Akka Scala Documentation

Release 2.3.2

Typesafe Inc

April 09, 2014

CONTENTS

1 Introduction **1**
- 1.1 What is Akka? . . . 1
- 1.2 Why Akka? . . . 2
- 1.3 Getting Started . . . 3
- 1.4 The Obligatory Hello World . . . 6
- 1.5 Use-case and Deployment Scenarios . . . 7
- 1.6 Examples of use-cases for Akka . . . 7

2 General **9**
- 2.1 Terminology, Concepts . . . 9
- 2.2 Actor Systems . . . 11
- 2.3 What is an Actor? . . . 13
- 2.4 Supervision and Monitoring . . . 15
- 2.5 Actor References, Paths and Addresses . . . 18
- 2.6 Location Transparency . . . 24
- 2.7 Akka and the Java Memory Model . . . 25
- 2.8 Message Delivery Reliability . . . 27
- 2.9 Configuration . . . 32

3 Actors **69**
- 3.1 Actors . . . 69
- 3.2 Typed Actors . . . 87
- 3.3 Fault Tolerance . . . 91
- 3.4 Dispatchers . . . 103
- 3.5 Mailboxes . . . 106
- 3.6 Routing . . . 112
- 3.7 FSM . . . 126
- 3.8 Persistence . . . 135
- 3.9 Testing Actor Systems . . . 151
- 3.10 Actor DSL . . . 166

4 Futures and Agents **169**
- 4.1 Futures . . . 169
- 4.2 Agents . . . 175

5 Networking **179**
- 5.1 Cluster Specification . . . 179
- 5.2 Cluster Usage . . . 187
- 5.3 Remoting . . . 206
- 5.4 Serialization . . . 215
- 5.5 I/O . . . 220
- 5.6 Using TCP . . . 221
- 5.7 Using UDP . . . 230
- 5.8 ZeroMQ . . . 232

5.9 Camel . . . 236

6 Utilities **247**
6.1 Event Bus . . . 247
6.2 Logging . . . 252
6.3 Scheduler . . . 259
6.4 Duration . . . 262
6.5 Circuit Breaker . . . 263
6.6 Akka Extensions . . . 267
6.7 Microkernel . . . 269

7 HowTo: Common Patterns **271**
7.1 Throttling Messages . . . 271
7.2 Balancing Workload Across Nodes . . . 271
7.3 Work Pulling Pattern to throttle and distribute work, and prevent mailbox overflow . . . 271
7.4 Ordered Termination . . . 271
7.5 Akka AMQP Proxies . . . 272
7.6 Shutdown Patterns in Akka 2 . . . 272
7.7 Distributed (in-memory) graph processing with Akka . . . 272
7.8 Case Study: An Auto-Updating Cache Using Actors . . . 272
7.9 Discovering message flows in actor systems with the Spider Pattern . . . 273
7.10 Scheduling Periodic Messages . . . 273
7.11 Template Pattern . . . 274

8 Experimental Modules **275**
8.1 Persistence . . . 275
8.2 Multi Node Testing . . . 292
8.3 Actors (Java with Lambda Support) . . . 297
8.4 FSM (Java with Lambda Support) . . . 316
8.5 External Contributions . . . 325

9 Information for Akka Developers **352**
9.1 Building Akka . . . 352
9.2 Multi JVM Testing . . . 354
9.3 I/O Layer Design . . . 357
9.4 Developer Guidelines . . . 359
9.5 Documentation Guidelines . . . 360
9.6 Team . . . 363

10 Project Information **364**
10.1 Migration Guides . . . 364
10.2 Issue Tracking . . . 372
10.3 Licenses . . . 372
10.4 Sponsors . . . 373
10.5 Project . . . 373

11 Additional Information **375**
11.1 Frequently Asked Questions . . . 375
11.2 Books . . . 378
11.3 Other Language Bindings . . . 378
11.4 Akka in OSGi . . . 378
11.5 Incomplete List of HTTP Frameworks . . . 379

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 scalable, resilient and responsive applications—see the Reactive Manifesto for more details. For fault-tolerance we adopt the "let it crash" model which the telecom industry has used with great success to build applications that self-heal and systems that never stop. Actors also provide 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.

Please note that all code samples compile, so if you want direct access to the sources, have a look over at the Akka Docs subproject on github: for Java and Scala.

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 (several million actors per GB of heap memory).

See the chapter for *Scala* or *Java*.

Fault Tolerance

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

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 pure message passing and everything is asynchronous.

For an overview of the cluster support see the *Java* and *Scala* documentation chapters.

Persistence

Messages received by an actor can optionally be persisted and replayed when the actor is started or restarted. This allows actors to recover their state, even after JVM crashes or when being migrated to another node.

You can find more details in the respective chapter for *Java* or *Scala*.

1.1.2 Scala and Java APIs

Akka has both a *Scala Documentation* 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 Commercial Support

Akka is available from Typesafe Inc. under a commercial license which includes development or production support, 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 context of performance but also 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:

- Investment and Merchant Banking
- Retail
- Social Media
- Simulation
- Gaming and Betting
- Automobile and Traffic Systems
- Health Care
- Data Analytics

and much more. Any system with the need for high-throughput and low latency is a good candidate for using Akka.

Actors let you manage service failures (Supervisors), load management (back-off strategies, timeouts and processing-isolation), as well as 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 Typesafe Activator and try out one of Akka Template Projects.

1.3.3 Download

There are several ways to download Akka. You can download it as part of the Typesafe Platform (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 consists of several JARs containing different features.

- `akka-actor` – Classic Actors, Typed Actors, IO Actor etc.
- `akka-agent` – Agents, integrated with Scala STM
- `akka-camel` – Apache Camel integration
- `akka-cluster` – Cluster membership management, elastic routers.
- `akka-kernel` – Akka microkernel for running a bare-bones mini application server

- `akka-osgi` – base bundle for using Akka in OSGi containers, containing the `akka-actor` classes
- `akka-osgi-aries` – Aries blueprint for provisioning actor systems
- `akka-remote` – Remote Actors
- `akka-slf4j` – SLF4J Logger (event bus listener)
- `akka-testkit` – Toolkit for testing Actor systems
- `akka-zeromq` – ZeroMQ integration

In addition to these stable modules there are several which are on their way into the stable core but are still marked "experimental" at this point. This does not mean that they do not function as intended, it primarily means that their API has not yet solidified enough in order to be considered frozen. You can help accelerating this process by giving feedback on these modules on our mailing list.

- `akka-contrib` – an assortment of contributions which may or may not be moved into core modules, see *External Contributions* for more details.

The filename of the actual JAR is for example `akka-actor_2.10-2.3.2.jar` (and analog for the other modules).

How to see the JARs dependencies of each Akka module is described in the *Dependencies* section.

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://repo.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.

Warning: The use of Akka SNAPSHOTs, nightlies and milestone releases is discouraged unless you know what you are doing.

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 (Scala)* / *Microkernel (Java)*.

1.3.8 Using a build tool

Akka can be used with build tools that support Maven repositories.

1.3.9 Maven repositories

For Akka version 2.1-M2 and onwards:

Maven Central

For previous Akka versions:

Akka Repo Typesafe Repo

1.3.10 Using Akka with Maven

The simplest way to get started with Akka and Maven is to check out the Typesafe Activator tutorial named Akka Main in Java.

Since Akka is published to Maven Central (for versions since 2.1-M2), is it enough to add the Akka dependencies to the POM. For example, here is the dependency for akka-actor:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor_2.10</artifactId>
  <version>2.3.2</version>
</dependency>
```

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

1.3.11 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 := "2.10.3"

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

libraryDependencies +=
  "com.typesafe.akka" %% "akka-actor" % "2.3.2"
```

Note: the libraryDependencies setting above is specific to SBT v0.12.x and higher. If you are using an older version of SBT, the libraryDependencies should look like this:

```
libraryDependencies +=
  "com.typesafe.akka" % "akka-actor_2.10" % "2.3.2"
```

1.3.12 Using Akka with Gradle

Requires at least Gradle 1.4 Uses the Scala plugin

```
apply plugin: 'scala'

repositories {
  mavenCentral()
}

dependencies {
  compile 'org.scala-lang:scala-library:2.10.3'
}

tasks.withType(ScalaCompile) {
  scalaCompileOptions.useAnt = false
}

dependencies {
```

```
  compile group: 'com.typesafe.akka', name: 'akka-actor_2.10', version: '2.3.2'
  compile group: 'org.scala-lang', name: 'scala-library', version: '2.10.3'
}
```

1.3.13 Using Akka with Eclipse

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

1.3.14 Using Akka with IntelliJ IDEA

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

1.3.15 Using Akka with NetBeans

Setup SBT project and then use nbsbt to generate a NetBeans project.

You should also use nbscala for general scala support in the IDE.

1.3.16 Do not use -optimize Scala compiler flag

Warning: Akka has not been compiled or tested with -optimize Scala compiler flag. Strange behavior has been reported by users that have tried it.

1.3.17 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*

1.3.18 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 The Obligatory Hello World

The actor based version of the tough problem of printing a well-known greeting to the console is introduced in a Typesafe Activator tutorial named Akka Main in Scala.

The tutorial illustrates the generic launcher class `akka.Main` which expects only one command line argument: the class name of the application's main actor. This main method will then create the infrastructure needed for running the actors, start the given main actor and arrange for the whole application to shut down once the main actor terminates.

There is also another Typesafe Activator tutorial in the same problem domain that is named Hello Akka!. It describes the basics of Akka in more depth.

1.5 Use-case and Deployment Scenarios

1.5.1 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 (Scala)* / *Microkernel (Java)*

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. See *Microkernel (Scala)* / *Microkernel (Java)* for more information.

1.6 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

1.6.1 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 Terminology, Concepts

In this chapter we attempt to establish a common terminology to define a solid ground for communicating about concurrent, distributed systems which Akka targets. Please note that, for many of these terms, there is no single agreed definition. We simply seek to give working definitions that will be used in the scope of the Akka documentation.

2.1.1 Concurrency vs. Parallelism

Concurrency and parallelism are related concepts, but there are small differences. *Concurrency* means that two or more tasks are making progress even though they might not be executing simultaneously. This can for example be realized with time slicing where parts of tasks are executed sequentially and mixed with parts of other tasks. *Parallelism* on the other hand arise when the execution can be truly simultaneous.

2.1.2 Asynchronous vs. Synchronous

A method call is considered *synchronous* if the caller cannot make progress until the method returns a value or throws an exception. On the other hand, an *asynchronous* call allows the caller to progress after a finite number of steps, and the completion of the method may be signalled via some additional mechanism (it might be a registered callback, a Future, or a message).

A synchronous API may use blocking to implement synchrony, but this is not a necessity. A very CPU intensive task might give a similar behavior as blocking. In general, it is preferred to use asynchronous APIs, as they guarantee that the system is able to progress. Actors are asynchronous by nature: an actor can progress after a message send without waiting for the actual delivery to happen.

2.1.3 Non-blocking vs. Blocking

We talk about *blocking* if the delay of one thread can indefinitely delay some of the other threads. A good example is a resource which can be used exclusively by one thread using mutual exclusion. If a thread holds on to the resource indefinitely (for example accidentally running an infinite loop) other threads waiting on the resource can not progress. In contrast, *non-blocking* means that no thread is able to indefinitely delay others.

Non-blocking operations are preferred to blocking ones, as the overall progress of the system is not trivially guaranteed when it contains blocking operations.

2.1.4 Deadlock vs. Starvation vs. Live-lock

Deadlock arises when several participants are waiting on each other to reach a specific state to be able to progress. As none of them can progress without some other participant to reach a certain state (a “Catch-22” problem) all

affected subsystems stall. Deadlock is closely related to *blocking*, as it is necessary that a participant thread be able to delay the progression of other threads indefinitely.

In the case of *deadlock*, no participants can make progress, while in contrast *Starvation* happens, when there are participants that can make progress, but there might be one or more that cannot. Typical scenario is the case of a naive scheduling algorithm that always selects high-priority tasks over low-priority ones. If the number of incoming high-priority tasks is constantly high enough, no low-priority ones will be ever finished.

Livelock is similar to *deadlock* as none of the participants make progress. The difference though is that instead of being frozen in a state of waiting for others to progress, the participants continuously change their state. An example scenario when two participants have two identical resources available. They each try to get the resource, but they also check if the other needs the resource, too. If the resource is requested by the other participant, they try to get the other instance of the resource. In the unfortunate case it might happen that the two participants "bounce" between the two resources, never acquiring it, but always yielding to the other.

2.1.5 Race Condition

We call it a *Race condition* when an assumption about the ordering of a set of events might be violated by external non-deterministic effects. Race conditions often arise when multiple threads have a shared mutable state, and the operations of thread on the state might be interleaved causing unexpected behavior. While this is a common case, shared state is not necessary to have race conditions. One example could be a client sending unordered packets (e.g UDP datagrams) `P1`, `P2` to a server. As the packets might potentially travel via different network routes, it is possible that the server receives `P2` first and `P1` afterwards. If the messages contain no information about their sending order it is impossible to determine by the server that they were sent in a different order. Depending on the meaning of the packets this can cause race conditions.

Note: The only guarantee that Akka provides about messages sent between a given pair of actors is that their order is always preserved. see *Message Delivery Reliability*

2.1.6 Non-blocking Guarantees (Progress Conditions)

As discussed in the previous sections blocking is undesirable for several reasons, including the dangers of deadlocks and reduced throughput in the system. In the following sections we discuss various non-blocking properties with different strength.

Wait-freedom

A method is *wait-free* if every call is guaranteed to finish in a finite number of steps. If a method is *bounded wait-free* then the number of steps has a finite upper bound.

From this definition it follows that wait-free methods are never blocking, therefore deadlock can not happen. Additionally, as each participant can progress after a finite number of steps (when the call finishes), wait-free methods are free of starvation.

Lock-freedom

Lock-freedom is a weaker property than *wait-freedom*. In the case of lock-free calls, infinitely often some method finishes in a finite number of steps. This definition implies that no deadlock is possible for lock-free calls. On the other hand, the guarantee that *some call finishes* in a finite number of steps is not enough to guarantee that *all of them eventually finish*. In other words, lock-freedom is not enough to guarantee the lack of starvation.

Obstruction-freedom

Obstruction-freedom is the weakest non-blocking guarantee discussed here. A method is called *obstruction-free* if there is a point in time after which it executes in isolation (other threads make no steps, e.g.: become suspended),

it finishes in a bounded number of steps. All lock-free objects are obstruction-free, but the opposite is generally not true.

Optimistic concurrency control (OCC) methods are usually obstruction-free. The OCC approach is that every participant tries to execute its operation on the shared object, but if a participant detects conflicts from others, it rolls back the modifications, and tries again according to some schedule. If there is a point in time, where one of the participants is the only one trying, the operation will succeed.

2.1.7 Recommended literature

- The Art of Multiprocessor Programming, M. Herlihy and N Shavit, 2008. ISBN 978-0123705914
- Java Concurrency in Practice, B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes and D. Lea, 2006. ISBN 978-0321349606

2.2 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.

Note: An ActorSystem is a heavyweight structure that will allocate 1…N Threads, so create one per logical application.

2.2.1 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 normally 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 guidelines 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.

2.2.2 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.

2.2.3 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.—unless it is unavoidable; in the latter case see below.
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 with respect to fault-handling (both considering the granularity of configuration and the performance) and it also reduces the strain on the guardian actor, which is a single point of contention if over-used.

2.2.4 Blocking Needs Careful Management

In some cases it is unavoidable to do blocking operations, i.e. to put a thread to sleep for an indeterminate time, waiting for an external event to occur. Examples are legacy RDBMS drivers or messaging APIs, and the underlying reason is typically that (network) I/O occurs under the covers. When facing this, you may be tempted to just wrap the blocking call inside a `Future` and work with that instead, but this strategy is too simple: you are quite likely to find bottlenecks or run out of memory or threads when the application runs under increased load.

The non-exhaustive list of adequate solutions to the "blocking problem" includes the following suggestions:

- Do the blocking call within an actor (or a set of actors managed by a router [*Java*, *Scala*]), making sure to configure a thread pool which is either dedicated for this purpose or sufficiently sized.
- Do the blocking call within a `Future`, ensuring an upper bound on the number of such calls at any point in time (submitting an unbounded number of tasks of this nature will exhaust your memory or thread limits).
- Do the blocking call within a `Future`, providing a thread pool with an upper limit on the number of threads which is appropriate for the hardware on which the application runs.

- Dedicate a single thread to manage a set of blocking resources (e.g. a NIO selector driving multiple channels) and dispatch events as they occur as actor messages.

The first possibility is especially well-suited for resources which are single-threaded in nature, like database handles which traditionally can only execute one outstanding query at a time and use internal synchronization to ensure this. A common pattern is to create a router for N actors, each of which wraps a single DB connection and handles queries as sent to the router. The number N must then be tuned for maximum throughput, which will vary depending on which DBMS is deployed on what hardware.

Note: Configuring thread pools is a task best delegated to Akka, simply configure in the `application.conf` and instantiate through an `ActorSystem` [*Java*, *Scala*]

2.2.5 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.

2.3 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 a more in depth reference with all the details please refer to *Actors (Scala)* and *Untyped 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.

2.3.1 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.

2.3.2 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.

Optionally, an actor's state can be automatically recovered to the state before a restart by persisting received messages and replaying them after restart (see *Persistence*).

2.3.3 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.

2.3.4 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.

2.3.5 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.

2.3.6 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.

2.3.7 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" which will forward them to the EventStream as DeadLetters. The mailbox is then replaced within the actor reference with a system mailbox, redirecting all new messages to the EventStream as DeadLetters. 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.

2.4 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.

2.4.1 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. Stop the subordinate permanently
4. Escalate the failure, thereby failing itself

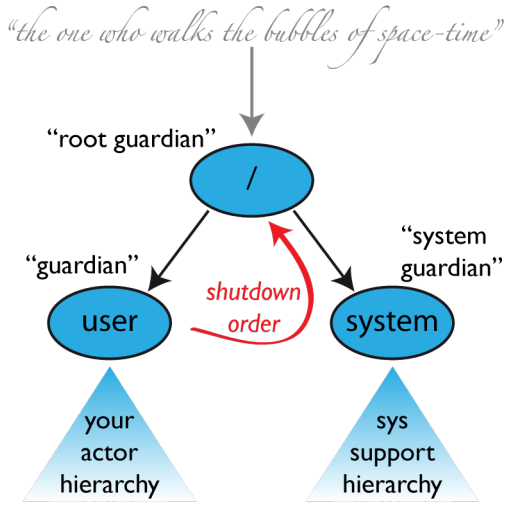
It is important to always view an actor as part of a supervision hierarchy, which explains the existence of the fourth choice (as a supervisor also is subordinate to another supervisor higher up) and has implications on the first three: resuming an actor resumes all its subordinates, restarting an actor 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 implicit 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.

Warning: Supervision related parent-child communication happens by special system messages that have their own mailboxes separate from user messages. This implies that supervision related events are not deterministically ordered relative to ordinary messages. In general, the user cannot influence the order of normal messages and failure notifications. For details and example see the *Discussion: Message Ordering* section.

2.4.2 The Top-Level Supervisors



An actor system will during its creation start at least three actors, shown in the image above. For more information about the consequences for actor paths see *Top-Level Scopes for Actor Paths*.

`/user`: The Guardian Actor

The actor which is probably most interacted with is the parent of all user-created actors, the guardian named `"/user"`. Actors created using `system.actorOf()` are children of this actor. This means that when this guardian terminates, all normal actors in the system will be shutdown, too. It also means that this guardian's supervisor strategy determines how the top-level normal actors are supervised. Since Akka 2.1 it is possible to configure this using the setting `akka.actor.guardian-supervisor-strategy`, which takes the fully-qualified class-name of a `SupervisorStrategyConfigurator`. When the guardian escalates a failure, the root guardian's response will be to terminate the guardian, which in effect will shut down the whole actor system.

`/system`: The System Guardian

This special guardian has been introduced in order to achieve an orderly shut-down sequence where logging remains active while all normal actors terminate, even though logging itself is implemented using actors. This is realized by having the system guardian watch the user guardian and initiate its own shut-down upon reception of the `Terminated` message. The top-level system actors are supervised using a strategy which will restart indefinitely upon all types of `Exception` except for `ActorInitializationException` and `ActorKilledException`, which will terminate the child in question. All other throwables are escalated, which will shut down the whole actor system.

`/`: The Root Guardian

The root guardian is the grand-parent of all so-called "top-level" actors and supervises all the special actors mentioned in *Top-Level Scopes for Actor Paths* using the `SupervisorStrategy.stoppingStrategy`, whose purpose is to terminate the child upon any type of `Exception`. All other throwables will be escalated ... but to whom? Since every real actor has a supervisor, the supervisor of the root guardian cannot be a real

actor. And because this means that it is "outside of the bubble", it is called the "bubble-walker". This is a synthetic `ActorRef` which in effect stops its child upon the first sign of trouble and sets the actor system's `isTerminated` status to `true` as soon as the root guardian is fully terminated (all children recursively stopped).

2.4.3 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:

1. suspend the actor (which means that it will not process normal messages until resumed), and recursively suspend all children
2. call the old instance's `preRestart` hook (defaults to sending termination requests to all children and calling `postStop`)
3. wait for all children which were requested to terminate (using `context.stop()`) during `preRestart` to actually terminate; this—like all actor operations—is non-blocking, the termination notice from the last killed child will effect the progression to the next step
4. create new actor instance by invoking the originally provided factory again
5. invoke `postRestart` on the new instance (which by default also calls `preStart`)
6. send restart request to all children which were not killed in step 3; restarted children will follow the same process recursively, from step 2
7. resume the actor

2.4.4 What Lifecycle Monitoring Means

Note: Lifecycle Monitoring in Akka is usually referred to as `DeathWatch`

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. In order to start listening for `Terminated` messages, invoke `ActorContext.watch(targetActorRef)`. To stop listening, invoke `ActorContext.unwatch(targetActorRef)`. 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.

2.4.5 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.

