

To run just these tests you would call `multi-jvm:test-only sample.Spec` at the sbt prompt.

8.2.7 Barriers

When running multi-JVM tests it's common to need to coordinate timing across nodes. To do this, multi-JVM test framework has the notion of a double-barrier (there is both an entry barrier and an exit barrier). To wait at the entry use the `enter` method. To wait at the exit use the `leave` method. It's also possible to pass a block of code which will be run between the barriers.

There are 2 implementations of the barrier: one is used for coordinating JVMs running on a single machine and is based on local files, another used in a distributed scenario (see below) and is based on apache ZooKeeper. These two cases are differentiated with `test.hosts` property defined. The choice for a proper barrier implementation is made in `AkkaRemoteSpec` which is a base class for all multi-JVM tests.

When creating a barrier you pass it a name. You can also pass a timeout. The default timeout is 60 seconds.

Here is an example of coordinating the starting of two nodes and then running something in coordination:

```
package sample

import org.scalatest.WordSpec
import org.scalatest.matchers.MustMatchers
import org.scalatest.BeforeAndAfterAll

import akka.cluster._

object SampleMultiJvmSpec extends AbstractRemoteActorMultiJvmSpec {
  val NrOfNodes = 2
  def commonConfig = ConfigFactory.parseString("""
    // Declare your configuration here.
  """)
}

class SampleMultiJvmNode1 extends AkkaRemoteSpec(SampleMultiJvmSpec.nodeConfigs(0))
  with WordSpec with MustMatchers {
  import SampleMultiJvmSpec._

  "A cluster" must {

    "have jvm options" in {
      System.getProperty("akka.remote.port", "") must be("9991")
      akka.config.Config.config.getString("test.name", "") must be("node1")
    }

    "be able to start all nodes" in {
      barrier("start")
      println("All nodes are started!")
```

```
      barrier("end")
    }
  }
}

class SampleMultiJvmNode2 extends AkkaRemoteSpec(SampleMultiJvmSpec.nodeConfigs(1))
  with WordSpec with MustMatchers {
  import SampleMultiJvmSpec._

  "A cluster" must {

    "have jvm options" in {
      System.getProperty("akka.remote.port", "") must be("9992")
      akka.config.Config.config.getString("test.name", "") must be("node2")
    }

    "be able to start all nodes" in {
      barrier("start")
      println("All nodes are started!")
      barrier("end")
    }
  }
}
```

8.2.8 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

8.2.9 Running tests on many machines

The same tests that are run on a single machine using sbt-multi-jvm can be run on multiple machines using schoir (read the same as `esquire`) plugin. The plugin is included just like sbt-multi-jvm:

```
resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.typesafe.schoir" % "schoir" % "0.1.1")
```

The interaction with the plugin is through `schoir:master` input task. This input task optionally accepts the path to the file with the following properties:

```
git.url=git@github.com:akka/akka.git
external.addresses.for.ssh=host1:port1,...,hostN:portN
internal.host.names=host1,...,hostN
```

Alternative to specifying the property file, one can set respective settings in the build file:

```
gitUrl := "git@github.com:akka/akka.git",
machinesExt := List(InetAddress("host1", port1)),
machinesInt := List("host1")
```

The reason the first property is called `git.url` is that the plugin sets up a temporary remote branch on git to test against the local working copy. After the tests are finished the changes are regained and the branch is deleted.

Each test machine starts a node in zookeeper server ensemble that can be used for synchronization. Since the server is started on a fixed port, it's not currently possible to run more than one test session on the same machine at the same time.

The machines that are used for testing (slaves) should have ssh access to the outside world and be able to talk to each other with the internal addresses given. On the master machine ssh client is required. Obviosly git and sbt should be installed on both master and slave machines.

8.3 Developer Guidelines

8.3.1 Code Style

The Akka code style follows this document .

Akka is using `Scalariform` to format the source code as part of the build. So just hack away and then run `sbt compile` and it will reformat the code according to Akka standards.

8.3.2 Process

- Make sure you have signed the Akka CLA, if not, ask for it on the Mailing List.
- Pick a ticket, if there is no ticket for your work then create one first.
- Start working in a feature branch. Name it something like `wip-<ticket number>-<descriptive name>-<your username>`.
- When you are done, create a GitHub Pull-Request towards the targeted branch and email the Akka Mailing List that you want it reviewed
- When there's consensus on the review, someone from the Akka Core Team will merge it.

8.3.3 Commit messages

Please follow these guidelines when creating public commits and writing commit messages.

1. If your work spans multiple local commits (for example; if you do safe point commits while working in a topic branch or work in a branch for long time doing merges/rebases etc.) then please do **not** commit it all but rewrite the history by squashing the commits into a single big commit which you write a good commit message for (like discussed below). Here is a great article for how to do that: http://sandofsky.com/blog/git-workflow.html. Every commit should be able to be used in isolation, cherry picked etc.
2. First line should be a descriptive sentence what the commit is doing. It should be possible to fully understand what the commit does by just reading this single line. It is **not** ok to only list the ticket number, type "minor fix" or similar. Include reference to ticket number, prefixed with #, at the end of the first line. If the commit is a **small** fix, then you are done. If not, go to 3.
3. Following the single line description should be a blank line followed by an enumerated list with the details of the commit.

Example:

```
Completed replication over BookKeeper based transaction log with configurable actor snapshotting e

  * Details 1
  * Details 2
  * Details 3
```

8.3.4 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 (Scala)* module.

Multi-JVM Testing

Included in the example is an sbt trait for multi-JVM testing which will fork JVMs for multi-node testing. There is support for running applications (objects with main methods) and running ScalaTest tests.

NetworkFailureTest

You can use the 'NetworkFailureTest' trait to test network failure.

8.4 Documentation Guidelines

The Akka documentation uses reStructuredText as its markup language and is built using Sphinx.

8.4.1 Sphinx

More to come...

8.4.2 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".
```

8.4.3 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
```

8.5 Team

Name	Role	Email
Jonas Bonér	Founder, Despot, Committer	jonas AT jonasboner DOT com
Viktor Klang	Project Owner	viktor DOT klang AT gmail DOT com
Roland Kuhn	Committer	
Patrik Nordwall	Committer	patrik DOT nordwall AT gmail DOT com
Derek Williams	Committer	derek AT nebvin DOT ca
Henrik Engström	Committer	
Peter Vlugter	Committer	
Martin Krasser	Committer	krasserm AT googlemail DOT com
Raymond Roestenburg	Committer	
Piotr Gabryanczyk	Committer	
Debasish Ghosh	Alumni	dghosh AT acm DOT org
Ross McDonald	Alumni	rossajmcd AT gmail DOT com
Eckhart Hertzler	Alumni	
Mikael Högqvist	Alumni	
Tim Perrett	Alumni	
Jeanfrancois Arcand	Alumni	jfarcand AT apache DOT org
Jan Van Besien	Alumni	
Michael Kober	Alumni	
Peter Veentjer	Alumni	
Irmo Manie	Alumni	
Heiko Seeberger	Alumni	
Hiram Chirino	Alumni	
Scott Clasen	Alumni	

CHAPTER

NINE

PROJECT INFORMATION

9.1 Migration Guides

9.1.1 Migration Guide 1.3.x to 2.0.x

The 2.0 release contains several new features which require source-level changes in client code. This API cleanup is planned to be the last one for a significant amount of time.

New Concepts

First you should take some time to understand the new concepts of *Actor Systems*, *Supervision and Monitoring*, and *Actor References, Paths and Addresses*.

Migration Kit

Nobody likes a big refactoring that takes several days to complete until anything is able to run again. Therefore we provide a migration kit that makes it possible to do the migration changes in smaller steps.

The migration kit only covers the most common usage of Akka. It is not intended as a final solution. The whole migration kit is marked as deprecated and will be removed in Akka 2.1.

The migration kit is provided in separate jar files. Add the following dependency:

```
"com.typesafe.akka" % "akka-actor-migration" % "2.0.4"
```

The first step of the migration is to do some trivial replacements. Search and replace the following (be careful with the non qualified names):

Search	Replace with
`akka.actor.Actor`	`akka.actor.OldActor`
`extends Actor`	`extends OldActor`
`akka.actor.Scheduler`	`akka.actor.OldScheduler`
`Scheduler`	`OldScheduler`
`akka.event.EventHandler`	`akka.event.OldEventHandler`
`EventHandler`	`OldEventHandler`
`akka.config.Config`	`akka.config.OldConfig`
`Config`	`OldConfig`

For Scala users the migration kit also contains some implicit conversions to be able to use some old methods. These conversions are useful from tests or other code used outside actors.

```
import akka.migration._
```

Thereafter you need to fix compilation errors that are not handled by the migration kit, such as:

- Definition of supervisors

- Definition of dispatchers
- ActorRegistry

When everything compiles you continue by replacing/removing the `OldXxx` classes one-by-one from the migration kit with appropriate migration.

When using the migration kit there will be one global actor system, which loads the configuration `akka.conf` from the same locations as in Akka 1.x. This means that while you are using the migration kit you should not create your own `ActorSystem`, but instead use the `akka.actor.GlobalActorSystem`. In order to voluntarily exit the JVM you must `shutdown` the `GlobalActorSystem` Last task of the migration would be to create your own `ActorSystem`.