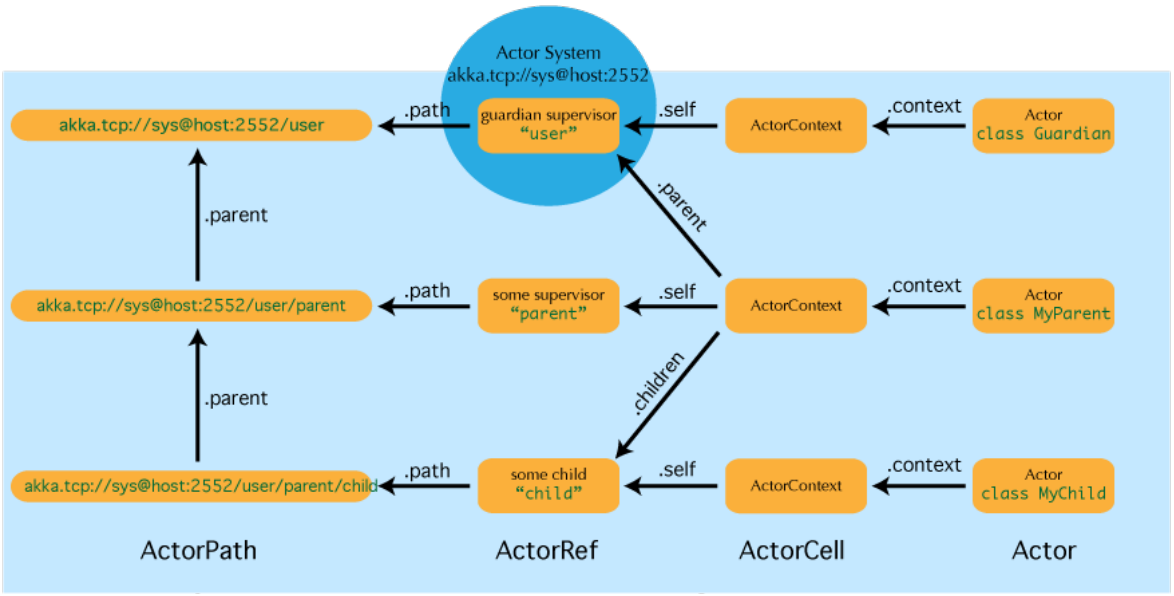
The `AllForOneStrategy` is applicable in cases where the ensemble of children has such tight dependencies among them, that a failure of one child affects the function of the others, i.e. they are inextricably linked. Since a restart does not clear out the mailbox, it often is best to terminate the children 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; this can easily be done by watching their lifecycle: if the `Terminated` message is not handled by the supervisor, it will throw a `DeathPactException` 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* or *routing-java*.

2.5 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.

2.5.1 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` method.

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 will not function if sent across a network connection to a remote JVM.
- 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 also be reachable 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 remote 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. `akka.pattern.ask` creates this actor reference.
  - `DeadLetterActorRef` is the default implementation of the dead letters service to which Akka routes all messages whose destinations are shut down or non-existent.
  - `EmptyLocalActorRef` is what Akka returns when looking up a non-existent local actor path: it is equivalent to a `DeadLetterActorRef`, but it retains its path so that Akka can send it over the network and compare it to other existing actor references for that path, some of which might have been obtained before the actor died.
- 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`.

2.5.2 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.

What is the Difference Between Actor Reference and Path?

An actor reference designates a single actor and the life-cycle of the reference matches that actor's life-cycle; an actor path represents a name which may or may not be inhabited by an actor and the path itself does not have a life-cycle, it never becomes invalid. You can create an actor path without creating an actor, but you cannot create an actor reference without creating corresponding actor.

Note: That definition does not hold for `actorFor`, which is one of the reasons why `actorFor` is deprecated in favor of `actorSelection`.

You can create an actor, terminate it, and then create a new actor with the same actor path. The newly created actor is a new incarnation of the actor. It is not the same actor. An actor reference to the old incarnation is not valid for the new incarnation. Messages sent to the old actor reference will not be delivered to the new incarnation even though they have the same path.

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-sys/user/service-a/worker1"                   // purely local
"akka.tcp://my-sys@host.example.com:5678/user/service-b" // remote
```

Here, `akka.tcp` is the default remote transport for the 2.2 release; other transports are pluggable. A remote host using UDP would be accessible by using `akka.udp`. The interpretation of the host and 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 than 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.

2.5.3 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.

Creating Actors

An actor system is typically started by creating actors beneath 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 (through its `ActorContext`) 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.actorSelection` method. The selection can be used for communicating with said actor and the actor corresponding to the selection is looked up when delivering each message.

To acquire an `ActorRef` that is bound to the life-cycle of a specific actor you need to send a message, such as the built-in `Identify` message, to the actor and use the `sender()` reference of a reply from the actor.

Note: `actorFor` is deprecated in favor of `actorSelection` because actor references acquired with `actorFor` behave differently for local and remote actors. In the case of a local actor reference, the named actor needs to exist before the lookup, or else the acquired reference will be an `EmptyLocalActorRef`. This will be true even if an actor with that exact path is created after acquiring the actor reference. For remote actor references acquired with *actorFor* the behaviour is different and sending messages to such a reference will under the hood look up the actor by path on the remote system for every message send.

Absolute vs. Relative Paths

In addition to `ActorSystem.actorSelection` there is also `ActorContext.actorSelection`, which is available inside any actor as `context.actorSelection`. This yields an actor selection 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.actorSelection("../brother") ! msg
```

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

```
context.actorSelection("/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 way 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. `actorSelection` 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).
- `actorSelection` only ever looks up existing actors when messages are delivered, i.e. does not create actors, or verify existence of actors when the selection is created.
- `actorFor` (deprecated in favor of actorSelection) only ever looks up an existing actor, i.e. does not create one.

2.5.4 Actor Reference and Path Equality

Equality of `ActorRef` match the intention that an `ActorRef` corresponds to the target actor incarnation. Two actor references are compared equal when they have the same path and point to the same actor incarnation. A reference pointing to a terminated actor does not compare equal to a reference pointing to another (re-created) actor with the same path. Note that a restart of an actor caused by a failure still means that it is the same actor incarnation, i.e. a restart is not visible for the consumer of the `ActorRef`.

Remote actor references acquired with `actorFor` do not include the full information about the underlying actor identity and therefore such references do not compare equal to references acquired with `actorOf`, `sender`, or `context.self`. Because of this `actorFor` is deprecated in favor of `actorSelection`.

If you need to keep track of actor references in a collection and do not care about the exact actor incarnation you can use the `ActorPath` as key, because the identifier of the target actor is not taken into account when comparing actor paths.

2.5.5 Reusing Actor Paths

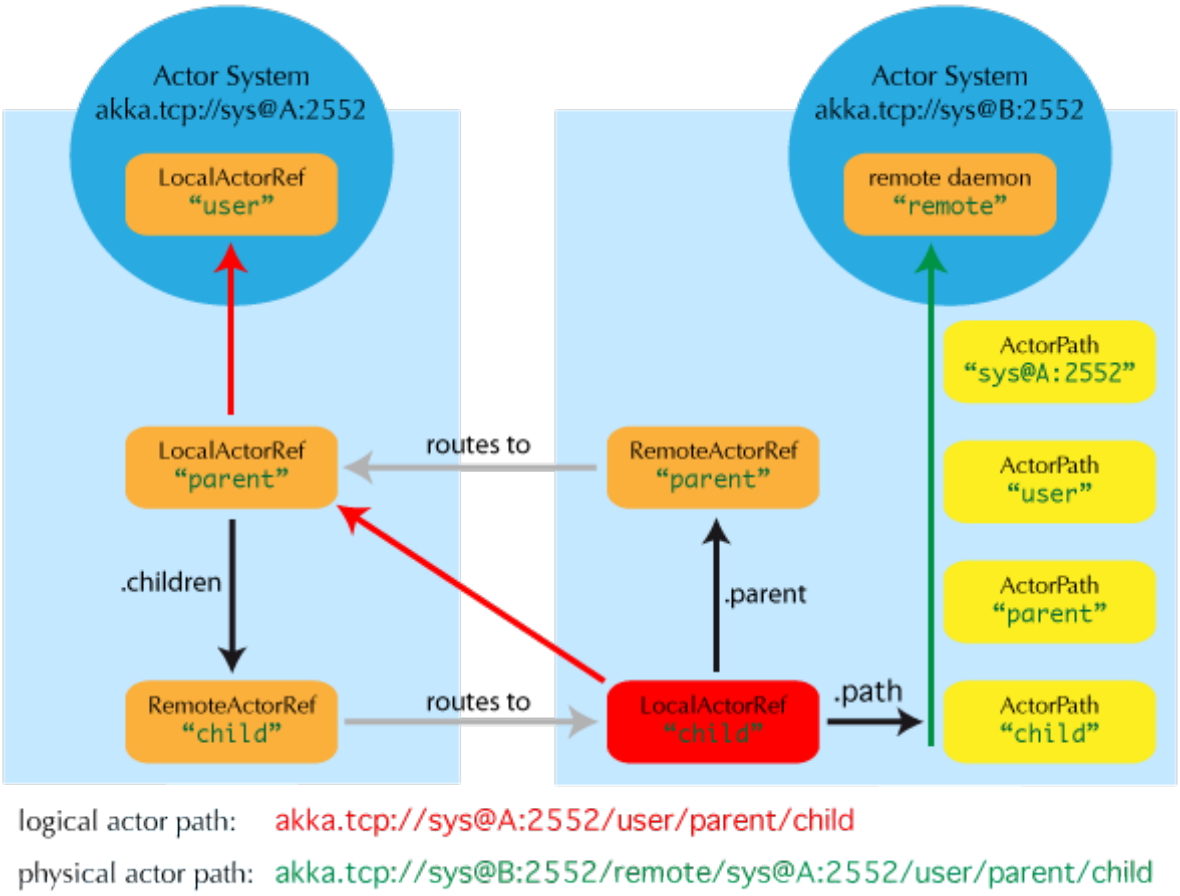
When an actor is terminated, its reference will point to the dead letter mailbox, DeathWatch will publish its final transition and in general it is not expected to come back to life again (since the actor life cycle does not allow this). While it is possible to create an actor at a later time with an 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 acquired with `actorFor` 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.

2.5.6 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.



2.5.7 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.

2.5.8 Top-Level Scopes for Actor Paths

At the root of the path hierarchy resides the root guardian above which all other actors are found; its name is `"/"`. 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 below this one.
- `"/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 (on a best-effort basis: messages may be lost even within the local JVM).
- `"/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

The need to structure the name space for actors like this arises from a central and very simple design goal: everything in the hierarchy is an actor, and all actors function in the same way. Hence you can not only look up the actors you created, you can also look up the system guardian and send it a message (which it will dutifully

discard in this case). This powerful principle means that there are no quirks to remember, it makes the whole system more uniform and consistent.

If you want to read more about the top-level structure of an actor system, have a look at *The Top-Level Supervisors*.

2.6 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.

2.6.1 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.

2.6.2 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!).

2.6.3 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).

2.6.4 Peer-to-Peer vs. Client-Server

Akka Remoting is a communication module for connecting actor systems in a peer-to-peer fashion, and it is the foundation for Akka Clustering. The design of remoting is driven by two (related) design decisions:

1. Communication between involved systems is symmetric: if a system A can connect to a system B then system B must also be able to connect to system A independently.
2. The role of the communicating systems are symmetric in regards to connection patterns: there is no system that only accepts connections, and there is no system that only initiates connections.

The consequence of these decisions is that it is not possible to safely create pure client-server setups with predefined roles (violates assumption 2) and using setups involving Network Address Translation or Load Balancers (violates assumption 1).

For client-server setups it is better to use HTTP or Akka I/O.

2.6.5 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)*.

2.7 Akka and the Java Memory Model

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

2.7.1 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.

2.7.2 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.

Note: In layman's terms this means that changes to internal fields of the actor are visible when the next message is processed by that actor. So fields in your actor need not be volatile or equivalent.

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

2.7.3 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.

2.7.4 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.

2.7.5 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.

2.8 Message Delivery Reliability

Akka helps you build reliable applications which make use of multiple processor cores in one machine ("scaling up") or distributed across a computer network ("scaling out"). The key abstraction to make this work is that all interactions between your code units—actors—happen via message passing, which is why the precise semantics of how messages are passed between actors deserve their own chapter.

In order to give some context to the discussion below, consider an application which spans multiple network hosts. The basic mechanism for communication is the same whether sending to an actor on the local JVM or to a remote actor, but of course there will be observable differences in the latency of delivery (possibly also depending on the bandwidth of the network link and the message size) and the reliability. In case of a remote message send there are obviously more steps involved which means that more can go wrong. Another aspect is that local sending will just pass a reference to the message inside the same JVM, without any restrictions on the underlying object which is sent, whereas a remote transport will place a limit on the message size.

Writing your actors such that every interaction could possibly be remote is the safe, pessimistic bet. It means to only rely on those properties which are always guaranteed and which are discussed in detail below. This has of course some overhead in the actor's implementation. If you are willing to sacrifice full location transparency—for example in case of a group of closely collaborating actors—you can place them always on the same JVM and enjoy stricter guarantees on message delivery. The details of this trade-off are discussed further below.

As a supplementary part we give a few pointers at how to build stronger reliability on top of the built-in ones. The chapter closes by discussing the role of the "Dead Letter Office".

2.8.1 The General Rules

These are the rules for message sends (i.e. the `tell` or `!` method, which also underlies the `ask` pattern):

- **at-most-once delivery**, i.e. no guaranteed delivery
- **message ordering per sender–receiver pair**

The first rule is typically found also in other actor implementations while the second is specific to Akka.

Discussion: What does "at-most-once" mean?

When it comes to describing the semantics of a delivery mechanism, there are three basic categories:

- **at-most-once** delivery means that for each message handed to the mechanism, that message is delivered zero or one times; in more casual terms it means that messages may be lost.

- **at-least-once** delivery means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that messages may be duplicated but not lost.
- **exactly-once** delivery means that for each message handed to the mechanism exactly one delivery is made to the recipient; the message can neither be lost nor duplicated.

The first one is the cheapest—highest performance, least implementation overhead—because it can be done in a fire-and-forget fashion without keeping state at the sending end or in the transport mechanism. The second one requires retries to counter transport losses, which means keeping state at the sending end and having an acknowledgement mechanism at the receiving end. The third is most expensive—and has consequently worst performance—because in addition to the second it requires state to be kept at the receiving end in order to filter out duplicate deliveries.

Discussion: Why No Guaranteed Delivery?

At the core of the problem lies the question what exactly this guarantee shall mean:

1. The message is sent out on the network?
2. The message is received by the other host?
3. The message is put into the target actor's mailbox?
4. The message is starting to be processed by the target actor?
5. The message is processed successfully by the target actor?

Each one of these have different challenges and costs, and it is obvious that there are conditions under which any message passing library would be unable to comply; think for example about configurable mailbox types and how a bounded mailbox would interact with the third point, or even what it would mean to decide upon the "successfully" part of point five.

Along those same lines goes the reasoning in Nobody Needs Reliable Messaging. The only meaningful way for a sender to know whether an interaction was successful is by receiving a business-level acknowledgement message, which is not something Akka could make up on its own (neither are we writing a "do what I mean" framework nor would you want us to).

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

Another angle on this issue is that by providing only basic guarantees those use cases which do not need stronger reliability do not pay the cost of their implementation; it is always possible to add stronger reliability on top of basic ones, but it is not possible to retro-actively remove reliability in order to gain more performance.

Discussion: Message Ordering

The rule more specifically is that *for a given pair of actors, messages sent from the first to the second will not be received out-of-order.* This is illustrated in the following:

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, any of the messages may be dropped, i.e. not arrive at `A2`

Note: It is important to note that Akka's guarantee applies to the order in which messages are enqueued into the recipient's mailbox. If the mailbox implementation does not respect FIFO order (e.g. a `PriorityMailbox`), then the order of processing by the actor can deviate from the enqueueing order.

Please note that this rule is **not transitive**:

Actor `A` sends message `M1` to actor `C`

Actor `A` then sends message `M2` to actor `B`

Actor `B` forwards message `M2` to actor `C`

Actor `C` may receive `M1` and `M2` in any order

Causal transitive ordering would imply that `M2` is never received before `M1` at actor `C` (though any of them might be lost). This ordering can be violated due to different message delivery latencies when `A`, `B` and `C` reside on different network hosts, see more below.

Note: Actor creation is treated as a message sent from the parent to the child, with the same semantics as discussed above. Sending a message to an actor in a way which could be reordered with this initial creation message means that the message might not arrive because the actor does not exist yet. An example where the message might arrive too early would be to create a remote-deployed actor R1, send its reference to another remote actor R2 and have R2 send a message to R1. An example of well-defined ordering is a parent which creates an actor and immediately sends a message to it.

Communication of failure

Please note, that the ordering guarantees discussed above only hold for user messages between actors. Failure of a child of an actor is communicated by special system messages that are not ordered relative to ordinary user messages. In particular:

Child actor `C` sends message `M` to its parent `P`

Child actor fails with failure `F`

Parent actor `P` might receive the two events either in order `M`, `F` or `F`, `M`

The reason for this is that internal system messages has their own mailboxes therefore the ordering of enqueue calls of a user and system message cannot guarantee the ordering of their dequeue times.

2.8.2 The Rules for In-JVM (Local) Message Sends

Be careful what you do with this section!

Relying on the stronger reliability in this section is not recommended since it will bind your application to local-only deployment: an application may have to be designed differently (as opposed to just employing some message exchange patterns local to some actors) in order to be fit for running on a cluster of machines. Our credo is "design once, deploy any way you wish", and to achieve this you should only rely on The General Rules.

Reliability of Local Message Sends

The Akka test suite relies on not losing messages in the local context (and for non-error condition tests also for remote deployment), meaning that we actually do apply the best effort to keep our tests stable. A local `tell` operation can however fail for the same reasons as a normal method call can on the JVM:

- `StackOverflowError`
- `OutOfMemoryError`
- other `VirtualMachineError`

In addition, local sends can fail in Akka-specific ways:

- if the mailbox does not accept the message (e.g. full BoundedMailbox)
- if the receiving actor fails while processing the message or is already terminated

While the first is clearly a matter of configuration the second deserves some thought: the sender of a message does not get feedback if there was an exception while processing, that notification goes to the supervisor instead. This is in general not distinguishable from a lost message for an outside observer.

Ordering of Local Message Sends

Assuming strict FIFO mailboxes the abovementioned caveat of non-transitivity of the message ordering guarantee is eliminated under certain conditions. As you will note, these are quite subtle as it stands, and it is even possible that future performance optimizations will invalidate this whole paragraph. The possibly non-exhaustive list of counter-indications is:

- Before receiving the first reply from a top-level actor, there is a lock which protects an internal interim queue, and this lock is not fair; the implication is that enqueue requests from different senders which arrive during the actor's construction (figuratively, the details are more involved) may be reordered depending on low-level thread scheduling. Since completely fair locks do not exist on the JVM this is unfixable.
- The same mechanism is used during the construction of a Router, more precisely the routed ActorRef, hence the same problem exists for actors deployed with Routers.
- As mentioned above, the problem occurs anywhere a lock is involved during enqueueing, which may also apply to custom mailboxes.

This list has been compiled carefully, but other problematic scenarios may have escaped our analysis.

How does Local Ordering relate to Network Ordering

As explained in the previous paragraph local message sends obey transitive causal ordering under certain conditions. If the remote message transport would respect this ordering as well, that would translate to transitive causal ordering across one network link, i.e. if exactly two network hosts are involved. Involving multiple links, e.g. the three actors on three different nodes mentioned above, then no guarantees can be made.

The current remote transport does **not** support this (again this is caused by non-FIFO wake-up order of a lock, this time serializing connection establishment).

As a speculative view into the future it might be possible to support this ordering guarantee by re-implementing the remote transport layer based completely on actors; at the same time we are looking into providing other low-level transport protocols like UDP or SCTP which would enable higher throughput or lower latency by removing this guarantee again, which would mean that choosing between different implementations would allow trading guarantees versus performance.

2.8.3 Higher-level abstractions

Based on a small and consistent tool set in Akka's core, Akka also provides powerful, higher-level abstractions on top it.

Messaging Patterns

As discussed above a straight-forward answer to the requirement of reliable delivery is an explicit ACK–RETRY protocol. In its simplest form this requires

- a way to identify individual messages to correlate message with acknowledgement
- a retry mechanism which will resend messages if not acknowledged in time
- a way for the receiver to detect and discard duplicates

The third becomes necessary by virtue of the acknowledgements not being guaranteed to arrive either. An ACK-RETRY protocol with business-level acknowledgements is supported by *Channels* of the Akka Persistence module. Duplicates can be detected by tracking the sequence numbers of messages received via channels. Another way of implementing the third part would be to make processing the messages idempotent on the level of the business logic.

Another example of implementing all three requirements is shown at *Reliable Proxy Pattern* (which is now superseded by *Channels*).

Event Sourcing

Event sourcing (and sharding) is what makes large websites scale to billions of users, and the idea is quite simple: when a component (think actor) processes a command it will generate a list of events representing the effect of the command. These events are stored in addition to being applied to the component's state. The nice thing about this scheme is that events only ever are appended to the storage, nothing is ever mutated; this enables perfect replication and scaling of consumers of this event stream (i.e. other components may consume the event stream as a means to replicate the component's state on a different continent or to react to changes). If the component's state is lost—due to a machine failure or by being pushed out of a cache—it can easily be reconstructed by replaying the event stream (usually employing snapshots to speed up the process). *Event sourcing* is supported by Akka Persistence.

Mailbox with Explicit Acknowledgement

By implementing a custom mailbox type it is possible retry message processing at the receiving actor's end in order to handle temporary failures. This pattern is mostly useful in the local communication context where delivery guarantees are otherwise sufficient to fulfill the application's requirements.

Please note that the caveats for The Rules for In-JVM (Local) Message Sends do apply.

An example implementation of this pattern is shown at *Mailbox with Explicit Acknowledgement*.

2.8.4 Dead Letters

Messages which cannot be delivered (and for which this can be ascertained) will be delivered to a synthetic actor called `/deadLetters`. This delivery happens on a best-effort basis; it may fail even within the local JVM (e.g. during actor termination). Messages sent via unreliable network transports will be lost without turning up as dead letters.

What Should I Use Dead Letters For?

The main use of this facility is for debugging, especially if an actor send does not arrive consistently (where usually inspecting the dead letters will tell you that the sender or recipient was set wrong somewhere along the way). In order to be useful for this purpose it is good practice to avoid sending to deadLetters where possible, i.e. run your application with a suitable dead letter logger (see more below) from time to time and clean up the log output. This exercise—like all else—requires judicious application of common sense: it may well be that avoiding to send to a terminated actor complicates the sender's code more than is gained in debug output clarity.

The dead letter service follows the same rules with respect to delivery guarantees as all other message sends, hence it cannot be used to implement guaranteed delivery.

How do I Receive Dead Letters?

An actor can subscribe to class `akka.actor.DeadLetter` on the event stream, see *event-stream-java* (Java) or *Event Stream* (Scala) for how to do that. The subscribed actor will then receive all dead letters published in the (local) system from that point onwards. Dead letters are not propagated over the network, if you want to collect them in one place you will have to subscribe one actor per network node and forward them manually. Also consider that dead letters are generated at that node which can determine that a send operation is failed, which for a remote send can be the local system (if no network connection can be established) or the remote one (if the actor you are sending to does not exist at that point in time).

Dead Letters Which are (Usually) not Worrisome

Every time an actor does not terminate by its own decision, there is a chance that some messages which it sends to itself are lost. There is one which happens quite easily in complex shutdown scenarios that is usually benign: seeing a `akka.dispatch.Terminate` message dropped means that two termination requests were given, but of course only one can succeed. In the same vein, you might see `akka.actor.Terminated` messages from children while stopping a hierarchy of actors turning up in dead letters if the parent is still watching the child when the parent terminates.

2.9 Configuration

You can start using Akka without defining any configuration, since sensible default values are provided. Later on you might need to amend the settings to change the default behavior or adapt for specific runtime environments. Typical examples of settings that you might amend:

- log level and logger backend
- enable remoting
- message serializers
- definition of routers
- tuning of dispatchers

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.

2.9.1 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.

2.9.2 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.

If you are using Maven to package your application, you can also make use of the Apache Maven Shade Plugin support for Resource Transformers to merge all the reference.confs on the build classpath into one.

The plugin configuration might look like this:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-shade-plugin</artifactId>
 <version>1.5</version>
 <executions>
  <execution>
   <phase>package</phase>
   <goals>
    <goal>shade</goal>
   </goals>
   <configuration>
    <shadedArtifactAttached>true</shadedArtifactAttached>
    <shadedClassifierName>allinone</shadedClassifierName>
    <artifactSet>
     <includes>
      <include>*:*</include>
     </includes>
    </artifactSet>
    <transformers>
      <transformer
       implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
       <resource>reference.conf</resource>
      </transformer>
      <transformer
       implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
       <manifestEntries>
        <Main-Class>akka.Main</Main-Class>
       </manifestEntries>
      </transformer>
    </transformers>
   </configuration>
  </execution>
 </executions>
</plugin>
```

2.9.3 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 {

  # Loggers to register at boot time (akka.event.Logging$DefaultLogger logs
  # to STDOUT)
  loggers = ["akka.event.slf4j.Slf4jLogger"]

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

  # Log level for the very basic logger activated during ActorSystem startup.
  # This logger prints the log messages to stdout (System.out).
  # Options: OFF, ERROR, WARNING, INFO, DEBUG
  stdout-loglevel = "DEBUG"

  actor {
    provider = "akka.cluster.ClusterActorRefProvider"

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

  remote {
    # The port clients should connect to. Default is 2552.
    netty.tcp.port = 4711
  }
}
```

2.9.4 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.

2.9.5 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 2.10.3 (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();
System.out.println(system.settings());
// this is a shortcut for system.settings().config().root().render()
```

2.9.6 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.

2.9.7 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: *extending-akka-java.settings*

2.9.8 Configuring multiple ActorSystem

If you have more than one `ActorSystem` (or you're writing a library and have an `ActorSystem` that may be separate from the application's) you may want to separate the configuration for each system.

Given that `ConfigFactory.load()` merges all resources with matching name from the whole class path, it is easiest to utilize that functionality and differentiate 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(config))
```

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-pool
          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))
```

2.9.9 Reading configuration from a custom location

You can replace or supplement `application.conf` either in code or using system properties.

If you're using `ConfigFactory.load()` (which Akka does by default) you can replace `application.conf` by defining `-Dconfig.resource=whatever`, `-Dconfig.file=whatever`, or `-Dconfig.url=whatever`.

From inside your replacement file specified with `-Dconfig.resource` and friends, you can `include "application"` if you still want to use `application.{conf,json,properties}` as well. Settings specified before `include "application"` would be overridden by the included file, while those after would override the included file.

In code, there are many customization options.

There are several overloads of `ConfigFactory.load()`; these allow you to specify something to be sandwiched between system properties (which override) and the defaults (from `reference.conf`), replacing the usual `application.{conf,json,properties}` and replacing `-Dconfig.file` and friends.

The simplest variant of `ConfigFactory.load()` takes a resource basename (instead of `application`); `myname.conf`, `myname.json`, and `myname.properties` would then be used instead of `application.{conf,json,properties}`.

The most flexible variant takes a `Config` object, which you can load using any method in `ConfigFactory`. For example you could put a config string in code using `ConfigFactory.parseString()` or you could make a map and `ConfigFactory.parseMap()`, or you could load a file.

You can also combine your custom config with the usual config, that might look like:

```
// make a Config with just your special setting
Config myConfig =
  ConfigFactory.parseString("something=somethingElse");
// load the normal config stack (system props,
// then application.conf, then reference.conf)
Config regularConfig =
  ConfigFactory.load();
// override regular stack with myConfig
Config combined =
  myConfig.withFallback(regularConfig);
// put the result in between the overrides
// (system props) and defaults again
Config complete =
  ConfigFactory.load(combined);
// create ActorSystem
ActorSystem system =
  ActorSystem.create("myname", complete);
```

When working with `Config` objects, keep in mind that there are three "layers" in the cake:

- `ConfigFactory.defaultOverrides()` (system properties)
- the app's settings
- `ConfigFactory.defaultReference()` (reference.conf)

The normal goal is to customize the middle layer while leaving the other two alone.

- `ConfigFactory.load()` loads the whole stack
- the overloads of `ConfigFactory.load()` let you specify a different middle layer
- the `ConfigFactory.parse()` variations load single files or resources

To stack two layers, use `override.withFallback(fallback)`; try to keep system props (`defaultOverrides()`) on top and `reference.conf` (`defaultReference()`) on the bottom.

Do keep in mind, you can often just add another `include` statement in `application.conf` rather than writing code. Includes at the top of `application.conf` will be overridden by the rest of `application.conf`, while those at the bottom will override the earlier stuff.

2.9.10 Actor Deployment Configuration

Deployment settings for specific actors can be defined in the `akka.actor.deployment` section of the configuration. In the deployment section it is possible to define things like dispatcher, mailbox, router settings, and

remote deployment. Configuration of these features are described in the chapters detailing corresponding topics. An example may look like this:

```
akka.actor.deployment {

  # '/user/actorA/actorB' is a remote deployed actor
  /actorA/actorB {
    remote = "akka.tcp://sampleActorSystem@127.0.0.1:2553"
  }

  # all direct children of '/user/actorC' have a dedicated dispatcher
  "/actorC/*" {
    dispatcher = my-dispatcher
  }

  # '/user/actorD/actorE' has a special priority mailbox
  /actorD/actorE {
    mailbox = prio-mailbox
  }

  # '/user/actorF/actorG/actorH' is a random pool
  /actorF/actorG/actorH {
    router = random-pool
    nr-of-instances = 5
  }
}

my-dispatcher {
  fork-join-executor.parallelism-min = 10
  fork-join-executor.parallelism-max = 10
}
prio-mailbox {
  mailbox-type = "a.b.MyPrioMailbox"
}
```

The deployment section for a specific actor is identified by the path of the actor relative to `/user`.

You can use asterisks as wildcard matches for the actor path sections, so you could specify: `/*/sampleActor` and that would match all `sampleActor` on that level in the hierarchy. You can also use wildcard in the last position to match all actors at a certain level: `/someParent/*`. Non-wildcard matches always have higher priority to match than wildcards, so: `/foo/bar` is considered **more specific** than `/foo/*` and only the highest priority match is used. Please note that it **cannot** be used to partially match section, like this: `/foo*/bar`, `/f*o/bar` etc.

2.9.11 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 = "2.3.2"
```

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

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

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

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

# Log level for the very basic logger activated during ActorSystem startup.
# This logger prints the log messages to stdout (System.out).
# Options: OFF, 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

# Log at info level when messages are sent to dead letters.
# Possible values:
# on: all dead letters are logged
# off: no logging of dead letters
# n: positive integer, number of dead letters that will be logged
log-dead-letters = 10

# Possibility to turn off logging of dead letters while the actor system
# is shutting down. Logging is only done when enabled by 'log-dead-letters'
# setting.
log-dead-letters-during-shutdown = on

# 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 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 {

  # FQCN of the ActorRefProvider to be used; the below is the built-in default,
  # another one is akka.remote.RemoteActorRefProvider in the akka-remote bundle.
  provider = "akka.actor.LocalActorRefProvider"

  # The guardian "/user" will use this class to obtain its supervisorStrategy.
  # It needs to be a subclass of akka.actor.SupervisorStrategyConfigurator.
  # In addition to the default there is akka.actor.StoppingSupervisorStrategy.
  guardian-supervisor-strategy = "akka.actor.DefaultSupervisorStrategy"

  # 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 network, this is only intended for testing. Purely local deployments
# as marked with deploy.scope == LocalScope are exempt from verification.
serialize-creators = off

# Timeout for send operations to top-level actors which are in the process
# of being started. This is only relevant if using a bounded mailbox or the
# CallingThreadDispatcher for a top-level actor.
unstarted-push-timeout = 10s

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

# Mapping between ´deployment.router' short names to fully qualified class names
router.type-mapping {
  from-code = "akka.routing.NoRouter"
  round-robin-pool = "akka.routing.RoundRobinPool"
  round-robin-group = "akka.routing.RoundRobinGroup"
  random-pool = "akka.routing.RandomPool"
  random-group = "akka.routing.RandomGroup"
  balancing-pool = "akka.routing.BalancingPool"
  smallest-mailbox-pool = "akka.routing.SmallestMailboxPool"
  broadcast-pool = "akka.routing.BroadcastPool"
  broadcast-group = "akka.routing.BroadcastGroup"
  scatter-gather-pool = "akka.routing.ScatterGatherFirstCompletedPool"
  scatter-gather-group = "akka.routing.ScatterGatherFirstCompletedGroup"
  consistent-hashing-pool = "akka.routing.ConsistentHashingPool"
  consistent-hashing-group = "akka.routing.ConsistentHashingGroup"
}

deployment {

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

    # The id of the dispatcher to use for this actor.
    # If undefined or empty the dispatcher specified in code
    # (Props.withDispatcher) is used, or default-dispatcher if not
    # specified at all.
    dispatcher = ""

    # The id of the mailbox to use for this actor.
    # If undefined or empty the default mailbox of the configured dispatcher
    # is used or if there is no mailbox configuration the mailbox specified
    # in code (Props.withMailbox) is used.
    # If there is a mailbox defined in the configured dispatcher then that
    # overrides this setting.
    mailbox = ""

    # routing (load-balance) scheme to use
    # - available: "from-code", "round-robin", "random", "smallest-mailbox",
    #              "scatter-gather", "broadcast"
```

```
        # - or:        Fully qualified class name of the router class.
        #              The class must extend akka.routing.CustomRouterConfig and
        #              have a public constructor with com.typesafe.config.Config
        #              and optional akka.actor.DynamicAccess parameter.
        # - default is "from-code";
        # Whether or not an actor is transformed to a Router is decided in code
        # only (Props.withRouter). 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 route messages to these paths using ActorSelection,
        #   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 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

        # number of virtual nodes per node for consistent-hashing router
        virtual-nodes-factor = 10

        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 = []
        }

        # To use a dedicated dispatcher for the routees of the pool you can
        # define the dispatcher configuration inline with the property name
        # 'pool-dispatcher' in the deployment section of the router.
        # For example:
        # pool-dispatcher {
        #   fork-join-executor.parallelism-min = 5
        #   fork-join-executor.parallelism-max = 5
        # }

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

          enabled = off

          # 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 used to evaluate if a 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

        # 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, PinnedDispatcher, or a FQCN to a class inheriting
    # MessageDispatcherConfigurator with a public constructor with
    # both com.typesafe.config.Config parameter and
    # akka.dispatch.DispatcherPrerequisites parameters.
    # PinnedDispatcher must be used together with executor=thread-pool-executor.
    type = "Dispatcher"

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

    # This will be used if you have set "executor = "default-executor"".
    # If an ActorSystem is created with a given ExecutionContext, this
    # ExecutionContext will be used as the default executor for all
    # dispatchers in the ActorSystem configured with
    # executor = "default-executor". Note that "default-executor"
    # is the default value for executor, and therefore used if not
    # specified otherwise. If no ExecutionContext is given,
    # the executor configured in "fallback" will be used.
    default-executor {
      fallback = "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
```

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

    # The core pool size factor is used to determine thread pool core size
    # using the following formula: ceil(available processors * factor).
    # Resulting size is then bounded by the core-pool-size-min and
    # core-pool-size-max values.
    core-pool-size-factor = 3.0

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

    # Minimum number of threads to cap factor-based max number to
    # (if using a bounded task queue)
    max-pool-size-min = 8

    # Max no of threads (if using a bounded task queue) is determined by
    # calculating: ceil(available processors * factor)
    max-pool-size-factor  = 3.0

    # Max number of threads to cap factor-based max number to
    # (if using a  bounded task queue)
    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

  # 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

    # If this dispatcher requires a specific type of mailbox, specify the
    # fully-qualified class name here; the actually created mailbox will
    # be a subtype of this type. The empty string signifies no requirement.
    mailbox-requirement = ""
  }

  default-mailbox {
    # FQCN of the MailboxType. The Class of the FQCN must have a public
    # constructor with
    # (akka.actor.ActorSystem.Settings, com.typesafe.config.Config) parameters.
    mailbox-type = "akka.dispatch.UnboundedMailbox"

    # If the mailbox is bounded then it uses this setting to determine its
    # capacity. The provided value must be positive.
    # NOTICE:
    # Up to version 2.1 the mailbox type was determined based on this setting;
    # this is no longer the case, the type must explicitly be a bounded mailbox.
    mailbox-capacity = 1000

    # If the mailbox is bounded then this is the timeout for enqueueing
    # in case the mailbox is full. Negative values signify infinite
    # timeout, which should be avoided as it bears the risk of dead-lock.
    mailbox-push-timeout-time = 10s

    # 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
  }

  mailbox {
    # Mapping between message queue semantics and mailbox configurations.
    # Used by akka.dispatch.RequiresMessageQueue[T] to enforce different
    # mailbox types on actors.
    # If your Actor implements RequiresMessageQueue[T], then when you create
    # an instance of that actor its mailbox type will be decided by looking
    # up a mailbox configuration via T in this mapping
    requirements {
      "akka.dispatch.UnboundedMessageQueueSemantics" =
        akka.actor.mailbox.unbounded-queue-based
      "akka.dispatch.BoundedMessageQueueSemantics" =
        akka.actor.mailbox.bounded-queue-based
      "akka.dispatch.DequeBasedMessageQueueSemantics" =
        akka.actor.mailbox.unbounded-deque-based
      "akka.dispatch.UnboundedDequeBasedMessageQueueSemantics" =
        akka.actor.mailbox.unbounded-deque-based
      "akka.dispatch.BoundedDequeBasedMessageQueueSemantics" =
        akka.actor.mailbox.bounded-deque-based
      "akka.dispatch.MultipleConsumerSemantics" =
        akka.actor.mailbox.unbounded-queue-based
    }

    unbounded-queue-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.UnboundedMailbox"
    }

    bounded-queue-based {
```

```
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.BoundedMailbox"
    }

    unbounded-deque-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.UnboundedDequeBasedMailbox"
    }

    bounded-deque-based {
      # FQCN of the MailboxType, The Class of the FQCN must have a public
      # constructor with (akka.actor.ActorSystem.Settings,
      # com.typesafe.config.Config) parameters.
      mailbox-type = "akka.dispatch.BoundedDequeBasedMailbox"
    }
  }

  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 et.c.)
    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

    # enable DEBUG logging of unhandled messages
    unhandled = off

    # enable WARN logging of misconfigured routers
    router-misconfiguration = off
  }

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

  # 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 {
    "[B" = bytes
    "java.io.Serializable" = java
  }
```

```
  # Configuration items which are used by the akka.actor.ActorDSL._ methods
  dsl {
    # Maximum queue size of the actor created by newInbox(); this protects
    # against faulty programs which use select() and consistently miss messages
    inbox-size = 1000

    # Default timeout to assume for operations like Inbox.receive et al
    default-timeout = 5s
  }
}

# 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 details.
scheduler {
  # The LightArrayRevolverScheduler is used as the default scheduler in the
  # system. It does not execute the scheduled tasks on exact time, but on every
  # tick, it will run everything that is (over)due. 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.
  # Note that it might take up to 1 tick to stop the Timer, so setting the
  # tick-duration to a high value will make shutting down the actor system
  # take longer.
  tick-duration = 10ms

  # The timer uses a circular wheel of buckets to store the timer tasks.
  # This should be set such that the majority of scheduled timeouts (for high
  # scheduling frequency) will be shorter than one rotation of the wheel
  # (ticks-per-wheel * ticks-duration)
  # THIS MUST BE A POWER OF TWO!
  ticks-per-wheel = 512

  # This setting selects the timer implementation which shall be loaded at
  # system start-up.
  # The class given here must implement the akka.actor.Scheduler interface
  # and offer a public constructor which takes three arguments:
  #  1) com.typesafe.config.Config
  #  2) akka.event.LoggingAdapter
  #  3) java.util.concurrent.ThreadFactory
  implementation = akka.actor.LightArrayRevolverScheduler

  # When shutting down the scheduler, there will typically be a thread which
  # needs to be stopped, and this timeout determines how long to wait for
  # that to happen. In case of timeout the shutdown of the actor system will
  # proceed without running possibly still enqueued tasks.
  shutdown-timeout = 5s
}

io {

  # By default the select loops run on dedicated threads, hence using a
  # PinnedDispatcher
  pinned-dispatcher {
    type = "PinnedDispatcher"
    executor = "thread-pool-executor"
    thread-pool-executor.allow-core-pool-timeout = off
  }

  tcp {
```

```
    # The number of selectors to stripe the served channels over; each of
    # these will use one select loop on the selector-dispatcher.
    nr-of-selectors = 1

    # Maximum number of open channels supported by this TCP module; there is
    # no intrinsic general limit, this setting is meant to enable DoS
    # protection by limiting the number of concurrently connected clients.
    # Also note that this is a "soft" limit; in certain cases the implementation
    # will accept a few connections more or a few less than the number configured
    # here. Must be an integer > 0 or "unlimited".
    max-channels = 256000

    # When trying to assign a new connection to a selector and the chosen
    # selector is at full capacity, retry selector choosing and assignment
    # this many times before giving up
    selector-association-retries = 10

    # The maximum number of connection that are accepted in one go,
    # higher numbers decrease latency, lower numbers increase fairness on
    # the worker-dispatcher
    batch-accept-limit = 10

    # The number of bytes per direct buffer in the pool used to read or write
    # network data from the kernel.
    direct-buffer-size = 128 KiB

    # The maximal number of direct buffers kept in the direct buffer pool for
    # reuse.
    direct-buffer-pool-limit = 1000

    # The duration a connection actor waits for a `Register` message from
    # its commander before aborting the connection.
    register-timeout = 5s

    # The maximum number of bytes delivered by a `Received` message. Before
    # more data is read from the network the connection actor will try to
    # do other work.
    max-received-message-size = unlimited

    # Enable fine grained logging of what goes on inside the implementation.
    # Be aware that this may log more than once per message sent to the actors
    # of the tcp implementation.
    trace-logging = off

    # Fully qualified config path which holds the dispatcher configuration
    # to be used for running the select() calls in the selectors
    selector-dispatcher = "akka.io.pinned-dispatcher"

    # Fully qualified config path which holds the dispatcher configuration
    # for the read/write worker actors
    worker-dispatcher = "akka.actor.default-dispatcher"

    # Fully qualified config path which holds the dispatcher configuration
    # for the selector management actors
    management-dispatcher = "akka.actor.default-dispatcher"

    # Fully qualified config path which holds the dispatcher configuration
    # on which file IO tasks are scheduled
    file-io-dispatcher = "akka.actor.default-dispatcher"

    # The maximum number of bytes (or "unlimited") to transfer in one batch
    # when using `WriteFile` command which uses `FileChannel.transferTo` to
    # pipe files to a TCP socket. On some OS like Linux `FileChannel.transferTo`
```

```
    # may block for a long time when network IO is faster than file IO.
    # Decreasing the value may improve fairness while increasing may improve
    # throughput.
    file-io-transferTo-limit = 512 KiB

    # The number of times to retry the `finishConnect` call after being notified about
    # OP_CONNECT. Retries are needed if the OP_CONNECT notification doesn't imply that
    # `finishConnect` will succeed, which is the case on Android.
    finish-connect-retries = 5
  }

  udp {

    # The number of selectors to stripe the served channels over; each of
    # these will use one select loop on the selector-dispatcher.
    nr-of-selectors = 1

    # Maximum number of open channels supported by this UDP module Generally
    # UDP does not require a large number of channels, therefore it is
    # recommended to keep this setting low.
    max-channels = 4096

    # The select loop can be used in two modes:
    # - setting "infinite" will select without a timeout, hogging a thread
    # - setting a positive timeout will do a bounded select call,
    #   enabling sharing of a single thread between multiple selectors
    #   (in this case you will have to use a different configuration for the
    #   selector-dispatcher, e.g. using "type=Dispatcher" with size 1)
    # - setting it to zero means polling, i.e. calling selectNow()
    select-timeout = infinite