Unordered Collection of Migration Items

Actors

Creating and starting actors Actors are created by passing in a `Props` instance into the actorOf factory method in a `ActorRefProvider`, which is the `ActorSystem` or `ActorContext`. Use the system to create top level actors. Use the context to create actors from other actors. The difference is how the supervisor hierarchy is arranged. When using the context the current actor will be supervisor of the created child actor. When using the system it will be a top level actor, that is supervised by the system (internal guardian actor).

`ActorRef.start()` has been removed. Actors are now started automatically when created. Remove all invocations of `ActorRef.start()`.

v1.3:

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

v2.0:

```
// top level actor
val firstActor = system.actorOf(Props[FirstActor], name = "first")

// child actor
class FirstActor extends Actor {
  val myActor = context.actorOf(Props[MyActor], name = "myactor")
```

Documentation:

- *Actors (Scala)*
- *Actors (Java)*

Stopping actors `ActorRef.stop()` has been moved. Use `ActorSystem` or `ActorContext` to stop actors.

v1.3:

```
actorRef.stop()
self.stop()
actorRef ! PoisonPill
```

v2.0:

```
context.stop(someChild)
context.stop(self)
system.stop(actorRef)
actorRef ! PoisonPill
```

Stop all actors

v1.3:

```
ActorRegistry.shutdownAll()
```

v2.0:

```
system.shutdown() // from outside of this system
context.system.shutdown() // from inside any actor
```

Documentation:

- *Actors (Scala)*
- *Actors (Java)*

Identifying Actors In v1.3 actors have `uuid` and `id` field. In v2.0 each actor has a unique logical `path`.

The `ActorRegistry` has been replaced by actor paths and lookup with `actorFor` in `ActorRefProvider` (`ActorSystem` or `ActorContext`). It is no longer possible to obtain references to all actors being implemented by a certain class (the reason being that this property is not known yet when an `ActorRef` is created because instantiation of the actor itself is asynchronous).

v1.3:

```
val actor =  Actor.registry.actorFor(uuid)
val actors =  Actor.registry.actorsFor(id)
```

v2.0:

```
val actor = context.actorFor("/user/serviceA/aggregator")
```

Documentation:

- *Actor References, Paths and Addresses*
- *Actors (Scala)*
- *Actors (Java)*

Reply to messages `self.channel` has been replaced with unified reply mechanism using `sender` (Scala) or `getSender()` (Java). This works for both tell (!) and ask (?). Sending to an actor reference never throws an exception, hence `tryTell` and `tryReply` are removed.

v1.3:

```
self.channel ! result
self.channel tryTell result
self.reply(result)
self.tryReply(result)
```

v2.0:

```
sender ! result
```

Documentation:

- *Actors (Scala)*
- *Actors (Java)*

`ActorRef.ask()` The mechanism for collecting an actor's reply in a `Future` has been reworked for better location transparency: it uses an actor under the hood. This actor needs to be disposable by the garbage collector in case no reply is ever received, and the decision is based upon a timeout. This timeout determines when the actor will stop itself and hence closes the window for a reply to be received; it is independent of the timeout applied when awaiting completion of the `Future`, however, the actor will complete the `Future` with an `AskTimeoutException` when it stops itself.

Since there is no good library default value for the ask-timeout, specification of a timeout is required for all usages as shown below.

Also, since the `ask` feature is coupling futures and actors, it is no longer offered on the `ActorRef` itself, but instead as a use pattern to be imported. While Scala's implicit conversions enable transparent replacement, Java code will have to be changed by more than just adding an import statement.

v1.3:

```
actorRef ? message // Scala
actorRef.ask(message, timeout); // Java
```

v2.0 (Scala):

```
import akka.pattern.ask

implicit val timeout: Timeout = ...
actorRef ? message              // uses implicit timeout
actorRef ask message            // uses implicit timeout
actorRef.ask(message)(timeout)  // uses explicit timeout
ask(actorRef, message)          // uses implicit timeout
ask(actorRef, message)(timeout) // uses explicit timeout
```

v2.0 (Java):

```
import akka.pattern.Patterns;

Patterns.ask(actorRef, message, timeout)
```

Documentation:

- *Actors (Scala)*
- *Actors (Java)*

`ActorRef.?(msg, timeout)` This method has a dangerous overlap with `ActorRef.?(msg)(implicit timeout)` due to the fact that Scala allows to pass a `Tuple` in place of the message without requiring extra parentheses:

```
actor ? (1, "hallo") // will send a tuple
actor ? (1, Timeout()) // will send 1 with an explicit timeout
```