    # When trying to assign a new connection to a selector and the chosen
    # selector is at full capacity, retry selector choosing and assignment
    # this many times before giving up
    selector-association-retries = 10

    # The maximum number of datagrams that are read in one go,
    # higher numbers decrease latency, lower numbers increase fairness on
    # the worker-dispatcher
    receive-throughput = 3

    # The number of bytes per direct buffer in the pool used to read or write
    # network data from the kernel.
    direct-buffer-size = 128 KiB

    # The maximal number of direct buffers kept in the direct buffer pool for
    # reuse.
    direct-buffer-pool-limit = 1000

    # The maximum number of bytes delivered by a `Received` message. Before
    # more data is read from the network the connection actor will try to
    # do other work.
    received-message-size-limit = unlimited

    # Enable fine grained logging of what goes on inside the implementation.
    # Be aware that this may log more than once per message sent to the actors
    # of the tcp implementation.
    trace-logging = off

    # Fully qualified config path which holds the dispatcher configuration
    # to be used for running the select() calls in the selectors
    selector-dispatcher = "akka.io.pinned-dispatcher"
```

```
    # Fully qualified config path which holds the dispatcher configuration
    # for the read/write worker actors
    worker-dispatcher = "akka.actor.default-dispatcher"

    # Fully qualified config path which holds the dispatcher configuration
    # for the selector management actors
    management-dispatcher = "akka.actor.default-dispatcher"
  }

  udp-connected {

    # The number of selectors to stripe the served channels over; each of
    # these will use one select loop on the selector-dispatcher.
    nr-of-selectors = 1

    # Maximum number of open channels supported by this UDP module Generally
    # UDP does not require a large number of channels, therefore it is
    # recommended to keep this setting low.
    max-channels = 4096

    # The select loop can be used in two modes:
    # - setting "infinite" will select without a timeout, hogging a thread
    # - setting a positive timeout will do a bounded select call,
    #   enabling sharing of a single thread between multiple selectors
    #   (in this case you will have to use a different configuration for the
    #   selector-dispatcher, e.g. using "type=Dispatcher" with size 1)
    # - setting it to zero means polling, i.e. calling selectNow()
    select-timeout = infinite

    # When trying to assign a new connection to a selector and the chosen
    # selector is at full capacity, retry selector choosing and assignment
    # this many times before giving up
    selector-association-retries = 10

    # The maximum number of datagrams that are read in one go,
    # higher numbers decrease latency, lower numbers increase fairness on
    # the worker-dispatcher
    receive-throughput = 3

    # The number of bytes per direct buffer in the pool used to read or write
    # network data from the kernel.
    direct-buffer-size = 128 KiB

    # The maximal number of direct buffers kept in the direct buffer pool for
    # reuse.
    direct-buffer-pool-limit = 1000

    # The maximum number of bytes delivered by a `Received` message. Before
    # more data is read from the network the connection actor will try to
    # do other work.
    received-message-size-limit = unlimited

    # Enable fine grained logging of what goes on inside the implementation.
    # Be aware that this may log more than once per message sent to the actors
    # of the tcp implementation.
    trace-logging = off

    # Fully qualified config path which holds the dispatcher configuration
    # to be used for running the select() calls in the selectors
    selector-dispatcher = "akka.io.pinned-dispatcher"

    # Fully qualified config path which holds the dispatcher configuration
    # for the read/write worker actors
```

```
      worker-dispatcher = "akka.actor.default-dispatcher"

      # Fully qualified config path which holds the dispatcher configuration
      # for the selector management actors
      management-dispatcher = "akka.actor.default-dispatcher"
    }

  }


}
```

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-camel

```
#####################################
# Akka Camel Reference Config File #
#####################################

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

akka {
  camel {
    # FQCN of the ContextProvider to be used to create or locate a CamelContext
    # it must implement akka.camel.ContextProvider and have a no-arg constructor
    # the built-in default create a fresh DefaultCamelContext
    context-provider = akka.camel.DefaultContextProvider

    # Whether JMX should be enabled or disabled for the Camel Context
    jmx = off
    # enable/disable streaming cache on the Camel Context
    streamingCache = on
    consumer {
      # Configured setting which determines whether one-way communications
```

```
      # between an endpoint and this consumer actor
      # should be auto-acknowledged or application-acknowledged.
      # This flag has only effect when exchange is in-only.
      auto-ack = on

      # When endpoint is out-capable (can produce responses) reply-timeout is the
      # maximum time the endpoint can take to send the response before the message
      # exchange fails. This setting is used for out-capable, in-only,
      # manually acknowledged communication.
      reply-timeout = 1m

      # The duration of time to await activation of an endpoint.
      activation-timeout = 10s
    }

    #Scheme to FQCN mappings for CamelMessage body conversions
    conversions {
      "file" = "java.io.InputStream"
    }
  }
}
```

akka-cluster

```
#####################################
# Akka Cluster Reference Config File #
#####################################

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

akka {

  cluster {
    # Initial contact points of the cluster.
    # The nodes to join automatically at startup.
    # Comma separated full URIs defined by a string on the form of
    # "akka.tcp://system@hostname:port"
    # Leave as empty if the node is supposed to be joined manually.
    seed-nodes = []

    # how long to wait for one of the seed nodes to reply to initial join request
    seed-node-timeout = 5s

    # If a join request fails it will be retried after this period.
    # Disable join retry by specifying "off".
    retry-unsuccessful-join-after = 10s

    # Should the 'leader' in the cluster be allowed to automatically mark
    # unreachable nodes as DOWN after a configured time of unreachability?
    # Using auto-down implies that two separate clusters will automatically be
    # formed in case of network partition.
    # Disable with "off" or specify a duration to enable auto-down.
    auto-down-unreachable-after = off

    # deprecated in 2.3, use 'auto-down-unreachable-after' instead
    auto-down = off

    # The roles of this member. List of strings, e.g. roles = ["A", "B"].
    # The roles are part of the membership information and can be used by
    # routers or other services to distribute work to certain member types,
```

```
    # e.g. front-end and back-end nodes.
    roles = []

    role {
      # Minimum required number of members of a certain role before the leader
      # changes member status of 'Joining' members to 'Up'. Typically used together
      # with 'Cluster.registerOnMemberUp' to defer some action, such as starting
      # actors, until the cluster has reached a certain size.
      # E.g. to require 2 nodes with role 'frontend' and 3 nodes with role 'backend':
      #   frontend.min-nr-of-members = 2
      #   backend.min-nr-of-members = 3
      #<role-name>.min-nr-of-members = 1
    }

    # Minimum required number of members before the leader changes member status
    # of 'Joining' members to 'Up'. Typically used together with
    # 'Cluster.registerOnMemberUp' to defer some action, such as starting actors,
    # until the cluster has reached a certain size.
    min-nr-of-members = 1

    # Enable/disable info level logging of cluster events
    log-info = on

    # Enable or disable JMX MBeans for management of the cluster
    jmx.enabled = on

    # how long should the node wait before starting the periodic tasks
    # maintenance tasks?
    periodic-tasks-initial-delay = 1s

    # how often should the node send out gossip information?
    gossip-interval = 1s

    # discard incoming gossip messages if not handled within this duration
    gossip-time-to-live = 2s

    # how often should the leader perform maintenance tasks?
    leader-actions-interval = 1s

    # how often should the node move nodes, marked as unreachable by the failure
    # detector, out of the membership ring?
    unreachable-nodes-reaper-interval = 1s

    # How often the current internal stats should be published.
    # A value of 0s can be used to always publish the stats, when it happens.
    # Disable with "off".
    publish-stats-interval = off

    # The id of the dispatcher to use for cluster actors. If not specified
    # default dispatcher is used.
    # If specified you need to define the settings of the actual dispatcher.
    use-dispatcher = ""

    # Gossip to random node with newer or older state information, if any with
    # this probability. Otherwise Gossip to any random live node.
    # Probability value is between 0.0 and 1.0. 0.0 means never, 1.0 means always.
    gossip-different-view-probability = 0.8

    # Reduced the above probability when the number of nodes in the cluster
    # greater than this value.
    reduce-gossip-different-view-probability = 400

    # Settings for the Phi accrual failure detector (http://ddg.jaist.ac.jp/pub/HDY+04.pdf
```

```
    # [Hayashibara et al]) used by the cluster subsystem to detect unreachable
    # members.
    failure-detector {

      # FQCN of the failure detector implementation.
      # It must implement akka.remote.FailureDetector and have
      # a public constructor with a com.typesafe.config.Config and
      # akka.actor.EventStream parameter.
      implementation-class = "akka.remote.PhiAccrualFailureDetector"

      # How often keep-alive heartbeat messages should be sent to each connection.
      heartbeat-interval = 1 s

      # Defines the failure detector threshold.
      # 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.
      threshold = 8.0

      # Number of the samples of inter-heartbeat arrival times to adaptively
      # calculate the failure timeout for connections.
      max-sample-size = 1000

      # Minimum standard deviation to use for the normal distribution in
      # AccrualFailureDetector. Too low standard deviation might result in
      # too much sensitivity for sudden, but normal, deviations in heartbeat
      # inter arrival times.
      min-std-deviation = 100 ms

      # Number of potentially lost/delayed heartbeats that will be
      # accepted before considering it to be an anomaly.
      # This margin is important to be able to survive sudden, occasional,
      # pauses in heartbeat arrivals, due to for example garbage collect or
      # network drop.
      acceptable-heartbeat-pause = 3 s

      # Number of member nodes that each member will send heartbeat messages to,
      # i.e. each node will be monitored by this number of other nodes.
      monitored-by-nr-of-members = 5

      # After the heartbeat request has been sent the first failure detection
      # will start after this period, even though no heartbeat mesage has
      # been received.
      expected-response-after = 5 s

    }

    metrics {
      # Enable or disable metrics collector for load-balancing nodes.
      enabled = on

      # FQCN of the metrics collector implementation.
      # It must implement akka.cluster.MetricsCollector and
      # have public constructor with akka.actor.ActorSystem parameter.
      # The default SigarMetricsCollector uses JMX and Hyperic SIGAR, if SIGAR
      # is on the classpath, otherwise only JMX.
      collector-class = "akka.cluster.SigarMetricsCollector"

      # How often metrics are sampled on a node.
      # Shorter interval will collect the metrics more often.
      collect-interval = 3s
```

```
    # How often a node publishes metrics information.
    gossip-interval = 3s

    # How quickly the exponential weighting of past data is decayed compared to
    # new data. Set lower to increase the bias toward newer values.
    # The relevance of each data sample is halved for every passing half-life
    # duration, i.e. after 4 times the half-life, a data sample's relevance is
    # reduced to 6% of its original relevance. The initial relevance of a data
    # sample is given by 1 - 0.5 ^ (collect-interval / half-life).
    # See http://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average
    moving-average-half-life = 12s
  }

  # If the tick-duration of the default scheduler is longer than the
  # tick-duration configured here a dedicated scheduler will be used for
  # periodic tasks of the cluster, otherwise the default scheduler is used.
  # See akka.scheduler settings for more details.
  scheduler {
    tick-duration = 33ms
    ticks-per-wheel = 512
  }

}

# Default configuration for routers
actor.deployment.default {
  # MetricsSelector to use
  # - available: "mix", "heap", "cpu", "load"
  # - or: Fully qualified class name of the MetricsSelector class.
  #       The class must extend akka.cluster.routing.MetricsSelector
  #       and have a public constructor with com.typesafe.config.Config
  #       parameter.
  # - default is "mix"
  metrics-selector = mix
}
actor.deployment.default.cluster {
  # enable cluster aware router that deploys to nodes in the cluster
  enabled = off

  # Maximum number of routees that will be deployed on each cluster
  # member node.
  # Note that nr-of-instances defines total number of routees, but
  # number of routees per node will not be exceeded, i.e. if you
  # define nr-of-instances = 50 and max-nr-of-instances-per-node = 2
  # it will deploy 2 routees per new member in the cluster, up to
  # 25 members.
  max-nr-of-instances-per-node = 1

  # Defines if routees are allowed to be located on the same node as
  # the head router actor, or only on remote nodes.
  # Useful for master-worker scenario where all routees are remote.
  allow-local-routees = on

  # Deprecated in 2.3, use routees.paths instead
  routees-path = ""

  # Use members with specified role, or all members if undefined or empty.
  use-role = ""

}

# Protobuf serializer for cluster messages
actor {
```

```
    serializers {
      akka-cluster = "akka.cluster.protobuf.ClusterMessageSerializer"
    }

    serialization-bindings {
      "akka.cluster.ClusterMessage" = akka-cluster
    }

    router.type-mapping {
      adaptive-pool = "akka.cluster.routing.AdaptiveLoadBalancingPool"
      adaptive-group = "akka.cluster.routing.AdaptiveLoadBalancingGroup"
    }
  }

}
```

akka-multi-node-testkit

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

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

akka {
  testconductor {

    # Timeout for joining a barrier: this is the maximum time any participants
    # waits for everybody else to join a named barrier.
    barrier-timeout = 30s

    # Timeout for interrogation of TestConductor's Controller actor
    query-timeout = 5s

    # Threshold for packet size in time unit above which the failure injector will
    # split the packet and deliver in smaller portions; do not give value smaller
    # than HashedWheelTimer resolution (would not make sense)
    packet-split-threshold = 100ms

    # amount of time for the ClientFSM to wait for the connection to the conductor
    # to be successful
    connect-timeout = 20s

    # Number of connect attempts to be made to the conductor controller
    client-reconnects = 10

    # minimum time interval which is to be inserted between reconnect attempts
    reconnect-backoff = 1s

    netty {
      # (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 = 1

        # 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 = 1.0
```

```
        # Max number of threads to cap factor-based number to
        pool-size-max = 2
      }

      # (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 = 1

        # 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 = 1.0

        # Max number of threads to cap factor-based number to
        pool-size-max = 2
      }
    }
  }
}
```

akka-persistence

```
#############################################
# Akka Persistence Reference Config File #
#############################################



akka {

  # Protobuf serialization for persistent messages
  actor {

    serializers {

      akka-persistence-snapshot = "akka.persistence.serialization.SnapshotSerializer"
      akka-persistence-message = "akka.persistence.serialization.MessageSerializer"
    }

    serialization-bindings {

      "akka.persistence.serialization.Snapshot" = akka-persistence-snapshot
      "akka.persistence.serialization.Message" = akka-persistence-message
    }
  }

  persistence {

    journal {

      # Maximum size of a persistent message batch written to the journal.
      # Only applies to internally created batches by processors that receive
      # persistent messages individually. Application-defined batches, even if
      # larger than this setting, are always written as a single isolated batch.
      max-message-batch-size = 200

      # Maximum size of a confirmation batch written to the journal.
      max-confirmation-batch-size = 10000
```

```
    # Maximum size of a deletion batch written to the journal.
    max-deletion-batch-size = 10000

    # Path to the journal plugin to be used
    plugin = "akka.persistence.journal.leveldb"

    # In-memory journal plugin.
    inmem {

      # Class name of the plugin.
      class = "akka.persistence.journal.inmem.InmemJournal"

      # Dispatcher for the plugin actor.
      plugin-dispatcher = "akka.actor.default-dispatcher"
    }

    # LevelDB journal plugin.
    leveldb {

      # Class name of the plugin.
      class = "akka.persistence.journal.leveldb.LeveldbJournal"

      # Dispatcher for the plugin actor.
      plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"

      # Dispatcher for message replay.
      replay-dispatcher = "akka.persistence.dispatchers.default-replay-dispatcher"

      # Storage location of LevelDB files.
      dir = "journal"

      # Use fsync on write
      fsync = on

      # Verify checksum on read.
      checksum = off

      # Native LevelDB (via JNI) or LevelDB Java port
      native = on
    }

    # Shared LevelDB journal plugin (for testing only).
    leveldb-shared {

      # Class name of the plugin.
      class = "akka.persistence.journal.leveldb.SharedLeveldbJournal"

      # Dispatcher for the plugin actor.
      plugin-dispatcher = "akka.actor.default-dispatcher"

      # timeout for async journal operations
      timeout = 10s

      store {

        # Dispatcher for shared store actor.
        store-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"

        # Dispatcher for message replay.
        replay-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"

        # Storage location of LevelDB files.
        dir = "journal"
```

```
      # Use fsync on write
      fsync = on

      # Verify checksum on read.
      checksum = off

      # Native LevelDB (via JNI) or LevelDB Java port
      native = on
    }
  }
}

snapshot-store {

  # Path to the snapshot store plugin to be used
  plugin = "akka.persistence.snapshot-store.local"

  # Local filesystem snapshot store plugin.
  local {

    # Class name of the plugin.
    class = "akka.persistence.snapshot.local.LocalSnapshotStore"

    # Dispatcher for the plugin actor.
    plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"

    # Dispatcher for streaming snapshot IO.
    stream-dispatcher = "akka.persistence.dispatchers.default-stream-dispatcher"

    # Storage location of snapshot files.
    dir = "snapshots"
  }
}

view {

  # Automated incremental view update.
  auto-update = on

  # Interval between incremental updates
  auto-update-interval = 5s

  # Maximum number of messages to replay per incremental view update. Set to
  # -1 for no upper limit.
  auto-update-replay-max = -1
}

dispatchers {
  default-plugin-dispatcher {
    type = PinnedDispatcher
    executor = "thread-pool-executor"
  }
  default-replay-dispatcher {
    type = Dispatcher
    executor = "fork-join-executor"
    fork-join-executor {
      parallelism-min = 2
      parallelism-max = 8
    }
  }
  default-stream-dispatcher {
    type = Dispatcher
```

```
        executor = "fork-join-executor"
        fork-join-executor {
          parallelism-min = 2
          parallelism-max = 8
        }
      }
    }
  }
}
```

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 {
      akka-containers = "akka.remote.serialization.MessageContainerSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      daemon-create = "akka.remote.serialization.DaemonMsgCreateSerializer"
    }


    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
      "akka.actor.ActorSelectionMessage" = akka-containers
      "com.google.protobuf.GeneratedMessage" = proto
      "akka.remote.DaemonMsgCreate" = daemon-create
    }

    deployment {

      default {

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

        target {

          # A list of hostnames and ports for instantiating the children of a
          # router
          #   The format should be on "akka.tcp://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 {

  ### General settings

  # Timeout after which the startup of the remoting subsystem is considered
  # to be failed. Increase this value if your transport drivers (see the
  # enabled-transports section) need longer time to be loaded.
  startup-timeout = 10 s

  # Timout after which the graceful shutdown of the remoting subsystem is
  # considered to be failed. After the timeout the remoting system is
  # forcefully shut down. Increase this value if your transport drivers
  # (see the enabled-transports section) need longer time to stop properly.
  shutdown-timeout = 10 s

  # Before shutting down the drivers, the remoting subsystem attempts to flush
  # all pending writes. This setting controls the maximum time the remoting is
  # willing to wait before moving on to shut down the drivers.
  flush-wait-on-shutdown = 2 s

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

  # Controls the backoff interval after a refused write is reattempted.
  # (Transports may refuse writes if their internal buffer is full)
  backoff-interval = 0.01 s

  # Acknowledgment timeout of management commands sent to the transport stack.
  command-ack-timeout = 30 s

  # If set to a nonempty string remoting will use the given dispatcher for
  # its internal actors otherwise the default dispatcher is used. Please note
  # that since remoting can load arbitrary 3rd party drivers (see
  # "enabled-transport" and "adapters" entries) it is not guaranteed that
  # every module will respect this setting.
  use-dispatcher = "akka.remote.default-remote-dispatcher"

  ### Security settings

  # Enable untrusted mode for full security of server managed actors, prevents
  # system messages to be send by clients, e.g. messages like 'Create',
  # 'Suspend', 'Resume', 'Terminate', 'Supervise', 'Link' etc.
  untrusted-mode = off

  # When 'untrusted-mode=on' inbound actor selections are by default discarded.
  # Actors with paths defined in this white list are granted permission to receive actor
  # selections messages.
  # E.g. trusted-selection-paths = ["/user/receptionist", "/user/namingService"]
  trusted-selection-paths = []

  # Should the remote server require that its peers share the same
  # secure-cookie (defined in the 'remote' section)? Secure cookies are passed
  # between during the initial handshake. Connections are refused if the initial
  # message contains a mismatching cookie or the cookie is missing.
  require-cookie = off
```

```
# Generate your own with the script availbale in
# '$AKKA_HOME/scripts/generate_config_with_secure_cookie.sh' or using
# 'akka.util.Crypt.generateSecureCookie'
secure-cookie = ""

### Logging

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

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

# Sets the log granularity level at which Akka logs remoting events. This setting
# can take the values OFF, ERROR, WARNING, INFO, DEBUG, or ON. For compatibility
# reasons the setting "on" will default to "debug" level. Please note that the effective
# logging level is still determined by the global logging level of the actor system:
# for example debug level remoting events will be only logged if the system
# is running with debug level logging.
# Failures to deserialize received messages also fall under this flag.
log-remote-lifecycle-events = on

# Logging of message types with payload size in bytes larger than
# this value. Maximum detected size per message type is logged once,
# with an increase threshold of 10%.
# By default this feature is turned off. Activate it by setting the property to
# a value in bytes, such as 1000b. Note that for all messages larger than this
# limit there will be extra performance and scalability cost.
log-frame-size-exceeding = off

### Failure detection and recovery

# Settings for the Phi accrual failure detector (http://ddg.jaist.ac.jp/pub/HDY+04.pdf
# [Hayashibara et al]) used by the remoting subsystem to detect failed
# connections.
transport-failure-detector {

  # FQCN of the failure detector implementation.
  # It must implement akka.remote.FailureDetector and have
  # a public constructor with a com.typesafe.config.Config and
  # akka.actor.EventStream parameter.
  implementation-class = "akka.remote.PhiAccrualFailureDetector"

  # How often keep-alive heartbeat messages should be sent to each connection.
  heartbeat-interval = 4 s

  # Defines the failure detector threshold.
  # 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.
  threshold = 7.0

  # Number of the samples of inter-heartbeat arrival times to adaptively
  # calculate the failure timeout for connections.
  max-sample-size = 100

  # Minimum standard deviation to use for the normal distribution in
  # AccrualFailureDetector. Too low standard deviation might result in
  # too much sensitivity for sudden, but normal, deviations in heartbeat
```

```
    # inter arrival times.
    min-std-deviation = 100 ms

    # Number of potentially lost/delayed heartbeats that will be
    # accepted before considering it to be an anomaly.
    # This margin is important to be able to survive sudden, occasional,
    # pauses in heartbeat arrivals, due to for example garbage collect or
    # network drop.
    acceptable-heartbeat-pause = 10 s
  }

  # Settings for the Phi accrual failure detector (http://ddg.jaist.ac.jp/pub/HDY+04.pdf
  # [Hayashibara et al]) used for remote death watch.
  watch-failure-detector {

    # FQCN of the failure detector implementation.
    # It must implement akka.remote.FailureDetector and have
    # a public constructor with a com.typesafe.config.Config and
    # akka.actor.EventStream parameter.
    implementation-class = "akka.remote.PhiAccrualFailureDetector"

    # How often keep-alive heartbeat messages should be sent to each connection.
    heartbeat-interval = 1 s

    # Defines the failure detector threshold.
    # 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.
    threshold = 10.0

    # Number of the samples of inter-heartbeat arrival times to adaptively
    # calculate the failure timeout for connections.
    max-sample-size = 200

    # Minimum standard deviation to use for the normal distribution in
    # AccrualFailureDetector. Too low standard deviation might result in
    # too much sensitivity for sudden, but normal, deviations in heartbeat
    # inter arrival times.
    min-std-deviation = 100 ms

    # Number of potentially lost/delayed heartbeats that will be
    # accepted before considering it to be an anomaly.
    # This margin is important to be able to survive sudden, occasional,
    # pauses in heartbeat arrivals, due to for example garbage collect or
    # network drop.
    acceptable-heartbeat-pause = 10 s


    # How often to check for nodes marked as unreachable by the failure
    # detector
    unreachable-nodes-reaper-interval = 1s

    # After the heartbeat request has been sent the first failure detection
    # will start after this period, even though no heartbeat mesage has
    # been received.
    expected-response-after = 3 s

  }

  # After failed to establish an outbound connection, the remoting will mark the
  # address as failed. This configuration option controls how much time should
  # be elapsed before reattempting a new connection. While the address is
```

```
    # gated, all messages sent to the address are delivered to dead-letters.
    # Since this setting limits the rate of reconnects setting it to a
    # very short interval (i.e. less than a second) may result in a storm of
    # reconnect attempts.
    retry-gate-closed-for = 5 s

    # After catastrophic communication failures that result in the loss of system
    # messages or after the remote DeathWatch triggers the remote system gets
    # quarantined to prevent inconsistent behavior.
    # This setting controls how long the Quarantine marker will be kept around
    # before being removed to avoid long-term memory leaks.
    # WARNING: DO NOT change this to a small value to re-enable communication with
    # quarantined nodes. Such feature is not supported and any behavior between
    # the affected systems after lifting the quarantine is undefined.
    prune-quarantine-marker-after = 5 d

    # This setting defines the maximum number of unacknowledged system messages
    # allowed for a remote system. If this limit is reached the remote system is
    # declared to be dead and its UID marked as tainted.
    system-message-buffer-size = 1000

    # This setting defines the maximum idle time after an individual
    # acknowledgement for system messages is sent. System message delivery
    # is guaranteed by explicit acknowledgement messages. These acks are
    # piggybacked on ordinary traffic messages. If no traffic is detected
    # during the time period configured here, the remoting will send out
    # an individual ack.
    system-message-ack-piggyback-timeout = 0.3 s

    # This setting defines the time after internal management signals
    # between actors (used for DeathWatch and supervision) that have not been
    # explicitly acknowledged or negatively acknowledged are resent.
    # Messages that were negatively acknowledged are always immediately
    # resent.
    resend-interval = 2 s

    # WARNING: this setting should not be not changed unless all of its consequences
    # are properly understood which assumes experience with remoting internals
    # or expert advice.
    # This setting defines the time after redelivery attempts of internal management
    # signals are stopped to a remote system that has been not confirmed to be alive by
    # this system before.
    initial-system-message-delivery-timeout = 3 m

    ### Transports and adapters

    # List of the transport drivers that will be loaded by the remoting.
    # A list of fully qualified config paths must be provided where
    # the given configuration path contains a transport-class key
    # pointing to an implementation class of the Transport interface.
    # If multiple transports are provided, the address of the first
    # one will be used as a default address.
    enabled-transports = ["akka.remote.netty.tcp"]

    # Transport drivers can be augmented with adapters by adding their
    # name to the applied-adapters setting in the configuration of a
    # transport. The available adapters should be configured in this
    # section by providing a name, and the fully qualified name of
    # their corresponding implementation. The class given here
    # must implement akka.akka.remote.transport.TransportAdapterProvider
    # and have public constructor without parameters.
    adapters {
      gremlin = "akka.remote.transport.FailureInjectorProvider"
```

```
    trttl = "akka.remote.transport.ThrottlerProvider"
  }

  ### Default configuration for the Netty based transport drivers

  netty.tcp {
    # The class given here must implement the akka.remote.transport.Transport
    # interface and offer a public constructor which takes two arguments:
    #  1) akka.actor.ExtendedActorSystem
    #  2) com.typesafe.config.Config
    transport-class = "akka.remote.transport.netty.NettyTransport"

    # Transport drivers can be augmented with adapters by adding their
    # name to the applied-adapters list. The last adapter in the
    # list is the adapter immediately above the driver, while
    # the first one is the top of the stack below the standard
    # Akka protocol
    applied-adapters = []

    transport-protocol = tcp

    # 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

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

    # Enables SSL support on this transport
    enable-ssl = false

    # Sets the connectTimeoutMillis of all outbound connections,
    # i.e. how long a connect may take until it is timed out
    connection-timeout = 15 s

    # If set to "<id.of.dispatcher>" then the specified dispatcher
    # will be used to accept inbound connections, and perform IO. If "" then
    # dedicated threads will be used.
    # Please note that the Netty driver only uses this configuration and does
    # not read the "akka.remote.use-dispatcher" entry. Instead it has to be
    # configured manually to point to the same dispatcher if needed.
    use-dispatcher-for-io = ""

    # Sets the high water mark for the in and outbound sockets,
    # set to 0b for platform default
    write-buffer-high-water-mark = 0b

    # Sets the low water mark for the in and outbound sockets,
    # set to 0b for platform default
    write-buffer-low-water-mark = 0b

    # Sets the send buffer size of the Sockets,
    # set to 0b for platform default
    send-buffer-size = 256000b

    # Sets the receive buffer size of the Sockets,
    # set to 0b for platform default
    receive-buffer-size = 256000b

    # Maximum message size the transport will accept, but at least
    # 32000 bytes.
```

```
    # Please note that UDP does not support arbitrary large datagrams,
    # so this setting has to be chosen carefully when using UDP.
    # Both send-buffer-size and receive-buffer-size settings has to
    # be adjusted to be able to buffer messages of maximum size.
    maximum-frame-size = 128000b

    # Sets the size of the connection backlog
    backlog = 4096

    # Enables the TCP_NODELAY flag, i.e. disables Nagle's algorithm
    tcp-nodelay = on

    # Enables TCP Keepalive, subject to the O/S kernel's configuration
    tcp-keepalive = on

    # Enables SO_REUSEADDR, which determines when an ActorSystem can open
    # the specified listen port (the meaning differs between *nix and Windows)
    # Valid values are "on", "off" and "off-for-windows"
    # due to the following Windows bug: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4476
    # "off-for-windows" of course means that it's "on" for all other platforms
    tcp-reuse-addr = off-for-windows

    # 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 = 1.0

      # Max number of threads to cap factor-based number to
      pool-size-max = 2
    }

    # 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 = 1.0

      # Max number of threads to cap factor-based number to
      pool-size-max = 2
    }


  }

  netty.udp = ${akka.remote.netty.tcp}
  netty.udp {
    transport-protocol = udp
  }

  netty.ssl = ${akka.remote.netty.tcp}
  netty.ssl = {
    # Enable SSL/TLS encryption.
```

```
    # This must be enabled on both the client and server to work.
    enable-ssl = true

    security {
      # This is the Java Key Store used by the server connection
      key-store = "keystore"

      # This password is used for decrypting the key store
      key-store-password = "changeme"

      # This password is used for decrypting the key
      key-password = "changeme"

      # This is the Java Key Store used by the client connection
      trust-store = "truststore"

      # This password is used for decrypting the trust store
      trust-store-password = "changeme"

      # Protocol to use for SSL encryption, choose from:
      # Java 6 & 7:
      #   'SSLv3', 'TLSv1'
      # Java 7:
      #   'TLSv1.1', 'TLSv1.2'
      protocol = "TLSv1"

      # Example: ["TLS_RSA_WITH_AES_128_CBC_SHA", "TLS_RSA_WITH_AES_256_CBC_SHA"]
      # You need to install the JCE Unlimited Strength Jurisdiction Policy
      # Files to use AES 256.
      # More info here:
      # http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html#SunJCE
      enabled-algorithms = ["TLS_RSA_WITH_AES_128_CBC_SHA"]

      # There are three options, in increasing order of security:
      # "" or SecureRandom => (default)
      # "SHA1PRNG" => Can be slow because of blocking issues on Linux
      # "AES128CounterSecureRNG" => fastest startup and based on AES encryption
      # algorithm
      # "AES256CounterSecureRNG"
      # The following use one of 3 possible seed sources, depending on
      # availability: /dev/random, random.org and SecureRandom (provided by Java)
      # "AES128CounterInetRNG"
      # "AES256CounterInetRNG" (Install JCE Unlimited Strength Jurisdiction
      # Policy Files first)
      # Setting a value here may require you to supply the appropriate cipher
      # suite (see enabled-algorithms section above)
      random-number-generator = ""
    }
  }

  ### Default configuration for the failure injector transport adapter

  gremlin {
    # Enable debug logging of the failure injector transport adapter
    debug = off
  }

  ### Default dispatcher for the remoting subsystem

  default-remote-dispatcher {
    type = Dispatcher
    executor = "fork-join-executor"
    fork-join-executor {
```

```
        # Min number of threads to cap factor-based parallelism number to
        parallelism-min = 2
        parallelism-max = 2
      }
    }


  }

}
```

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-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"
      thread-pool-executor.allow-core-timeout = off
    }
  }
}
```

CHAPTER

THREE

ACTORS

3.1 Actors

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

3.1.1 Creating Actors

Note: 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 *Actor References, Paths and Addresses*.

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 the late 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`.

Note further that the return type of the behavior defined above is `Unit`; if the actor shall reply to the received message then this must be done explicitly as explained below.

The result of the `receive` method is a partial function object, which is stored within the actor as its "initial behavior", see Become/Unbecome for further information on changing the behavior of an actor after its construction.

Props

`Props` is a configuration class to specify options for the creation of actors, think of it as an immutable and thus freely shareable recipe for creating an actor including associated deployment information (e.g. which dispatcher to use, see more below). Here are some examples of how to create a `Props` instance.

```
import akka.actor.Props

val props1 = Props[MyActor]
val props2 = Props(new ActorWithArgs("arg")) // careful, see below
val props3 = Props(classOf[ActorWithArgs], "arg")
```

The second variant shows how to pass constructor arguments to the `Actor` being created, but it should only be used outside of actors as explained below.

The last line shows a possibility to pass constructor arguments regardless of the context it is being used in. The presence of a matching constructor is verified during construction of the `Props` object, resulting in an `IllegalArgumentEception` if no or multiple matching constructors are found.

Dangerous Variants

```
// NOT RECOMMENDED within another actor:
// encourages to close over enclosing class
val props7 = Props(new MyActor)
```

This method is not recommended to be used within another actor because it encourages to close over the enclosing scope, resulting in non-serializable `Props` and possibly race conditions (breaking the actor encapsulation). We will provide a macro-based solution in a future release which allows similar syntax without the headaches, at which point this variant will be properly deprecated. On the other hand using this variant in a `Props` factory in the actor's companion object as documented under "Recommended Practices" below is completely fine.

There were two use-cases for these methods: passing constructor arguments to the actor—which is solved by the newly introduced `Props.apply(clazz, args)` method above or the recommended practice below—and creating actors "on the spot" as anonymous classes. The latter should be solved by making these actors named classes instead (if they are not declared within a top-level `object` then the enclosing instance's `this` reference needs to be passed as the first argument).

> **Warning:** Declaring one actor within another is very dangerous and breaks actor encapsulation. Never pass an actor's `this` reference into `Props`!

Recommended Practices

It is a good idea to provide factory methods on the companion object of each `Actor` which help keeping the creation of suitable `Props` as close to the actor definition as possible. This also avoids the pitfalls associated with using the `Props.apply(...)` method which takes a by-name argument, since within a companion object the given code block will not retain a reference to its enclosing scope:

```
object DemoActor {
  /**
   * Create Props for an actor of this type.
   * @param magciNumber The magic number to be passed to this actor's constructor.
   * @return a Props for creating this actor, which can then be further configured
```

```
   *         (e.g. calling `.withDispatcher()` on it)
   */
  def props(magicNumber: Int): Props = Props(new DemoActor(magicNumber))
}

class DemoActor(magicNumber: Int) extends Actor {
  def receive = {
    case x: Int => sender() ! (x + magicNumber)
  }
}

class SomeOtherActor extends Actor {
  // Props(new DemoActor(42)) would not be safe
  context.actorOf(DemoActor.props(42), "demo")
  // ...
}
```

Creating Actors with Props

Actors are created by passing a `Props` instance into the `actorOf` factory method which is available on `ActorSystem` and `ActorContext`.

```
import akka.actor.ActorSystem

// ActorSystem is a heavy object: create only one per application
val system = ActorSystem("mySystem")
val myActor = system.actorOf(Props[MyActor], "myactor2")
```

Using the `ActorSystem` will create top-level actors, supervised by the actor system's provided guardian actor, while using an actor's context will create a child actor.

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

It is recommended to create a hierarchy of children, grand-children and so on such that it fits the logical failure-handling structure of the application, see *Actor Systems*.

The call to `actorOf` returns an instance of `ActorRef`. This is a handle to the actor instance and the only way to interact with it. 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.

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 `$`, but it may contain URL encoded characters (eg. `%20` for a blank space). If the given name is already in use by another child to the same parent an *InvalidActorNameException* is thrown.

Actors are automatically started asynchronously when created.

Dependency Injection

If your Actor has a constructor that takes parameters then those need to be part of the `Props` as well, as described above. But there are cases when a factory method must be used, for example when the actual constructor arguments are determined by a dependency injection framework.

```
import akka.actor.IndirectActorProducer

class DependencyInjector(applicationContext: AnyRef, beanName: String)
  extends IndirectActorProducer {
```

```
  override def actorClass = classOf[Actor]
  override def produce =
    // obtain fresh Actor instance from DI framework ...
}

val actorRef = system.actorOf(
  Props(classOf[DependencyInjector], applicationContext, "hello"),
  "helloBean")
```

Warning: You might be tempted at times to offer an `IndirectActorProducer` which always returns the same instance, e.g. by using a `lazy val`. This is not supported, as it goes against the meaning of an actor restart, which is described here: *What Restarting Means*.
When using a dependency injection framework, actor beans *MUST NOT* have singleton scope.

Techniques for dependency injection and integration with dependency injection frameworks are described in more depth in the Using Akka with Dependency Injection guideline and the Akka Java Spring tutorial in Typesafe Activator.

The Inbox

When writing code outside of actors which shall communicate with actors, the `ask` pattern can be a solution (see below), but there are two thing it cannot do: receiving multiple replies (e.g. by subscribing an `ActorRef` to a notification service) and watching other actors' lifecycle. For these purposes there is the `Inbox` class:

```
implicit val i = inbox()
echo ! "hello"
i.receive() should be("hello")
```

There is an implicit conversion from inbox to actor reference which means that in this example the sender reference will be that of the actor hidden away within the inbox. This allows the reply to be received on the last line. Watching an actor is quite simple as well:

```
val target = // some actor
val i = inbox()
i watch target
```

3.1.2 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 (set configuration item `akka.actor.debug.unhandled` to `on` to have them converted into actual Debug messages).

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

  This strategy is typically declared inside the actor in order to have access to the actor's internal state within the decider function: since failure is communicated as a message sent to the supervisor and processed like other messages (albeit outside of the normal behavior), all values and variables within the actor are available, as is the `sender` reference (which will be the immediate child reporting the failure; if the original failure occurred within a distant descendant it is still reported one level up at a time).

- `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.`

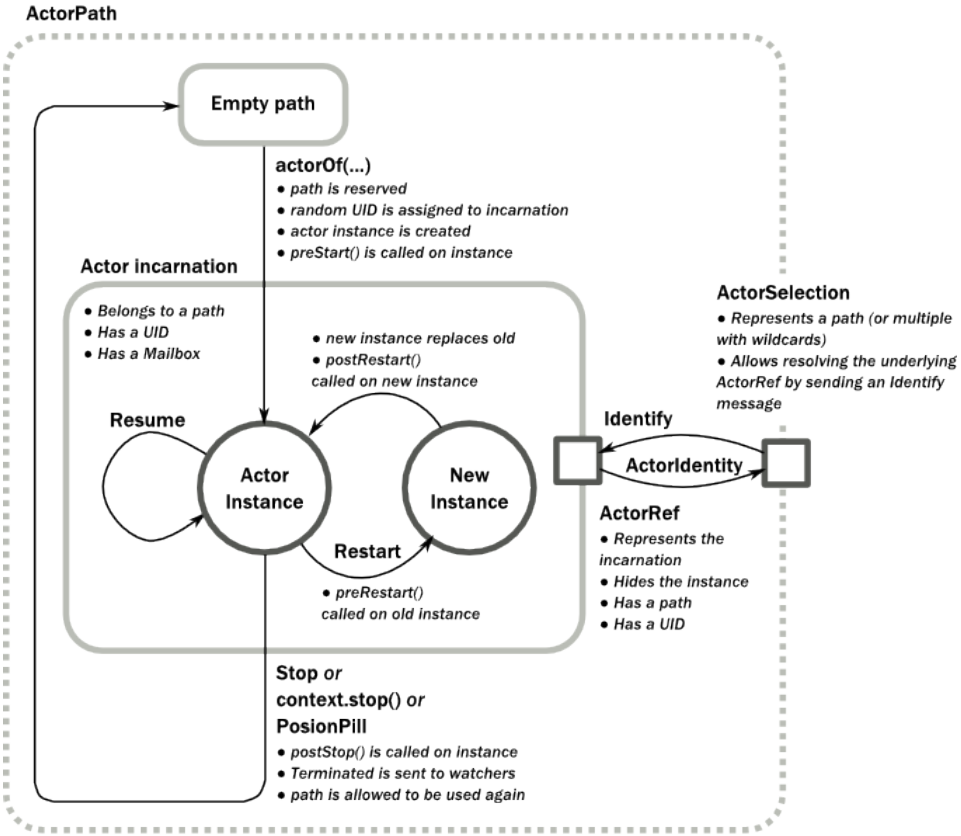
```
class FirstActor extends Actor {
  import context._
  val myActor = actorOf(Props[MyActor], name = "myactor")
  def receive = {
    case x => myActor ! x
  }
}
```

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

```
def preStart(): Unit = ()

def postStop(): Unit = ()

def preRestart(reason: Throwable, message: Option[Any]): Unit = {
  context.children foreach { child ⇒
    context.unwatch(child)
    context.stop(child)
  }
  postStop()
}

def postRestart(reason: Throwable): Unit = {
  preStart()
}
```

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

Actor Lifecycle



A path in an actor system represents a "place" which might be occupied by a living actor. Initially (apart from system initialized actors) a path is empty. When `actorOf()` is called it assigns an *incarnation* of the actor described by the passed `Props` to the given path. An actor incarnation is identified by the path *and a UID*. A restart only swaps the `Actor` instance defined by the `Props` but the incarnation and hence the UID remains the same.

The lifecycle of an incarnation ends when the actor is stopped. At that point the appropriate lifecycle events are called and watching actors are notified of the termination. After the incarnation is stopped, the path can be reused again by creating an actor with `actorOf()`. In this case the name of the new incarnation will be the same as the previous one but the UIDs will differ.

An `ActorRef` always represents an incarnation (path and UID) not just a given path. Therefore if an actor is stopped and a new one with the same name is created an `ActorRef` of the old incarnation will not point to the new one.

`ActorSelection` on the other hand points to the path (or multiple paths if wildcards are used) and is completely oblivious to which incarnation is currently occupying it. `ActorSelection` cannot be watched for this reason. It is possible to resolve the current incarnation's `ActorRef` living under the path by sending an `Identify` message to the `ActorSelection` which will be replied to with an `ActorIdentity` containing the correct reference (see *Identifying Actors via Actor Selection*). This can also be done with the `resolveOne` method of the `ActorSelection`, which returns a `Future` of the matching `ActorRef`.

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. In particular, the watching actor will receive a `Terminated` message even if the watched actor has already been terminated at the time of registration.

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)`. This works even if the `Terminated` message has already been enqueued in the mailbox; after calling `unwatch` no `Terminated` message for that actor will be processed anymore.

Start Hook

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

```
override def preStart() {
  child = context.actorOf(Props[MyActor], "child")
}
```

This method is called when the actor is first created. During restarts it is called by the default implementation of `postRestart`, which means that by overriding that method you can choose whether the initialization code in this method is called only exactly once for this actor or for every restart. Initialization code which is part of the actor's constructor will always be called when an instance of the actor class is created, which happens at every restart.

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors may be restarted in case an exception is thrown while processing a message (see *Supervision and Monitoring*). 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, or if an actor is restarted due to a sibling's failure. If the message is available, then that message's sender is also accessible in the usual way (i.e. by calling `sender`).

   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.

Warning: Be aware that the ordering of failure notifications relative to user messages is not deterministic. In particular, a parent might restart its child before it has processed the last messages sent by the child before the failure. See *Discussion: Message Ordering* for details.

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`.

3.1.3 Identifying Actors via Actor Selection

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 `ActorSelection` with the result:

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

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.