To remove this ambiguity, the latter variant is removed in version 2.0. If you were using it before, it will now send tuples where that is not desired. In order to correct all places in the code where this happens, simply import `akka.migration.ask` instead of `akka.pattern.ask` to obtain a variant which will give deprecation warnings where the old method signature is used:

```
import akka.migration.ask

actor ? (1, Timeout(2 seconds)) // will give deprecation warning
```

`UntypedActor.getContext()` (Java API only) `getContext()` in the Java API for UntypedActor is renamed to `getSelf()`.

v1.3:

```
actorRef.tell("Hello", getContext());
```

v2.0:

```
actorRef.tell("Hello", getSelf());
```

Documentation:

- *Actors (Java)*

Configuration A new, more powerful, configuration utility has been implemented. The format of the configuration file is very similar to the format in v1.3. In addition it also supports configuration files in json and properties format. The syntax is described in the HOCON specification.

v1.3:

```
include "other.conf"

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

v2.0:

```
include "other"

akka {
  loglevel = "DEBUG"
}
```

In v1.3 the default name of the configuration file was `akka.conf`. In v2.0 the default name is `application.conf`. It is still loaded from classpath or can be specified with java System properties (`-D` command line arguments).

v1.3:

```
-Dakka.config=<file path to configuration file>
-Dakka.output.config.source=on
```

v2.0:

```
-Dconfig.file=<file path to configuration file>
-Dakka.log-config-on-start=on
```

Several configuration properties have been changed, such as:

- `akka.event-handler-level` => `akka.loglevel`
- dispatcher `type` values are changed
- `akka.actor.throughput` => `akka.actor.default-dispatcher.throughput`
- `akka.remote.layer` => `akka.remote.transport`
- the global `time-unit` property is removed, all durations are specified with duration unit in the property value, `timeout = 5s`

Verify used configuration properties against the reference *Configuration*.

Documentation:

- *Configuration*

Logging EventHandler API has been replaced by LoggingAdapter, which publish log messages to the event bus. You can still plugin your own actor as event listener with the `akka.event-handlers` configuration property.

v1.3:

```
EventHandler.error(exception, this, message)
EventHandler.warning(this, message)
EventHandler.info(this, message)
EventHandler.debug(this, message)
EventHandler.debug(this, "Processing took %s ms".format(duration))
```

v2.0:

```
import akka.event.Logging

val log = Logging(context.system, this) // will include system name in message source
val log = Logging(system.eventStream, getClass.getName) // will not include system name
log.error(exception, message)
log.warning(message)
log.info(message)
log.debug(message)
log.debug("Processing took {} ms", duration)
```

Documentation:

- *Logging (Scala)*
- *Logging (Java)*
- *Event Bus (Scala)*
- *Event Bus (Java)*

Scheduler The functionality of the scheduler is identical, but the API is slightly adjusted.

v1.3:

```
//Schedules to send the "foo"-message to the testActor after 50ms
Scheduler.scheduleOnce(testActor, "foo", 50L, TimeUnit.MILLISECONDS)

// Schedules periodic send of "foo"-message to the testActor after 1s inital delay,
// and then with 200ms between successive sends
Scheduler.schedule(testActor, "foo", 1000L, 200L, TimeUnit.MILLISECONDS)

// Schedules a function to be executed (send the current time) to the testActor after 50ms
Scheduler.scheduleOnce({testActor ! System.currentTimeMillis}, 50L, TimeUnit.MILLISECONDS)
```

v2.0:

```
//Schedules to send the "foo"-message to the testActor after 50ms
system.scheduler.scheduleOnce(50 milliseconds, testActor, "foo")

// Schedules periodic send of "foo"-message to the testActor after 1s inital delay,
// and then with 200ms between successive sends
system.scheduler.schedule(1 second, 200 milliseconds, testActor, "foo")

// Schedules a function to be executed (send the current time) to the testActor after 50ms
system.scheduler.scheduleOnce(50 milliseconds) {
  testActor ! System.currentTimeMillis
}
```

The internal implementation of the scheduler is changed from `java.util.concurrent.ScheduledExecutorService` to a variant of `org.jboss.netty.util.HashedWheelTimer`.

Documentation:

- *Scheduler (Scala)*
- *Scheduler (Java)*

Supervision Akka v2.0 implements parental supervision. Actors can only be created by other actors — where the top-level actor is provided by the library — and each created actor is supervised by its parent. In contrast to the special supervision relationship between parent and child, each actor may monitor any other actor for termination.

v1.3:

```
self.link(actorRef)
self.unlink(actorRef)
```

v2.0:

```
class WatchActor extends Actor {
  val actorRef = ...
  // Terminated message will be delivered when the actorRef actor
  // is stopped
  context.watch(actorRef)

  val supervisedChild = context.actorOf(Props[ChildActor])

  def receive = {
    case Terminated(`actorRef`) ⇒ ...
  }
}
```

Note that `link` in v1.3 established a supervision relation, which `watch` doesn't. `watch` is only a way to get notification, `Terminated` message, when the monitored actor has been stopped.

Refererence to the supervisor

v1.3:

```
self.supervisor
```

v2.0:

```
context.parent
```

Supervisor Strategy

v1.3:

```
val supervisor = Supervisor(
  SupervisorConfig(
    OneForOneStrategy(List(classOf[Exception]), 3, 1000),
    Supervise(
      actorOf[MyActor1],
      Permanent) ::
    Supervise(
      actorOf[MyActor2],
      Permanent) ::
    Nil))
```

v2.0:

```
class MyActor extends Actor {
  override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 mi
      case _: ArithmeticException      ⇒ Resume
      case _: NullPointerException     ⇒ Restart
      case _: IllegalArgumentException ⇒ Stop
      case _: Exception                ⇒ Escalate
    }
```

```
  def receive = {
    case x =>
  }
}
```

Documentation:

- *Supervision and Monitoring*
- *Fault Tolerance (Java)*
- *Fault Tolerance (Scala)*
- *Actors (Scala)*
- *Actors (Java)*

Dispatchers Dispatchers are defined in configuration instead of in code.

v1.3:

```
// in code
val myDispatcher = Dispatchers.newExecutorBasedEventDrivenDispatcher(name)
  .withNewThreadPoolWithLinkedBlockingQueueWithCapacity(100)
  .setCorePoolSize(16)
  .setMaxPoolSize(128)
  .setKeepAliveTimeInMillis(60000)
  .build
```

v2.0:

```
// in config
my-dispatcher {
  type = Dispatcher
  core-pool-size-factor = 8.0
  max-pool-size-factor  = 16.0
  mailbox-capacity = 100
}
```

The dispatcher is assigned to the actor in a different way.

v1.3:

```
actorRef.dispatcher = MyGlobals.myDispatcher
self.dispatcher = MyGlobals.myDispatcher
```

v2.0:

```
val myActor = system.actorOf(Props[MyActor].withDispatcher("my-dispatcher"), "myactor")
```

Documentation:

- *Dispatchers (Java)*
- *Dispatchers (Scala)*

Spawn `spawn` has been removed and should be replaced by creating a `Future`. Be careful to not access any shared mutable state closed over by the body.

Scala:

```
Future { doSomething() } // will be executed asynchronously
```

Java:

```
Futures.future<String>(new Callable<String>() {
  public String call() {
    doSomething();
  }
}, executionContext);
```

Documentation:

- *Futures (Scala)*
- *Futures (Java)*
- *Akka and the Java Memory Model*

HotSwap In v2.0 `become` and `unbecome` metods are located in `ActorContext`, i.e. `context.become` and `context.unbecome`.

The special `HotSwap` and `RevertHotswap` messages in v1.3 has been removed. Similar can be implemented with your own message and using `context.become` and `context.unbecome` in the actor receiving the message. The rationale is that being able to replace any actor's behavior generically is not a good idea because actor implementors would have no way to defend against that; hence the change to lay it into the hands of the actor itself.

- *Actors (Scala)*
- *Actors (Java)*

Routing Routing has been redesign with improved performance and additional features as a result.

v1.3:

```
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))
}
val loadbalancer = actorOf[MyLoadBalancer].start()
```

v2.0:

```
val pinger = system.actorOf(Props(new Actor { def receive = { case x ⇒ println("Pinger: " + x) }
val ponger = system.actorOf(Props(new Actor { def receive = { case x ⇒ println("Ponger: " + x) }
val loadbalancer = system.actorOf(Props().withRouter(RoundRobinRouter(Seq(pinger, ponger))))
```

Documentation:

- *Routing (Scala)*
- *Routing (Java)*

ActorPool The ActorPool has been replaced by dynamically resizable routers.

v1.3:

```
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[ExampleActor]
}
```

v2.0:

```
// in configuration
akka.actor.deployment {
  /router2 {
    router = round-robin
    resizer {
      lower-bound = 2
      upper-bound = 15
    }
  }
}

// in code
val router2 = system.actorOf(Props[ExampleActor].withRouter(FromConfig())
```

Documentation:

- *Routing (Scala)*
- *Routing (Java)*

STM

In Akka v2.0 ScalaSTM is used rather than Multiverse.

Agent and Transactor have been ported to ScalaSTM. The API's for Agent and Transactor are basically the same, other than integration with ScalaSTM. See:

- *Agents (Scala)*
- *Agents (Java)*
- *Transactors (Scala)*
- *Transactors (Java)*

Imports

Scala To use ScalaSTM the import from Scala is:

```
import scala.concurrent.stm._
```

Java For Java there is a special helper object with Java-friendly methods:

```
import scala.concurrent.stm.japi.STM;
```

These methods can also be statically imported:

```
import static scala.concurrent.stm.japi.STM.*;
```

Other imports that are needed are in the stm package, particularly `Ref`:

```
import scala.concurrent.stm.Ref;
```

Transactions

Scala Both v1.3 and v2.0 provide an `atomic` block, however, the ScalaSTM `atomic` is a function from `InTxn` to return type.

v1.3:

```
atomic {
  // do something in transaction
}
```

v2.0:

```
atomic { implicit txn =>
  // do something in transaction
}
```

Note that in ScalaSTM the `InTxn` in the atomic function is usually marked as implicit as transactional references require an implicit `InTxn` on all methods. That is, the transaction is statically required and it is a compile-time warning to use a reference without a transaction. There is also a `Ref.View` for operations without requiring an `InTxn` statically. See below for more information.

Java In the ScalaSTM Java API helpers there are atomic methods which accept `java.lang.Runnable` and `java.util.concurrent.Callable`.

v1.3:

```
new Atomic() {
    public Object atomically() {
        // in transaction
        return null;
    }
}.execute();

SomeObject result = new Atomic<SomeObject>() {
    public SomeObject atomically() {
        // in transaction
        return ...;
    }
}.execute();
```

v2.0:

```
import static scala.concurrent.stm.japi.STM.atomic;
import java.util.concurrent.Callable;

atomic(new Runnable() {
    public void run() {
        // in transaction
    }
});

SomeObject result = atomic(new Callable<SomeObject>() {
    public SomeObject call() {
        // in transaction
        return ...;
    }
});
```

Ref

Scala Other than the import, creating a Ref is basically identical between Akka STM in v1.3 and ScalaSTM used in v2.0.

v1.3:

```
val ref = Ref(0)
```

v2.0:

```
val ref = Ref(0)
```

The API for Ref is similar. For example:

v1.3:

```
ref.get // get current value
ref()   // same as get

ref.set(1)  // set to new value, return old value
ref() = 1   // same as set
ref.swap(2) // same as set

ref alter { _ + 1 } // apply a function, return new value
```

v2.0:

```
ref.get // get current value
ref()   // same as get

ref.set(1)  // set to new value, return nothing
ref() = 1   // same as set
ref.swap(2) // set and return old value

ref transform { _ + 1 } // apply function, return nothing

ref transformIfDefined { case 1 => 2 } // apply partial function if defined
```

Ref.View In v1.3 using a `Ref` method outside of a transaction would automatically create a single-operation transaction. In v2.0 (in ScalaSTM) there is a `Ref.View` which provides methods without requiring a current transaction.

Scala The `Ref.View` can be accessed with the `single` method:

```
ref.single() // returns current value
ref.single() = 1 // set new value

// with atomic this would be:

atomic { implicit t => ref() }
atomic { implicit t => ref() = 1 }
```

Java As `Ref.View` in ScalaSTM does not require implicit transactions, this is more easily used from Java. `Ref` could be used, but requires explicit threading of transactions. There are helper methods in `japi.STM` for creating `Ref.View` references.

v1.3:

```
Ref<Integer> ref = new Ref<Integer>(0);
```

v2.0:

```
Ref.View<Integer> ref = STM.newRef(0);
```

The `set` and `get` methods work the same way for both versions.

v1.3:

```
ref.get();  // get current value
ref.set(1); // set new value
```

v2.0:

```
ref.get();  // get current value
ref.set(1); // set new value
```

There are also `transform`, `getAndTransform`, and `transformAndGet` methods in `japi.STM` which accept `japi.STM.Transformer` objects.

There are `increment` helper methods for `Ref.View<Integer>` and `Ref.View<Long>` references.

Transaction lifecycle callbacks

Scala It is also possible to hook into the transaction lifecycle in ScalaSTM. See the ScalaSTM documentation for the full range of possibilities.

v1.3:

```
atomic {
  deferred {
    // executes when transaction commits
  }
  compensating {
    // executes when transaction aborts
  }
}
```

v2.0:

```
atomic { implicit txn =>
  txn.afterCommit { txnStatus =>
    // executes when transaction commits
  }
  txn.afterRollback { txnStatus =>
    // executes when transaction rolls back
  }
}
```

Java Rather than using the `deferred` and `compensating` methods in `akka.stm.StmUtils`, use the `afterCommit` and `afterRollback` methods in `scala.concurrent.stm.japi.STM`, which behave in the same way and accept `Runnable`.

Transactional Datastructures In ScalaSTM see `TMap`, `TSet`, and `TArray` for transactional datastructures.

There are helper methods for creating these from Java in `japi.STM`: `newTMap`, `newTSet`, and `newTArray`. These datastructures implement the `scala.collection` interfaces and can also be used from Java with Scala's `JavaConversions`. There are helper methods that apply the conversions, returning `java.util` `Map`, `Set`, and `List`: `newMap`, `newSet`, and `newArrayAsList`.

9.2 Release Notes

9.2.1 Release 2.0

We've just released Akka 2.0 – a revolutionary step in programming for concurrency, fault-tolerance and scalability.

Building on the experiences from the Akka 1 series, we take Akka to the next level— resilience by default, scale up and out by configuration, extensibility by design and with a smaller footprint.

A lot of effort has gone into this release, and it's not just a version bump, it's truly worthy of the name "Akka 2.0". We'd like to take the opportunity to thank all of the excellent people that have made this release possible, to Jonas, to all committers, to all users and to the excellent Scala and Java ecosystems!

Highlights of Akka 2.0

Stats

700 tickets closed!

Code changes compared to Akka 1.3.1:

- 1020 files changed
- 98683 insertions(+)
- 57415 deletions(-)

Also including 331 pages of reference documentation, and tons of ScalaDoc.

Use the Akka Migration package to aid in migrating 1.3 code to 2.0.

Actors

- Distributable by design and asynchronous at the core
- `ActorSystems` lets you run multiple applications in isolation
- `Props` are immutable configuration for Actor instances
- `ActorPaths` makes it dead easy to address Actors
- Enforced parental supervision gives you Error Kernel design for free
- `DeathWatch` makes it possible to observe termination of Actors
- `Routers` unify what was `ActorPools` with `LoadBalancers` to form a flexible, extensible and transparent entity which quacks like an `ActorRef`
- `Stash`, an API made by Phillip Haller to do conditional receives
- Extensions enable you to augment `ActorSystems`
- Scale up and out through configuration
- API unification and simplification
- Excellent performance, up to 20 million msg/second on a single machine, see the blog article
- Slimmer footprint gives you around 2.7 million Actors per GB of memory

Dispatchers

- `Dispatchers` are now configured in the configuration file and not the code, for easy tuning of deployed applications
- `Dispatcher` was previously known as ExecutorBasedEventDrivenDispatcher
- `BalancingDispatcher` was previously known as `ExecutorBasedEventDrivenWorkStealingDispatcher`, is now work-sharing and you can configure which mailbox type should be used
- `PinnedDispatcher` was previously known as `ThreadBasedDispatcher`

- Create your own `Dispatchers` or `MessageQueues` (mailbox backing storage) and hook in through config
- Many different `MessageQueues`: `Priority`, `Bounded`, `Durable` (ZooKeeper, Beanstalk, File, Redis, Mongo)

Remoting

- Completely transparent in user code
- Pluggable transports, ships with a scalable Netty implementation
- Create actors remotely using configuration or in code

TypedActors

- Completely new implementation built on top of JDK Proxies
- 0 external dependencies, so now in akka-actor
- Built as an Akka Extension

Futures & Promises

- Harmonized API with SIP-14 (big props to the EPFL team: Philipp Haller, Aleksandar Prokopec, Heather Miller and Vojin Jovanovic)
- Smaller footprint
- Completely non-blocking implementation

Akka STM

- Now uses ScalaSTM
- `Transactors`
- `Agents`

EventBus

- A simple and easy to use API for Publish/Subscribe

Config

- Now using HOCON, extremely powerful and easy to use
- Big props to Havoc Pennington

Serialization

- Highly pluggable system for serializing objects
- Mappings go into configuration, no need to mix business logic and marshalling
- Built as an Akka Extension

Patterns

- "Ask/?" is now a `Pattern` — for Scala add `import akka.pattern.ask`, for Java use `akka.pattern.Patterns.ask()`.
- `gracefulStop`
- `pipeTo`

ExecutionContext

- One abstraction for asynchronous execution of logic

ømq

- An API for using Akka with ømq
- Huge thanks to Karim Osman and Ivan Porto Carrero
- Built as an Akka Extension

Brand new website, still at http://akka.io, huge thanks to Heather Miller for her outstanding work

Upcoming Releases

Things that will be released within the coming months:

- Akka Camel 2.0, codename "Alpakka", with the excellent work of Raymond Roestenburg and Piotr Gabryanczyk
- Akka AMQP 2.0, with the excellent work of John Stanford
- Akka Spring 2.0, with the excellent help of Josh Long

Akka is released under the Apache V2 license.

Happy hAkking!

9.3 Scaladoc API

9.3.1 Release Versions

2.0

- Akka - http://doc.akka.io/api/akka/2.0/

1.3.1

- Akka 1.3.1 - http://doc.akka.io/api/akka/1.3.1/
- Akka Modules 1.3.1 - http://doc.akka.io/api/akka-modules/1.3.1/

9.3.2 Akka Snapshot

Automatically published Scaladoc API for the latest SNAPSHOT version of Akka can be found here: http://doc.akka.io/api/akka/snapshot

9.4 Documentation for Other Versions

9.4.1 Release Versions

1.3.1

- Akka 1.3.1 - http://akka.io/docs/akka/1.3.1/ (or in PDF format)
- Akka Modules 1.3.1 - http://akka.io/docs/akka-modules/1.3.1/ (or in PDF format)

9.4.2 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)

9.5 Issue Tracking

Akka is using `Assembla` as its issue tracking system.

9.5.1 Browsing

Tickets

You can find the Akka tickets here

Roadmaps

The roadmap for each Akka milestone is here

9.5.2 Creating tickets

In order to create tickets you need to do the following:

Register here then log in

Then you also need to become a "Watcher" of the Akka space.

Link to create a new ticket

Thanks a lot for reporting bugs and suggesting features.

9.5.3 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" must {
    "do YYY" in {
      1 must be (1)
    }
```

```
  }
}
```