The path elements of an actor selection may contain wildcard patterns allowing for broadcasting of messages to that section:

```
// will look all children to serviceB with names starting with worker
context.actorSelection("/user/serviceB/worker*")
// will look up all siblings beneath same supervisor
context.actorSelection("../*")
```

Messages can be sent via the `ActorSelection` and the path of the `ActorSelection` is looked up when delivering each message. If the selection does not match any actors the message will be dropped.

To acquire an `ActorRef` for an `ActorSelection` you need to send a message to the selection and use the `sender()` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `ActorRef`. This message is handled specially by the actors which are traversed in the sense that if a concrete name lookup fails (i.e. a non-wildcard path element does not correspond to a live actor) then a negative result is generated. Please note that this does not mean that delivery of that reply is guaranteed, it still is a normal message.

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

class Follower extends Actor {
  val identifyId = 1
```

```
  context.actorSelection("/user/another") ! Identify(identifyId)

  def receive = {
    case ActorIdentity(`identifyId`, Some(ref)) =>
      context.watch(ref)
      context.become(active(ref))
    case ActorIdentity(`identifyId`, None) => context.stop(self)

  }

  def active(another: ActorRef): Actor.Receive = {
    case Terminated(`another`) => context.stop(self)
  }
}
```

You can also acquire an `ActorRef` for an `ActorSelection` with the `resolveOne` method of the `ActorSelection`. It returns a `Future` of the matching `ActorRef` if such an actor exists. It is completed with failure [[akka.actor.ActorNotFound]] if no such actor exists or the identification didn't complete within the supplied *timeout*.

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

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

An example demonstrating actor look-up is given in *Remoting Sample*.

Note: `actorFor` is deprecated in favor of `actorSelection` because actor references acquired with `actorFor` behaves different for local and remote actors. In the case of a local actor reference, the named actor needs to exist before the lookup, or else the acquired reference will be an `EmptyLocalActorRef`. This will be true even if an actor with that exact path is created after acquiring the actor reference. For remote actor references acquired with *actorFor* the behaviour is different and sending messages to such a reference will under the hood look up the actor by path on the remote system for every message send.

3.1.4 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)
```

3.1.5 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.

```
actorRef ! message
```

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 method. 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 }
import system.dispatcher // The ExecutionContext that will be used
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[String] // call by implicit conversion
    d <- (actorC ? Request).mapTo[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 affect 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.

Warning: 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 scala.concurrent.duration._
import akka.pattern.ask
val future = myActor.ask("hello")(5 seconds)
```

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

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

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

The `onComplete`, `onSuccess`, or `onFailure` 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.

```
target forward message
```

3.1.6 Receive messages

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

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

def receive: Actor.Receive
```

This method returns 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")
  }
}
```

3.1.7 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
```

3.1.8 Receive timeout

The *ActorContext* `setReceiveTimeout` defines the inactivity timeout after which the sending of a *ReceiveTimeout* message is triggered. When specified, the receive function should be able to handle an *akka.actor.ReceiveTimeout* message. 1 millisecond is the minimum supported timeout.

Please note that the receive timeout might fire and enqueue the *ReceiveTimeout* message right after another message was enqueued; hence it is **not guaranteed** that upon reception of the receive timeout there must have been an idle period beforehand as configured via this method.

Once set, the receive timeout stays in effect (i.e. continues firing repeatedly after inactivity periods). Pass in *Duration.Undefined* to switch off this feature.

```
import akka.actor.ReceiveTimeout
import scala.concurrent.duration._
class MyActor extends Actor {
  // To set an initial delay
  context.setReceiveTimeout(30 milliseconds)
  def receive = {
    case "Hello" =>
      // To set in a response to a message
      context.setReceiveTimeout(100 milliseconds)
    case ReceiveTimeout =>
      // To turn it off
      context.setReceiveTimeout(Duration.Undefined)
      throw new RuntimeException("Receive timed out")
  }
}
```

3.1.9 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 internal termination notifications 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() {
  // clean up some resources ...
}
```

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 scala.concurrent.Await

try {
  val stopped: Future[Boolean] = gracefulStop(actorRef, 5 seconds, Manager.Shutdown)
  Await.result(stopped, 6 seconds)
  // the actor has been stopped
} catch {
  // the actor wasn't stopped within 5 seconds
  case e: akka.pattern.AskTimeoutException =>
}
```

```
object Manager {
  case object Shutdown
}

class Manager extends Actor {
  import Manager._
  val worker = context.watch(context.actorOf(Props[Cruncher], "worker"))

  def receive = {
    case "job" => worker ! "crunch"
    case Shutdown =>
      worker ! PoisonPill
      context become shuttingDown
  }

  def shuttingDown: Receive = {
    case "job" => sender() ! "service unavailable, shutting down"
    case Terminated(`worker`) =>
      context stop self
  }
}
```

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()`.

In the above example a custom `Manager.Shutdown` message is sent to the target actor to initiate the process of stopping the actor. You can use `PoisonPill` for this, but then you have limited possibilities to perform interactions with other actors before stopping the target actor. Simple cleanup tasks can be handled in `postStop`.

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.

3.1.10 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)
  }
}
```

This variant of the `become` method is useful for many different things, such as to implement a Finite State Machine (FSM, for an example see Dining Hakkers). It will replace the current behavior (i.e. the top of the behavior stack), which means that you do not use `unbecome`, instead always the next behavior is explicitly installed.

The other way of using `become` does not replace but add to the top of the behavior stack. In this case care must be taken to ensure that the number of "pop" operations (i.e. `unbecome`) matches the number of "push" ones in the long run, otherwise this amounts to a memory leak (which is why this behavior is not the default).

```
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)
      }, discardOld = false) // push on top instead of replace
  }
}
```

```
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.

3.1.11 Stash

The *Stash* trait enables an actor to temporarily stash away messages that can not or should not be handled using the actor's current behavior. Upon changing the actor's message handler, i.e., right before invoking `context.become` or `context.unbecome`, all stashed messages can be "unstashed", thereby prepending them to the actor's mailbox. This way, the stashed messages can be processed in the same order as they have been received originally.

Note: The trait `Stash` extends the marker trait `RequiresMessageQueue[DequeBasedMessageQueueSemantics]` which requests the system to automatically choose a deque based mailbox implementation for the actor. If you want more control over the mailbox, see the documentation on mailboxes: *Mailboxes*.

Here is an example of the `Stash` in action:

```
import akka.actor.Stash
class ActorWithProtocol extends Actor with Stash {
  def receive = {
    case "open" =>
      unstashAll()
      context.become({
        case "write" => // do writing...
        case "close" =>
          unstashAll()
          context.unbecome()
        case msg => stash()
      }, discardOld = false) // stack on top instead of replacing
    case msg => stash()
  }
}
```

Invoking `stash()` adds the current message (the message that the actor received last) to the actor's stash. It is typically invoked when handling the default case in the actor's message handler to stash messages that aren't handled by the other cases. It is illegal to stash the same message twice; to do so results in an `IllegalStateException` being thrown. The stash may also be bounded in which case invoking `stash()` may lead to a capacity violation, which results in a `StashOverflowException`. The capacity of the stash can be configured using the `stash-capacity` setting (an `Int`) of the mailbox's configuration.

Invoking `unstashAll()` enqueues messages from the stash to the actor's mailbox until the capacity of the mailbox (if any) has been reached (note that messages from the stash are prepended to the mailbox). In case a bounded mailbox overflows, a `MessageQueueAppendFailedException` is thrown. The stash is guaranteed to be empty after calling `unstashAll()`.

The stash is backed by a `scala.collection.immutable.Vector`. As a result, even a very large number of messages may be stashed without a major impact on performance.

Warning: Note that the `Stash` trait must be mixed into (a subclass of) the `Actor` trait before any trait/class that overrides the `preRestart` callback. This means it's not possible to write `Actor with MyActor with Stash` if `MyActor` overrides `preRestart`.

Note that the stash is part of the ephemeral actor state, unlike the mailbox. Therefore, it should be managed like other parts of the actor's state which have the same property. The `Stash` trait's implementation of `preRestart` will call `unstashAll()`, which is usually the desired behavior.

Note: If you want to enforce that your actor can only work with an unbounded stash, then you should use the `UnboundedStash` trait instead.

3.1.12 Killing an Actor

You can kill an actor by sending a `Kill` message. This will cause the actor to throw a `ActorKilledException`, triggering a failure. The actor will suspend operation and its supervisor will be asked how to handle the failure, which may mean resuming the actor, restarting it or terminating it completely. See *What Supervision Means* for more information.

Use `Kill` like this:

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

3.1.13 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 (i.e. taken out of its mailbox and handed over to the current behavior), 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). Another possibility would be to have a look at the *PeekMailbox pattern*.

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 code within an actor throws an exception, that actor is suspended and the supervision process is started (see *Supervision and Monitoring*). Depending on the supervisor's decision the actor is resumed (as if nothing happened), restarted (wiping out its internal state and starting from scratch) or terminated.

3.1.14 Extending Actors using PartialFunction chaining

Sometimes it can be useful to share common behavior among a few actors, or compose one actor's behavior from multiple smaller functions. This is possible because an actor's `receive` method returns an `Actor.Receive`,

which is a type alias for `PartialFunction[Any,Unit]`, and partial functions can be chained together using the `PartialFunction#orElse` method. You can chain as many functions as you need, however you should keep in mind that "first match" wins - which may be important when combining functions that both can handle the same type of message.

For example, imagine you have a set of actors which are either `Producers` or `Consumers`, yet sometimes it makes sense to have an actor share both behaviors. This can be easily achieved without having to duplicate code by extracting the behaviors to traits and implementing the actor's `receive` as combination of these partial functions.

```
trait ProducerBehavior {
  this: Actor =>

  val producerBehavior: Receive = {
    case GiveMeThings =>
      sender() ! Give("thing")
  }
}

trait ConsumerBehavior {
  this: Actor with ActorLogging =>

  val consumerBehavior: Receive = {
    case ref: ActorRef =>
      ref ! GiveMeThings

    case Give(thing) =>
      log.info("Got a thing! It's {}", thing)
  }
}

class Producer extends Actor with ProducerBehavior {
  def receive = producerBehavior
}

class Consumer extends Actor with ActorLogging with ConsumerBehavior {
  def receive = consumerBehavior
}

class ProducerConsumer extends Actor with ActorLogging
  with ProducerBehavior with ConsumerBehavior {

  def receive = producerBehavior orElse consumerBehavior
}

// protocol
case object GiveMeThings
case class Give(thing: Any)
```

Instead of inheritance the same pattern can be applied via composition - one would simply compose the receive method using partial functions from delegates.

3.1.15 Initialization patterns

The rich lifecycle hooks of Actors provide a useful toolkit to implement various initialization patterns. During the lifetime of an `ActorRef`, an actor can potentially go through several restarts, where the old instance is replaced by a fresh one, invisibly to the outside observer who only sees the `ActorRef`.

One may think about the new instances as "incarnations". Initialization might be necessary for every incarnation of an actor, but sometimes one needs initialization to happen only at the birth of the first instance when the `ActorRef` is created. The following sections provide patterns for different initialization needs.

Initialization via constructor

Using the constructor for initialization has various benefits. First of all, it makes it possible to use `val` fields to store any state that does not change during the life of the actor instance, making the implementation of the actor more robust. The constructor is invoked for every incarnation of the actor, therefore the internals of the actor can always assume that proper initialization happened. This is also the drawback of this approach, as there are cases when one would like to avoid reinitializing internals on restart. For example, it is often useful to preserve child actors across restarts. The following section provides a pattern for this case.

Initialization via preStart

The method `preStart()` of an actor is only called once directly during the initialization of the first instance, that is, at creation of its `ActorRef`. In the case of restarts, `preStart()` is called from `postRestart()`, therefore if not overridden, `preStart()` is called on every incarnation. However, overriding `postRestart()` one can disable this behavior, and ensure that there is only one call to `preStart()`.

One useful usage of this pattern is to disable creation of new `ActorRefs` for children during restarts. This can be achieved by overriding `preRestart()`:

```
override def preStart(): Unit = {
  // Initialize children here
}

// Overriding postRestart to disable the call to preStart()
// after restarts
override def postRestart(reason: Throwable): Unit = ()

// The default implementation of preRestart() stops all the children
// of the actor. To opt-out from stopping the children, we
// have to override preRestart()
override def preRestart(reason: Throwable, message: Option[Any]): Unit = {
  // Keep the call to postStop(), but no stopping of children
  postStop()
}
```

Please note, that the child actors are *still restarted*, but no new `ActorRef` is created. One can recursively apply the same principles for the children, ensuring that their `preStart()` method is called only at the creation of their refs.

For more information see *What Restarting Means*.

Initialization via message passing

There are cases when it is impossible to pass all the information needed for actor initialization in the constructor, for example in the presence of circular dependencies. In this case the actor should listen for an initialization message, and use `become()` or a finite state-machine state transition to encode the initialized and uninitialized states of the actor.

```
var initializeMe: Option[String] = None

override def receive = {
  case "init" =>
    initializeMe = Some("Up and running")
    context.become(initialized, discardOld = true)

}

def initialized: Receive = {
  case "U OK?" => initializeMe foreach { sender() ! _ }
}
```

If the actor may receive messages before it has been initialized, a useful tool can be the `Stash` to save messages until the initialization finishes, and replaying them after the actor became initialized.

Warning: This pattern should be used with care, and applied only when none of the patterns above are applicable. One of the potential issues is that messages might be lost when sent to remote actors. Also, publishing an `ActorRef` in an uninitialized state might lead to the condition that it receives a user message before the initialization has been done.

3.2 Typed Actors

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 cannot 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.

3.2.1 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.

3.2.2 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.

3.2.3 Creating Typed Actors

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

Our example interface:

```
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

  @throws(classOf[Exception]) //declare it or you will get an UndeclaredThrowableException
  def squareTry(i: Int): Int //blocking send-request-reply with possible exception
}
```

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

```
class SquarerImpl(val name: String) extends Squarer {

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

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

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

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

  def squareTry(i: Int): Int = throw new Exception("Catch me!")
}
```

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.

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.

3.2.4 Method dispatch semantics

Methods returning:

- `Unit` will be dispatched with `fire-and-forget` semantics, exactly like `ActorRef.tell`
- `scala.concurrent.Future[_]` will use `send-request-reply` semantics, exactly like `ActorRef.ask`
- `scala.Option[_]` will use `send-request-reply` semantics, but *will* block to wait for an answer, and return `scala.None` if no answer was produced within the timeout, or `scala.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.

3.2.5 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.

3.2.6 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.

3.2.7 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:

```
//Inside your Typed Actor
val childSquarer: Squarer =
  TypedActor(TypedActor.context).typedActorOf(TypedProps[SquarerImpl]())
//Use "childSquarer" as a Squarer
```

You can also create a child Typed Actor in regular Akka Actors by giving the `ActorContext` as an input parameter to TypedActor.get(...).

3.2.8 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*.

3.2.9 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.

3.2.10 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.

3.2.11 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 pass the `ActorRef` to `typedActorOf`.

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

3.2.12 Lookup & Remoting

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

```
val typedActor: Foo with Bar =
  TypedActor(system).
    typedActorOf(
      TypedProps[FooBar],
      actorRefToRemoteActor)
//Use "typedActor" as a FooBar
```

3.2.13 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 {
  def doBar(str: String): Future[String] =
    Future.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)
```

3.3 Fault Tolerance

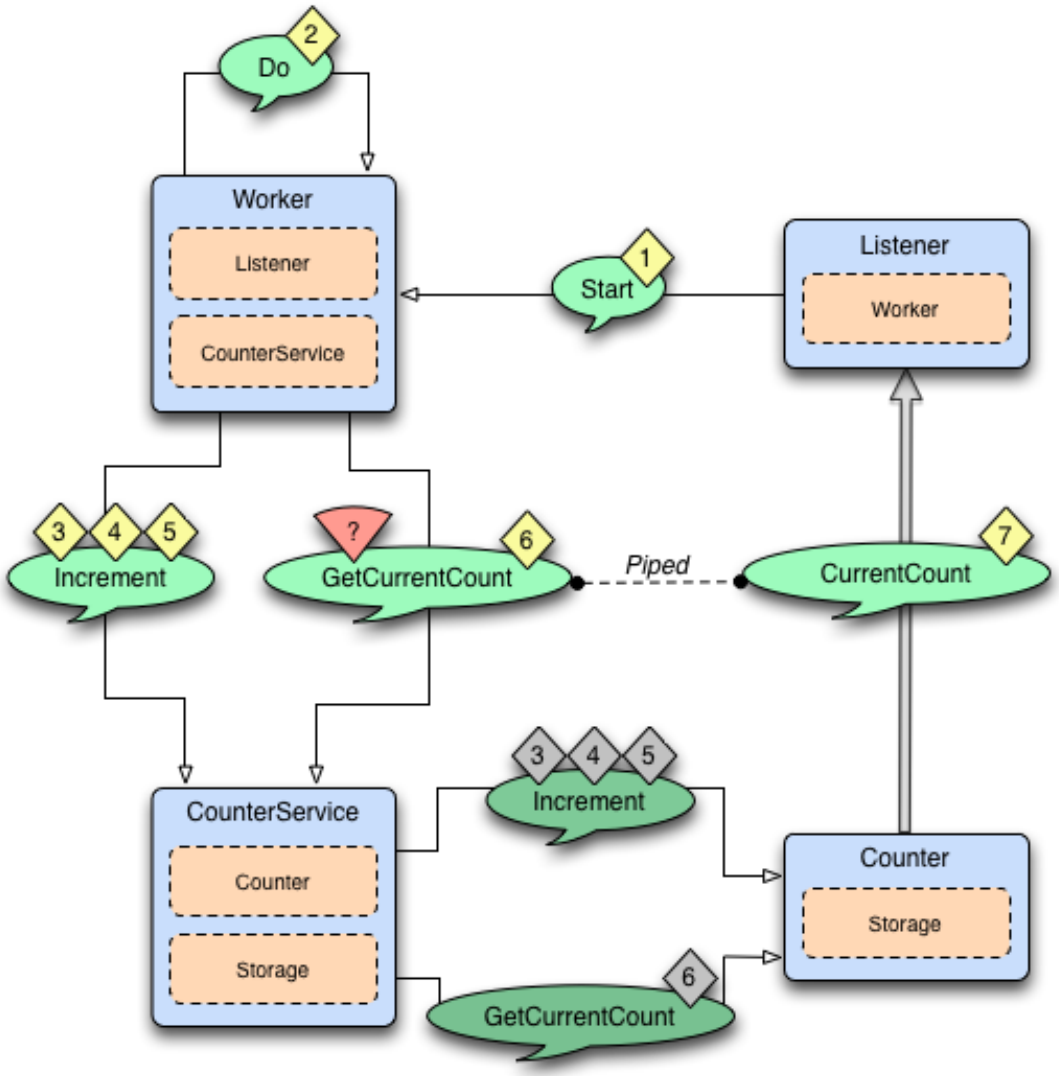
As explained in *Actor Systems* each actor is the supervisor of its children, and as such each actor defines fault handling supervisor strategy. This strategy cannot be changed afterwards as it is an integral part of the actor system's structure.

3.3.1 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.

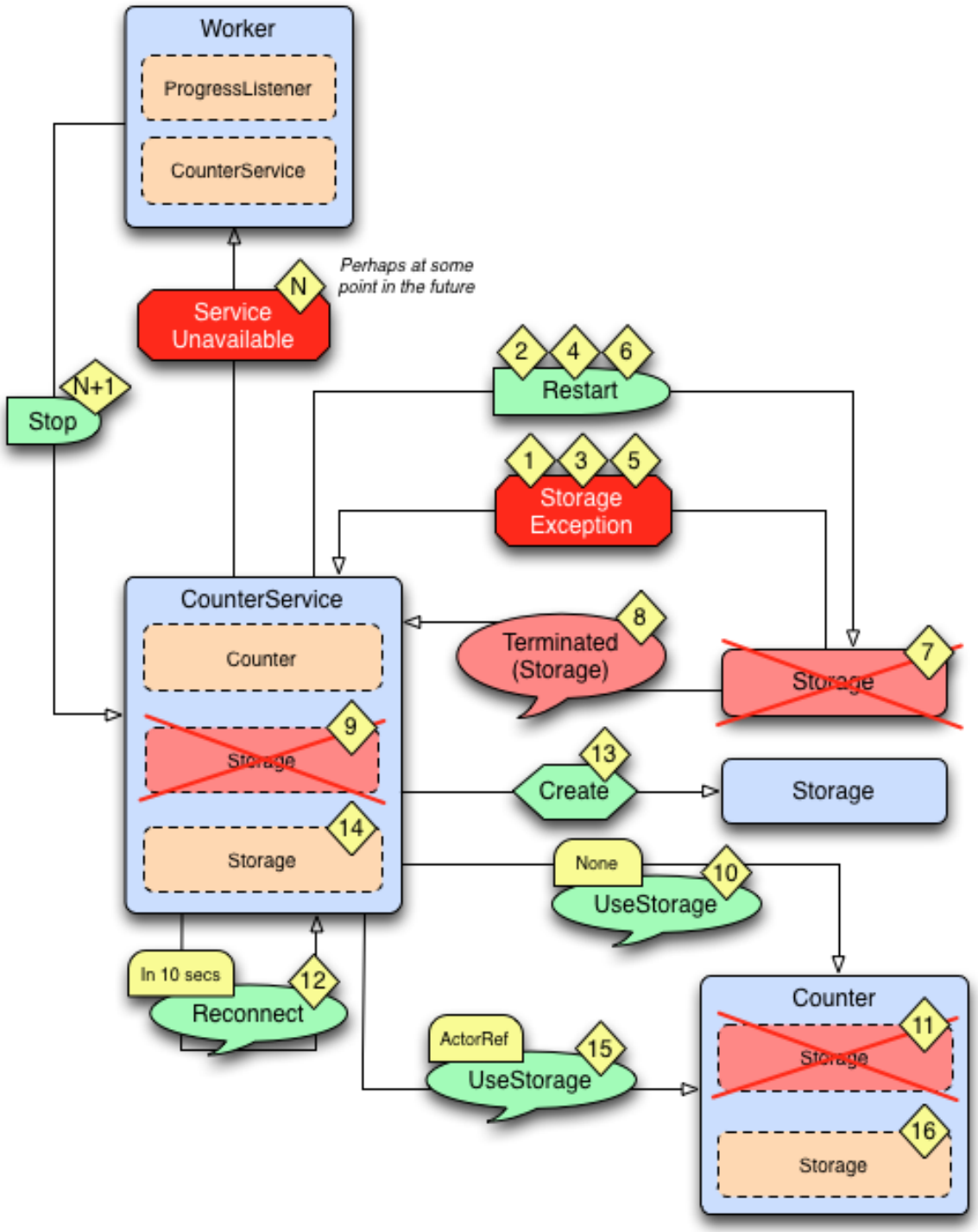
Diagrams of the Fault Tolerance Sample



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 `Counter` to use the new `Storage`

Full Source Code of the Fault Tolerance Sample

```
import akka.actor._
import akka.actor.SupervisorStrategy._
import scala.concurrent.duration._
import akka.util.Timeout
import akka.event.LoggingReceive
import akka.pattern.{ ask, pipe }
import com.typesafe.config.ConfigFactory

/**
 * 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()
  }
}

object Worker {
  case object Start
  case object Do
  case class Progress(percent: Double)
}

/**
 * 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
  import context.dispatcher // Use this Actors' Dispatcher as ExecutionContext

  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
  }
}

object CounterService {
  case class Increment(n: Int)
  case object GetCurrentCount
```

```
  case class CurrentCount(key: String, count: Long)
  class ServiceUnavailable(msg: String) extends RuntimeException(msg)

  private case object Reconnect
}

/**
 * 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 seconds) {
      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

  import context.dispatcher // Use this Actors' Dispatcher as ExecutionContext

  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(classOf[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 :+= (sender() -> msg)
    }
  }

}

object Counter {
  case class UseStorage(storage: Option[ActorRef])
}

/**
 * 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)) }
  }

}

object Storage {
  case class Store(entry: Entry)
  case class Get(key: String)
  case class Entry(key: String, value: Long)
  class StorageException(msg: String) extends RuntimeException(msg)
}

/**
 * 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))
  }
}

object DummyDB {
  import Storage.StorageException
  private var db = Map[String, Long]()

  @throws(classOf[StorageException])
  def save(key: String, value: Long): Unit = synchronized {
    if (11 <= value && value <= 14)
      throw new StorageException("Simulated store failure " + value)
    db += (key -> value)
  }

  @throws(classOf[StorageException])
  def load(key: String): Option[Long] = synchronized {
    db.get(key)
  }
}
```

3.3.2 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 scala.concurrent.duration._

override val supervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
    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 child actor is stopped if the limit is exceeded.

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.

Note: If the strategy is declared inside the supervising actor (as opposed to within a companion object) its decider has access to all internal state of the actor in a thread-safe fashion, including obtaining a reference to the currently failed child (available as the `sender` of the failure message).

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.

You can combine your own strategy with the default strategy:

```
import akka.actor.OneForOneStrategy
import akka.actor.SupervisorStrategy._
import scala.concurrent.duration._

override val supervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
    case _: ArithmeticException => Resume
    case t =>
      super.supervisorStrategy.decider.applyOrElse(t, (_: Any) => Escalate)
  }
```

Stopping Supervisor Strategy

Closer to the Erlang way is the strategy to just stop children when they fail and then take corrective action in the supervisor when DeathWatch signals the loss of the child. This strategy is also provided pre-packaged as `SupervisorStrategy.stoppingStrategy` with an accompanying `StoppingSupervisorStrategy` configurator to be used when you want the `"/user"` guardian to apply it.

Logging of Actor Failures

By default the `SupervisorStrategy` logs failures unless they are escalated. Escalated failures are supposed to be handled, and potentially logged, at a level higher in the hierarchy.

You can mute the default logging of a `SupervisorStrategy` by setting `loggingEnabled` to `false` when instantiating it. Customized logging can be done inside the `Decider`. Note that the reference to the currently failed child is available as the `sender` when the `SupervisorStrategy` is declared inside the supervising actor.

You may also customize the logging in your own `SupervisorStrategy` implementation by overriding the `logFailure` method.

3.3.3 Supervision of Top-Level Actors

Toplevel actors means those which are created using `system.actorOf()`, and they are children of the *User Guardian*. There are no special rules applied in this case, the guardian simply applies the configured strategy.

3.3.4 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 scala.concurrent.duration._

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

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

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

class FaultHandlingDocSpec extends AkkaSpec with ImplicitSender {

  import FaultHandlingDocSpec._

  "A supervisor" must {

    "apply the chosen strategy for its child" in {
```

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

      supervisor ! Props[Child]
      val child = expectMsgType[ActorRef] // retrieve answer from TestKit's testActor
      EventFilter.warning(occurrences = 1) intercept {
        child ! 42 // set state to 42
        child ! "get"
        expectMsg(42)

        child ! new ArithmeticException // crash it
        child ! "get"
        expectMsg(42)
      }
      EventFilter[NullPointerException](occurrences = 1) intercept {
        child ! new NullPointerException // crash it harder
        child ! "get"
        expectMsg(0)
      }
      EventFilter[IllegalArgumentException](occurrences = 1) intercept {
        watch(child) // have testActor watch "child"
        child ! new IllegalArgumentException // break it
        expectMsgPF() { case Terminated(`child`) => () }
      }
      EventFilter[Exception]("CRASH", occurrences = 2) intercept {
        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
        expectMsgPF() {
          case t @ Terminated(`child2`) if t.existenceConfirmed => ()
        }
        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)
      }
      // code here
    }
  }
}
```

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*, 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
expectMsgPF() { case 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.

```
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
expectMsgPF() {
  case t @ Terminated(`child2`) if t.existenceConfirmed => ()
}
```

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 scala.concurrent.duration._

  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
      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)
```

3.4 Dispatchers

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*.

3.4.1 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 the specified `default-executor`. If an ActorSystem is created with an ExecutionContext passed in, this ExecutionContext will be used as the default executor for all dispatchers in this ActorSystem. If no ExecutionContext is given, it will fallback to the executor specified in `akka.actor.default-dispatcher.default-executor.fallback`. By default this is a "fork-join-executor", which gives excellent performance in most cases.

3.4.2 Looking up a Dispatcher

Dispatchers implement the `ExecutionContext` interface and can thus be used to run `Future` invocations etc.

```
// for use with Futures, Scheduler, etc.
implicit val executionContext = system.dispatchers.lookup("my-dispatcher")
```

3.4.3 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 to configure the dispatcher:

```
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*.

Then you create the actor as usual and define the dispatcher in the deployment configuration.

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor], "myactor")
```

```
akka.actor.deployment {
  /myactor {
    dispatcher = my-dispatcher
  }
}
```

An alternative to the deployment configuration is to define the dispatcher in code. If you define the `dispatcher` in the deployment configuration then this value will be used instead of programmatically provided parameter.

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

Note: The dispatcher you specify in `withDispatcher` and the `dispatcher` property in the deployment configuration 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"`

3.4.4 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`, and that pool will have only one thread.

Note that it's not guaranteed that the *same* thread is used over time, since the core pool timeout is used for `PinnedDispatcher` to keep resource usage down in case of idle actors. To use the same thread all the time you need to add `thread-pool-executor.allow-core-timeout=off` to the configuration of the `PinnedDispatcher`.

3.5 Mailboxes

An Akka `Mailbox` holds the messages that are destined for an `Actor`. Normally each `Actor` has its own mailbox, but with for example a `BalancingPool` all routees will share a single mailbox instance.

3.5.1 Mailbox Selection

Requiring a Message Queue Type for an Actor

It is possible to require a certain type of message queue for a certain type of actor by having that actor extend the parameterized trait `RequiresMessageQueue`. Here is an example:

```
import akka.dispatch.RequiresMessageQueue
import akka.dispatch.BoundedMessageQueueSemantics

class MyBoundedActor extends MyActor
  with RequiresMessageQueue[BoundedMessageQueueSemantics]
```

The type parameter to the `RequiresMessageQueue` trait needs to be mapped to a mailbox in configuration like this:

```
bounded-mailbox {
  mailbox-type = "akka.dispatch.BoundedMailbox"
  mailbox-capacity = 1000
  mailbox-push-timeout-time = 10s
}

akka.actor.mailbox.requirements {
  "akka.dispatch.BoundedMessageQueueSemantics" = bounded-mailbox
}
```

Now every time you create an actor of type `MyBoundedActor` it will try to get a bounded mailbox. If the actor has a different mailbox configured in deployment, either directly or via a dispatcher with a specified mailbox type, then that will override this mapping.

Note: The type of the queue in the mailbox created for an actor will be checked against the required type in the trait and if the queue doesn't implement the required type then actor creation will fail.

Requiring a Message Queue Type for a Dispatcher

A dispatcher may also have a requirement for the mailbox type used by the actors running on it. An example is the BalancingDispatcher which requires a message queue that is thread-safe for multiple concurrent consumers. Such a requirement is formulated within the dispatcher configuration section like this:

```
my-dispatcher {
  mailbox-requirement = org.example.MyInterface
}
```

The given requirement names a class or interface which will then be ensured to be a supertype of the message queue's implementation. In case of a conflict—e.g. if the actor requires a mailbox type which does not satisfy this requirement—then actor creation will fail.

How the Mailbox Type is Selected

When an actor is created, the `ActorRefProvider` first determines the dispatcher which will execute it. Then the mailbox is determined as follows:

1. If the actor's deployment configuration section contains a `mailbox` key then that names a configuration section describing the mailbox type to be used.
2. If the actor's `Props` contains a mailbox selection—i.e. `withMailbox` was called on it—then that names a configuration section describing the mailbox type to be used.
3. If the dispatcher's configuration section contains a `mailbox-type` key the same section will be used to configure the mailbox type.
4. If the actor requires a mailbox type as described above then the mapping for that requirement will be used to determine the mailbox type to be used; if that fails then the dispatcher's requirement—if any—will be tried instead.
5. If the dispatcher requires a mailbox type as described above then the mapping for that requirement will be used to determine the mailbox type to be used.

6. The default mailbox `akka.actor.default-mailbox` will be used.

Default Mailbox

When the mailbox is not specified as described above the default mailbox is used. By default it is an unbounded mailbox, which is backed by a `java.util.concurrent.ConcurrentLinkedQueue`.

`SingleConsumerOnlyUnboundedMailbox` is an even more efficient mailbox, and it can be used as the default mailbox, but it cannot be used with a BalancingDispatcher.

Configuration of `SingleConsumerOnlyUnboundedMailbox` as default mailbox:

```
akka.actor.default-mailbox {
  mailbox-type = "akka.dispatch.SingleConsumerOnlyUnboundedMailbox"
}
```

Which Configuration is passed to the Mailbox Type

Each mailbox type is implemented by a class which extends `MailboxType` and takes two constructor arguments: a `ActorSystem.Settings` object and a `Config` section. The latter is computed by obtaining the named configuration section from the actor system's configuration, overriding its `id` key with the configuration path of the mailbox type and adding a fall-back to the default mailbox configuration section.

3.5.2 Builtin implementations

Akka comes shipped with a number of mailbox implementations:

- UnboundedMailbox - The default mailbox
  - Backed by a `java.util.concurrent.ConcurrentLinkedQueue`
  - Blocking: No
  - Bounded: No
  - Configuration name: "unbounded" or "akka.dispatch.UnboundedMailbox"
- SingleConsumerOnlyUnboundedMailbox
  - Backed by a very efficient Multiple Producer Single Consumer queue, cannot be used with BalancingDispatcher
  - Blocking: No
  - Bounded: No
  - Configuration name: "akka.dispatch.SingleConsumerOnlyUnboundedMailbox"
- BoundedMailbox
  - Backed by a `java.util.concurrent.LinkedBlockingQueue`
  - Blocking: Yes
  - Bounded: Yes
  - Configuration name: "bounded" or "akka.dispatch.BoundedMailbox"
- UnboundedPriorityMailbox
  - Backed by a `java.util.concurrent.PriorityBlockingQueue`
  - Blocking: Yes
  - Bounded: No
  - Configuration name: "akka.dispatch.UnboundedPriorityMailbox"

- BoundedPriorityMailbox
  - Backed by a `java.util.PriorityBlockingQueue` wrapped in an `akka.util.BoundedBlockingQueue`
  - Blocking: Yes
  - Bounded: Yes
  - Configuration name: "akka.dispatch.BoundedPriorityMailbox"

3.5.3 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 UnboundedPriorityMailbox(
    // 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 = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
  //Other dispatcher configuration goes here
}
```

And then an example on how you would use it:

```
// We create a new Actor that just prints out what it processes
class Logger extends 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)
  }
}
```

```
val a = system.actorOf(Props(classOf[Logger], this).withDispatcher(
  "prio-dispatcher"))

/*
 * Logs:
 * 'highpriority
 * 'highpriority
 * 'pigdog
 * 'pigdog2
 * 'pigdog3
 * 'lowpriority
 * 'lowpriority
 */
```

It is also possible to configure a mailbox type directly like this:

```
prio-mailbox {
  mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
  //Other mailbox configuration goes here
}

akka.actor.deployment {
  /priomailboxactor {
    mailbox = prio-mailbox
  }
}
```

And then use it either from deployment like this:

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor], "priomailboxactor")
```

Or code like this:

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor].withMailbox("prio-mailbox"))
```

3.5.4 Creating your own Mailbox type

An example is worth a thousand quacks:

```
import akka.actor.ActorRef
import akka.actor.ActorSystem
import akka.dispatch.Envelope
import akka.dispatch.MailboxType
import akka.dispatch.MessageQueue
import akka.dispatch.ProducesMessageQueue
import com.typesafe.config.Config
import java.util.concurrent.ConcurrentLinkedQueue
import scala.Option

// Marker trait used for mailbox requirements mapping
trait MyUnboundedMessageQueueSemantics

object MyUnboundedMailbox {
  // This is the MessageQueue implementation
  class MyMessageQueue extends MessageQueue
    with MyUnboundedMessageQueueSemantics {

    private final val queue = new ConcurrentLinkedQueue[Envelope]()

    // these should be implemented; queue used as example
```

```
    def enqueue(receiver: ActorRef, handle: Envelope): Unit =
      queue.offer(handle)
    def dequeue(): Envelope = queue.poll()
    def numberOfMessages: Int = queue.size
    def hasMessages: Boolean = !queue.isEmpty
    def cleanUp(owner: ActorRef, deadLetters: MessageQueue) {
      while (hasMessages) {
        deadLetters.enqueue(owner, dequeue())
      }
    }
  }
}

// This is the Mailbox implementation
class MyUnboundedMailbox extends MailboxType
  with ProducesMessageQueue[MyUnboundedMailbox.MyMessageQueue] {

  import MyUnboundedMailbox._

  // This constructor signature must exist, it will be called by Akka
  def this(settings: ActorSystem.Settings, config: Config) = {
    // put your initialization code here
    this()
  }

  // The create method is called to create the MessageQueue
  final override def create(owner: Option[ActorRef],
                            system: Option[ActorSystem]): MessageQueue =
    new MyMessageQueue()
}
```

And then you just specify the FQCN of your MailboxType as the value of the "mailbox-type" in the dispatcher configuration, or the mailbox 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 or mailbox setting using this mailbox type; the mailbox type will be instantiated once for each dispatcher or mailbox setting using it.

You can also use the mailbox as a requirement on the dispatcher like this:

```
custom-dispatcher {
  mailbox-requirement =
  "docs.dispatcher.MyUnboundedJMessageQueueSemantics"
}

akka.actor.mailbox.requirements {
  "docs.dispatcher.MyUnboundedJMessageQueueSemantics" =
  custom-dispatcher-mailbox
}

custom-dispatcher-mailbox {
  mailbox-type = "docs.dispatcher.MyUnboundedJMailbox"
}
```

Or by defining the requirement on your actor class like this:

```
class MySpecialActor extends Actor
  with RequiresMessageQueue[MyUnboundedMessageQueueSemantics] {
  // ...
}
```

3.5.5 Special Semantics of `system.actorOf`

In order to make `system.actorOf` both synchronous and non-blocking while keeping the return type `ActorRef` (and the semantics that the returned ref is fully functional), special handling takes place for this case. Behind the scenes, a hollow kind of actor reference is constructed, which is sent to the system's guardian actor who actually creates the actor and its context and puts those inside the reference. Until that has happened, messages sent to the `ActorRef` will be queued locally, and only upon swapping the real filling in will they be transferred into the real mailbox. Thus,

```
val props: Props = ...
// this actor uses MyCustomMailbox, which is assumed to be a singleton
system.actorOf(props.withDispatcher("myCustomMailbox")) ! "bang"
assert(MyCustomMailbox.instance.getLastEnqueuedMessage == "bang")
```

will probably fail; you will have to allow for some time to pass and retry the check à la `TestKit.awaitCond`.

3.6 Routing

Messages can be sent via a router to efficiently route them to destination actors, known as its *routees*. A `Router` can be used inside or outside of an actor, and you can manage the routees yourselves or use a self contained router actor with configuration capabilities.

Different routing strategies can be used, according to your application's needs. Akka comes with several useful routing strategies right out of the box. But, as you will see in this chapter, it is also possible to *create your own*.

3.6.1 A Simple Router

The following example illustrates how to use a `Router` and manage the routees from within an actor.

```
import akka.routing.ActorRefRoutee
import akka.routing.Router
import akka.routing.RoundRobinRoutingLogic

class Master extends Actor {
  var router = {
    val routees = Vector.fill(5) {
      val r = context.actorOf(Props[Worker])
      context watch r
      ActorRefRoutee(r)
    }
    Router(RoundRobinRoutingLogic(), routees)
  }

  def receive = {
    case w: Work =>
      router.route(w, sender())
    case Terminated(a) =>
      router = router.removeRoutee(a)
      val r = context.actorOf(Props[Worker])
      context watch r
      router = router.addRoutee(r)
  }
}
```

We create a `Router` and specify that it should use `RoundRobinRoutingLogic` when routing the messages to the routees.

The routing logic shipped with Akka are:

- `akka.routing.RoundRobinRoutingLogic`

- `akka.routing.RandomRoutingLogic`
- `akka.routing.SmallestMailboxRoutingLogic`
- `akka.routing.BroadcastRoutingLogic`
- `akka.routing.ScatterGatherFirstCompletedRoutingLogic`
- `akka.routing.ConsistentHashingRoutingLogic`

We create the routees as ordinary child actors wrapped in `ActorRefRoutee`. We watch the routees to be able to replace them if they are terminated.

Sending messages via the router is done with the `route` method, as is done for the `Work` messages in the example above.

The `Router` is immutable and the `RoutingLogic` is thread safe; meaning that they can also be used outside of actors.

Note: In general, any message sent to a router will be sent onwards to its routees, but there is one exception. The special *Broadcast Messages* will send to *all* of a router's routees

---

3.6.2 A Router Actor

A router can also be created as a self contained actor that manages the routees itself and loads routing logic and other settings from configuration.

This type of router actor comes in two distinct flavors:

- Pool - The router creates routees as child actors and removes them from the router if they terminate.
- Group - The routee actors are created externally to the router and the router sends messages to the specified path using actor selection, without watching for termination.

The settings for a router actor can be defined in configuration or programmatically. Although router actors can be defined in the configuration file, they must still be created programmatically, i.e. you cannot make a router through external configuration alone. If you define the router actor in the configuration file then these settings will be used instead of any programmatically provided parameters.

You send messages to the routees via the router actor in the same way as for ordinary actors, i.e. via its `ActorRef`. The router actor forwards messages onto its routees without changing the original sender. When a routee replies to a routed message, the reply will be sent to the original sender, not to the router actor.

Note: In general, any message sent to a router will be sent onwards to its routees, but there are a few exceptions. These are documented in the *Specially Handled Messages* section below.

Pool

The following code and configuration snippets show how to create a *round-robin* router that forwards messages to five `Worker` routees. The routees will be created as the router's children.

```
akka.actor.deployment {
  /parent/router1 {
    router = round-robin-pool
    nr-of-instances = 5
  }
}
```

```
val router1: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

Here is the same example, but with the router configuration provided programmatically instead of from configuration.

```
val router2: ActorRef =
  context.actorOf(RoundRobinPool(5).props(Props[Worker]), "router2")
```

Remote Deployed Routees

In addition to being able to create local actors as routees, you can instruct the router to deploy its created children on a set of remote hosts. Routees will be deployed in round-robin fashion. In order to deploy routees remotely, wrap the router configuration in a `RemoteRouterConfig`, attaching the remote addresses of the nodes to deploy to. Remote deployment requires the `akka-remote` module to be included in the classpath.

```
import akka.actor.{ Address, AddressFromURIString }
import akka.remote.routing.RemoteRouterConfig
val addresses = Seq(
  Address("akka.tcp", "remotesys", "otherhost", 1234),
  AddressFromURIString("akka.tcp://othersys@anotherhost:1234"))
val routerRemote = system.actorOf(
  RemoteRouterConfig(RoundRobinPool(5), addresses).props(Props[Echo]))
```

Senders

By default, when a routee sends a message, it will *implicitly set itself as the sender*.

```
sender() ! x // replies will go to this actor
```

However, it is often useful for routees to set the *router* as a sender. For example, you might want to set the router as the sender if you want to hide the details of the routees behind the router. The following code snippet shows how to set the parent router as sender.

```
sender().tell("reply", context.parent) // replies will go back to parent
sender().!("reply")(context.parent) // alternative syntax (beware of the parens!)
```

Supervision

Routees that are created by a pool router will be created as the router's children. The router is therefore also the children's supervisor.

The supervision strategy of the router actor can be configured with the `supervisorStrategy` property of the Pool. If no configuration is provided, routers default to a strategy of "always escalate". This means that errors are passed up to the router's supervisor for handling. The router's supervisor will decide what to do about any errors.

Note the router's supervisor will treat the error as an error with the router itself. Therefore a directive to stop or restart will cause the router *itself* to stop or restart. The router, in turn, will cause its children to stop and restart.

It should be mentioned that the router's restart behavior has been overridden so that a restart, while still re-creating the children, will still preserve the same number of actors in the pool.