9.6 Licenses

9.6.1 Akka License

```
This software is licensed under the Apache 2 license, quoted below.

Copyright 2009-2011 Typesafe Inc. <http://www.typesafe.com>

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.
```

9.6.2 Akka Committer License Agreement

All committers have signed this CLA

```
          Based on: http://www.apache.org/licenses/icla.txt

                      Typesafe Inc.
    Individual Contributor License Agreement ("Agreement") V2.0
             http://www.scalablesolutions.se/licenses/

Thank you for your interest in Akka, a Typesafe Inc. (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: _____
```

9.6.3 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): http://github.com/akka/akka/tree/v2.0.4/project/build/AkkaProject.scala#L127

9.7 Sponsors

9.7.1 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

9.8 Commercial Support

Commercial support is provided by Typesafe. Akka is now part of the Typesafe Stack.

9.9 Mailing List

Akka User Google Group

Akka Developer Google Group

9.10 Downloads

http://akka.io/downloads/

9.11 Source Code

Akka uses Git and is hosted at Github.

- Akka: clone the Akka repository from http://github.com/akka/akka

9.12 Releases Repository

The Akka Maven repository can be found at http://akka.io/releases/.

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/releases/
- 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/

9.13 Snapshots Repository

Nightly builds are available in http://akka.io/snapshots/ and proxied through http://repo.typesafe.com/typesafe/snapshots/ as both `SNAPSHOT` and timestamped versions.

For timestamped versions, pick a timestamp from http://repo.typesafe.com/typesafe/snapshots/com/typesafe/akka/akka-actor/. All Akka modules that belong to the same build have the same timestamp.

Make sure that you add the repository to the sbt resolvers or maven repositories:

```
resolvers += "Typesafe Snapshots" at "http://repo.typesafe.com/typesafe/snapshots/"
```

Define the library dependencies with the timestamp as version. For example:

```
libraryDependencies += "com.typesafe.akka" % "akka-actor" % "2.0-20111215-000549"

libraryDependencies += "com.typesafe.akka" % "akka-remote" % "2.0-20111215-000549"
```

CHAPTER

TEN

ADDITIONAL INFORMATION

10.1 Here is a list of recipes for all things Akka

- Martin Krassers Akka Event Sourcing example

10.2 Companies and Open Source projects using Akka

10.2.1 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 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 to 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 at 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/

10.2.2 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 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

10.3 Third-party Integrations

10.3.1 The Play! Framework

Play 2.0 is based upon Akka. Uses all its eventing and threading using Akka actors and futures.

Read more here: http://www.playframework.org/2.0.

10.3.2 Scalatra

Scalatra has Akka integration.

Read more here: https://github.com/scalatra/scalatra/blob/develop/akka/src/main/scala/org/scalatra/akka/AkkaSupport.scala

10.3.3 Gatling

Gatling is an Open Source Stress Tool.

Read more here: http://gatling-tool.org/

10.4 Other Language Bindings

10.4.1 JRuby

Read more here: https://github.com/iconara/mikka.

10.4.2 Groovy/Groovy++

Read more here: https://gist.github.com/620439.

10.4.3 Clojure

Read more here: http://blog.darevay.com/2011/06/clojure-and-akka-a-match-made-in/.

CHAPTER

ELEVEN

LINKS

- *Migration Guides*
- Downloads
- Source Code
- *Scaladoc API*
- *Documentation for Other Versions*
- *Issue Tracking*
- *Commercial Support*