This means that if you have not specified `supervisorStrategy` of the router or its parent a failure in a routee will escalate to the parent of the router, which will by default restart the router, which will restart all routees (it uses Escalate and does not stop routees during restart). The reason is to make the default behave such that adding `withRouter` to a child's definition does not change the supervision strategy applied to the child. This might be an inefficiency that you can avoid by specifying the strategy when defining the router.

Setting the strategy is easily done:

```
val escalator = OneForOneStrategy() {
  case e ⇒ testActor ! e; SupervisorStrategy.Escalate
}
```

```
val router = system.actorOf(RoundRobinPool(1, supervisorStrategy = escalator).props(
  routeeProps = Props[TestActor]))
```

Note: If the child of a pool router terminates, the pool router will not automatically spawn a new child. In the event that all children of a pool router have terminated the router will terminate itself unless it is a dynamic router, e.g. using a resizer.

Group

Sometimes, rather than having the router actor create its routees, it is desirable to create routees separately and provide them to the router for its use. You can do this by passing an paths of the routees to the router's configuration. Messages will be sent with `ActorSelection` to these paths.

The example below shows how to create a router by providing it with the path strings of three routee actors.

```
akka.actor.deployment {
  /parent/router3 {
    router = round-robin-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
val router3: ActorRef =
  context.actorOf(FromConfig.props(), "router3")
```

Here is the same example, but with the router configuration provided programmatically instead of from configuration.

```
val router4: ActorRef =
  context.actorOf(RoundRobinGroup(paths).props(), "router4")
```

The routee actors are created externally from the router:

```
system.actorOf(Props[Workers], "workers")
```

```
class Workers extends Actor {
  context.actorOf(Props[Worker], name = "w1")
  context.actorOf(Props[Worker], name = "w2")
  context.actorOf(Props[Worker], name = "w3")
  // ...
```

The paths may contain protocol and address information for actors running on remote hosts. Remoting requires the `akka-remote` module to be included in the classpath.

```
akka.actor.deployment {
  /parent/remoteGroup {
    router = round-robin-group
    routees.paths = [
      "akka.tcp://app@10.0.0.1:2552/user/workers/w1",
      "akka.tcp://app@10.0.0.2:2552/user/workers/w1",
      "akka.tcp://app@10.0.0.3:2552/user/workers/w1"]
  }
}
```

3.6.3 Router usage

In this section we will describe how to create the different types of router actors.

The router actors in this section are created from within a top level actor named `parent`. Note that deployment paths in the configuration starts with `/parent/` followed by the name of the router actor.

```
system.actorOf(Props[Parent], "parent")
```

RoundRobinPool and RoundRobinGroup

Routes in a round-robin fashion to its routees.

RoundRobinPool defined in configuration:

```
akka.actor.deployment {
  /parent/router1 {
    router = round-robin-pool
    nr-of-instances = 5
  }
}
```

```
val router1: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

RoundRobinPool defined in code:

```
val router2: ActorRef =
  context.actorOf(RoundRobinPool(5).props(Props[Worker]), "router2")
```

RoundRobinGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router3 {
    router = round-robin-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
val router3: ActorRef =
  context.actorOf(FromConfig.props(), "router3")
```

RoundRobinGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router4: ActorRef =
  context.actorOf(RoundRobinGroup(paths).props(), "router4")
```

RandomPool and RandomGroup

This router type selects one of its routees randomly for each message.

RandomPool defined in configuration:

```
akka.actor.deployment {
  /parent/router5 {
    router = random-pool
    nr-of-instances = 5
  }
}
```

```
val router5: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router5")
```

RandomPool defined in code:

```
val router6: ActorRef =
  context.actorOf(RandomPool(5).props(Props[Worker]), "router6")
```

RandomGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router7 {
    router = random-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
val router7: ActorRef =
  context.actorOf(FromConfig.props(), "router7")
```

RandomGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router8: ActorRef =
  context.actorOf(RandomGroup(paths).props(), "router8")
```

BalancingPool

A Router that will try to redistribute work from busy routees to idle routees. All routees share the same mailbox.

BalancingPool defined in configuration:

```
akka.actor.deployment {
  /parent/router9 {
    router = balancing-pool
    nr-of-instances = 5
  }
}
```

```
val router9: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router9")
```

BalancingPool defined in code:

```
val router10: ActorRef =
  context.actorOf(BalancingPool(5).props(Props[Worker]), "router10")
```

Addition configuration for the balancing dispatcher, which is used by the pool, can be configured in the `pool-dispatcher` section of the router deployment configuration.

```
akka.actor.deployment {
  /parent/router9b {
    router = balancing-pool
    nr-of-instances = 5
    pool-dispatcher {
      attempt-teamwork = off
    }
  }
}
```

There is no Group variant of the BalancingPool.

SmallestMailboxPool

A Router that tries to send to the non-suspended child 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

SmallestMailboxPool defined in configuration:

```
akka.actor.deployment {
  /parent/router11 {
    router = smallest-mailbox-pool
    nr-of-instances = 5
  }
}
```

```
val router11: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router11")
```

SmallestMailboxPool defined in code:

```
val router12: ActorRef =
  context.actorOf(SmallestMailboxPool(5).props(Props[Worker]), "router12")
```

There is no Group variant of the SmallestMailboxPool because the size of the mailbox and the internal dispatching state of the actor is not practically available from the paths of the routees.

BroadcastPool and BroadcastGroup

A broadcast router forwards the message it receives to *all* its routees.

BroadcastPool defined in configuration:

```
akka.actor.deployment {
  /parent/router13 {
    router = broadcast-pool
    nr-of-instances = 5
  }
}
```

```
val router13: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router13")
```

BroadcastPool defined in code:

```
val router14: ActorRef =
  context.actorOf(BroadcastPool(5).props(Props[Worker]), "router14")
```

BroadcastGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router15 {
    router = broadcast-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

```
val router15: ActorRef =
  context.actorOf(FromConfig.props(), "router15")
```

BroadcastGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router16: ActorRef =
  context.actorOf(BroadcastGroup(paths).props(), "router16")
```

Note: Broadcast routers always broadcast *every* message to their routees. If you do not want to broadcast every message, then you can use a non-broadcasting router and use *Broadcast Messages* as needed.

ScatterGatherFirstCompletedPool and ScatterGatherFirstCompletedGroup

The ScatterGatherFirstCompletedRouter will send the message on to all its routees. It then waits for first reply it gets back. This result will be sent back to original sender. Other replies are discarded.

It is expecting at least one reply within a configured duration, otherwise it will reply with `akka.pattern.AskTimeoutException` in a `akka.actor.Status.Failure`.

ScatterGatherFirstCompletedPool defined in configuration:

```
akka.actor.deployment {
  /parent/router17 {
    router = scatter-gather-pool
    nr-of-instances = 5
    within = 10 seconds
  }
}
```

```
val router17: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router17")
```

ScatterGatherFirstCompletedPool defined in code:

```
val router18: ActorRef =
  context.actorOf(ScatterGatherFirstCompletedPool(5, within = 10.seconds).
    props(Props[Worker]), "router18")
```

ScatterGatherFirstCompletedGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router19 {
    router = scatter-gather-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    within = 10 seconds
  }
}
```

```
val router19: ActorRef =
  context.actorOf(FromConfig.props(), "router19")
```

ScatterGatherFirstCompletedGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router20: ActorRef =
  context.actorOf(ScatterGatherFirstCompletedGroup(paths,
    within = 10.seconds).props(), "router20")
```

ConsistentHashingPool and ConsistentHashingGroup

The ConsistentHashingPool uses consistent hashing to select a routee based on the sent message. This article gives good insight into how consistent hashing is implemented.

There is 3 ways to define what data to use for the consistent hash key.

- You can define `hashMapping` of the router to map incoming messages to their consistent hash key. This makes the decision transparent for the sender.
- The messages may implement `akka.routing.ConsistentHashingRouter.ConsistentHashable`. The key is part of the message and it's convenient to define it together with the message definition.
- The messages can be be wrapped in a `akka.routing.ConsistentHashingRouter.ConsistentHashableEnve` to define what data to use for the consistent hash key. The sender knows the key to use.

These ways to define the consistent hash key can be use together and at the same time for one router. The `hashMapping` is tried first.

Code example:

```
import akka.actor.Actor
import akka.routing.ConsistentHashingRouter.ConsistentHashable

class Cache extends Actor {
  var cache = Map.empty[String, String]

  def receive = {
    case Entry(key, value) => cache += (key -> value)
    case Get(key)          => sender() ! cache.get(key)
    case Evict(key)        => cache -= key
  }
}

case class Evict(key: String)

case class Get(key: String) extends ConsistentHashable {
  override def consistentHashKey: Any = key
}

case class Entry(key: String, value: String)
```

```
import akka.actor.Props
import akka.routing.ConsistentHashingPool
import akka.routing.ConsistentHashingRouter.ConsistentHashMapping
import akka.routing.ConsistentHashingRouter.ConsistentHashableEnvelope

def hashMapping: ConsistentHashMapping = {
  case Evict(key) => key
}

val cache: ActorRef =
  context.actorOf(ConsistentHashingPool(10, hashMapping = hashMapping).
    props(Props[Cache]), name = "cache")

cache ! ConsistentHashableEnvelope(
  message = Entry("hello", "HELLO"), hashKey = "hello")
cache ! ConsistentHashableEnvelope(
  message = Entry("hi", "HI"), hashKey = "hi")

cache ! Get("hello")
expectMsg(Some("HELLO"))

cache ! Get("hi")
expectMsg(Some("HI"))

cache ! Evict("hi")
cache ! Get("hi")
expectMsg(None)
```

In the above example you see that the `Get` message implements `ConsistentHashable` itself, while the `Entry` message is wrapped in a `ConsistentHashableEnvelope`. The `Evict` message is handled by the `hashMapping` partial function.

ConsistentHashingPool defined in configuration:

```
akka.actor.deployment {
  /parent/router21 {
    router = consistent-hashing-pool
    nr-of-instances = 5
    virtual-nodes-factor = 10
  }
}
```

```
val router21: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router21")
```

ConsistentHashingPool defined in code:

```
val router22: ActorRef =
  context.actorOf(ConsistentHashingPool(5).props(Props[Worker]),
    "router22")
```

ConsistentHashingGroup defined in configuration:

```
akka.actor.deployment {
  /parent/router23 {
    router = consistent-hashing-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
    virtual-nodes-factor = 10
  }
}
```

```
val router23: ActorRef =
  context.actorOf(FromConfig.props(), "router23")
```

ConsistentHashingGroup defined in code:

```
val paths = List("/user/workers/w1", "/user/workers/w2", "/user/workers/w3")
val router24: ActorRef =
  context.actorOf(ConsistentHashingGroup(paths).props(), "router24")
```

`virtual-nodes-factor` is the number of virtual nodes per routee that is used in the consistent hash node ring to make the distribution more uniform.

3.6.4 Specially Handled Messages

Most messages sent to router actors will be forwarded according to the routers' routing logic. However there are a few types of messages that have special behavior.

Note that these special messages, except for the `Broadcast` message, are only handled by self contained router actors and not by the `akka.routing.Router` component described in *A Simple Router*.

Broadcast Messages

A `Broadcast` message can be used to send a message to *all* of a router's routees. When a router receives a `Broadcast` message, it will broadcast that message's *payload* to all routees, no matter how that router would normally route its messages.

The example below shows how you would use a `Broadcast` message to send a very important message to every routee of a router.

```
import akka.routing.Broadcast
router ! Broadcast("Watch out for Davy Jones' locker")
```

In this example the router receives the `Broadcast` message, extracts its payload (`"Watch out for Davy Jones' locker"`), and then sends the payload on to all of the router's routees. It is up to each each routee actor to handle the received payload message.

PoisonPill Messages

A `PoisonPill` message has special handling for all actors, including for routers. When any actor receives a `PoisonPill` message, that actor will be stopped. See the *PoisonPill* documentation for details.

```
import akka.actor.PoisonPill
router ! PoisonPill
```

For a router, which normally passes on messages to routees, it is important to realise that `PoisonPill` messages are processed by the router only. `PoisonPill` messages sent to a router will *not* be sent on to routees.

However, a `PoisonPill` message sent to a router may still affect its routees, because it will stop the router and when the router stops it also stops its children. Stopping children is normal actor behavior. The router will stop routees that it has created as children. Each child will process its current message and then stop. This may lead to some messages being unprocessed. See the documentation on *Stopping actors* for more information.

If you wish to stop a router and its routees, but you would like the routees to first process all the messages currently in their mailboxes, then you should not send a `PoisonPill` message to the router. Instead you should wrap a `PoisonPill` message inside a `Broadcast` message so that each routee will receive the `PoisonPill` message. Note that this will stop all routees, even if the routees aren't children of the router, i.e. even routees programmatically provided to the router.

```
import akka.actor.PoisonPill
import akka.routing.Broadcast
router ! Broadcast(PoisonPill)
```

With the code shown above, each routee will receive a `PoisonPill` message. Each routee will continue to process its messages as normal, eventually processing the `PoisonPill`. This will cause the routee to stop. After all routees have stopped the router will itself be *stopped automatically* unless it is a dynamic router, e.g. using a resizer.

Note: Brendan W McAdams' excellent blog post Distributing Akka Workloads - And Shutting Down Afterwards discusses in more detail how `PoisonPill` messages can be used to shut down routers and routees.

Kill Messages

`Kill` messages are another type of message that has special handling. See *Killing an Actor* for general information about how actors handle `Kill` messages.

When a `Kill` message is sent to a router the router processes the message internally, and does *not* send it on to its routees. The router will throw an `ActorKilledException` and fail. It will then be either resumed, restarted or terminated, depending how it is supervised.

Routees that are children of the router will also be suspended, and will be affected by the supervision directive that is applied to the router. Routees that are not the routers children, i.e. those that were created externally to the router, will not be affected.

```
import akka.actor.Kill
router ! Kill
```

As with the `PoisonPill` message, there is a distinction between killing a router, which indirectly kills its children (who happen to be routees), and killing routees directly (some of whom may not be children.) To kill routees directly the router should be sent a `Kill` message wrapped in a `Broadcast` message.

```
import akka.actor.Kill
import akka.routing.Broadcast
router ! Broadcast(Kill)
```

Managagement Messages

- Sending `akka.routing.GetRoutees` to a router actor will make it send back its currently used routees in a `akka.routing.Routees` message.
- Sending `akka.routing.AddRoutee` to a router actor will add that routee to its collection of routees.

- Sending `akka.routing.RemoveRoutee` to a router actor will remove that routee to its collection of routees.
- Sending `akka.routing.AdjustPoolSize` to a pool router actor will add or remove that number of routees to its collection of routees.

These management messages may be handled after other messages, so if you send `AddRoutee` immediately followed an ordinary message you are not guaranteed that the routees have been changed when the ordinary message is routed. If you need to know when the change has been applied you can send `AddRoutee` followed by `GetRoutees` and when you receive the `Routees` reply you know that the preceeding change has been applied.

3.6.5 Dynamically Resizable Pool

Most pools can be used with a fixed number of routees or with a resize strategy to adjust the number of routees dynamically.

Pool with resizer defined in configuration:

```
akka.actor.deployment {
  /parent/router25 {
    router = round-robin-pool
    resizer {
      lower-bound = 2
      upper-bound = 15
      messages-per-resize = 100
    }
  }
}
```

```
val router25: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router25")
```

Several more configuration options are available and described in `akka.actor.deployment.default.resizer` section of the reference *Configuration*.

Pool with resizer defined in code:

```
val resizer = DefaultResizer(lowerBound = 2, upperBound = 15)
val router26: ActorRef =
  context.actorOf(RoundRobinPool(5, Some(resizer)).props(Props[Worker]),
    "router26")
```

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” `RouterActor` 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.

3.6.6 How Routing is Designed within Akka

On the surface routers look like normal actors, but they are actually implemented differently. Routers are designed to be extremely efficient at receiving messages and passing them quickly on to routees.

A normal actor can be used for routing messages, but an actor’s single-threaded processing can become a bottleneck. Routers can achieve much higher throughput with an optimization to the usual message-processing pipeline that allows concurrent routing. This is achieved by embedding routers’ routing logic directly in their `ActorRef`

rather than in the router actor. Messages sent to a router's `ActorRef` can be immediately routed to the routee, bypassing the single-threaded router actor entirely.

The cost to this is, of course, that the internals of routing code are more complicated than if routers were implemented with normal actors. Fortunately all of this complexity is invisible to consumers of the routing API. However, it is something to be aware of when implementing your own routers.

3.6.7 Custom Router

You can 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.

Before creating your own router you should consider whether a normal actor with router-like behavior might do the job just as well as a full-blown router. As explained *above*, the primary benefit of routers over normal actors is their higher performance. But they are somewhat more complicated to write than normal actors. Therefore if lower maximum throughput is acceptable in your application you may wish to stick with traditional actors. This section, however, assumes that you wish to get maximum performance and so demonstrates how you can create your own router.

The router created in this example is replicating each message to a few destinations.

Start with the routing logic:

```
import scala.collection.immutable
import scala.concurrent.forkjoin.ThreadLocalRandom
import akka.routing.RoundRobinRoutingLogic
import akka.routing.RoutingLogic
import akka.routing.Routee
import akka.routing.SeveralRoutees

class RedundancyRoutingLogic(nbrCopies: Int) extends RoutingLogic {
  val roundRobin = RoundRobinRoutingLogic()
  def select(message: Any, routees: immutable.IndexedSeq[Routee]): Routee = {
    val targets = (1 to nbrCopies).map(_ => roundRobin.select(message, routees))
    SeveralRoutees(targets)
  }
}
```

`select` will be called for each message and in this example pick a few destinations by round-robin, by reusing the existing `RoundRobinRoutingLogic` and wrap the result in a `SeveralRoutees` instance. `SeveralRoutees` will send the message to all of the supplied routues.

The implementation of the routing logic must be thread safe, since it might be used outside of actors.

A unit test of the routing logic:

```
case class TestRoutee(n: Int) extends Routee {
  override def send(message: Any, sender: ActorRef): Unit = ()
}

  val logic = new RedundancyRoutingLogic(nbrCopies = 3)

  val routees = for (n <- 1 to 7) yield TestRoutee(n)

  val r1 = logic.select("msg", routees)
  r1.asInstanceOf[SeveralRoutees].routees should be(
    Vector(TestRoutee(1), TestRoutee(2), TestRoutee(3)))

  val r2 = logic.select("msg", routees)
  r2.asInstanceOf[SeveralRoutees].routees should be(
    Vector(TestRoutee(4), TestRoutee(5), TestRoutee(6)))

  val r3 = logic.select("msg", routees)
```

```
  r3.asInstanceOf[SeveralRoutees].routees should be(
    Vector(TestRoutee(7), TestRoutee(1), TestRoutee(2)))
```

You could stop here and use the `RedundancyRoutingLogic` with a `akka.routing.Router` as described in *A Simple Router*.

Let us continue and make this into a self contained, configurable, router actor.

Create a class that extends `Pool`, `Group` or `CustomRouterConfig`. That class is a factory for the routing logic and holds the configuration for the router. Here we make it a `Group`.

```
import akka.dispatch.Dispatchers
import akka.routing.Group
import akka.routing.Router
import akka.japi.Util.immutableSeq
import com.typesafe.config.Config

case class RedundancyGroup(override val paths: immutable.Iterable[String], nbrCopies: Int) extends

  def this(config: Config) = this(
    paths = immutableSeq(config.getStringList("routees.paths")),
    nbrCopies = config.getInt("nbr-copies"))

  override def createRouter(system: ActorSystem): Router =
    new Router(new RedundancyRoutingLogic(nbrCopies))

  override val routerDispatcher: String = Dispatchers.DefaultDispatcherId
}
```

This can be used exactly as the router actors provided by Akka.

```
for (n <- 1 to 10) system.actorOf(Props[Storage], "s" + n)

val paths = for (n <- 1 to 10) yield ("/user/s" + n)
val redundancy1: ActorRef =
  system.actorOf(RedundancyGroup(paths, nbrCopies = 3).props(),
    name = "redundancy1")
redundancy1 ! "important"
```

Note that we added a constructor in `RedundancyGroup` that takes a `Config` parameter. That makes it possible to define it in configuration.

```
akka.actor.deployment {
  /redundancy2 {
    router = "docs.routing.RedundancyGroup"
    routees.paths = ["/user/s1", "/user/s2", "/user/s3"]
    nbr-copies = 5
  }
}
```

Note the fully qualified class name in the `router` property. The router class must extend `akka.routing.RouterConfig` (`Pool`, `Group` or `CustomRouterConfig`) and have constructor with one `com.typesafe.config.Config` parameter. The deployment section of the configuration is passed to the constructor.

```
val redundancy2: ActorRef = system.actorOf(FromConfig.props(),
  name = "redundancy2")
redundancy2 ! "very important"
```

3.6.8 Configuring Dispatchers

The dispatcher for created children of the pool will be taken from `Props` as described in *Dispatchers*.

To make it easy to define the dispatcher of the routees of the pool you can define the dispatcher inline in the deployment section of the config.

```
akka.actor.deployment {
  /poolWithDispatcher {
    router = random-pool
    nr-of-instances = 5
    pool-dispatcher {
      fork-join-executor.parallelism-min = 5
      fork-join-executor.parallelism-max = 5
    }
  }
}
```

That is the only thing you need to do enable a dedicated dispatcher for a pool.

Note: If you use a group of actors and route to their paths, 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: ActorRef = system.actorOf(
  // "head" router actor will run on "router-dispatcher" dispatcher
  // Worker routees will run on "pool-dispatcher" dispatcher
  RandomPool(5, routerDispatcher = "router-dispatcher").props(Props[Worker]),
  name = "poolWithDispatcher")
```

Note: It is not allowed to configure the `routerDispatcher` to be a `akka.dispatch.BalancingDispatcherConfigurator` since the messages meant for the special router actor cannot be processed by any other actor.

3.7 FSM

3.7.1 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'.*

3.7.2 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 scala.concurrent.duration._
```

The contract of our "Buncher" actor is that it 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: immutable.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: immutable.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 | 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:

- `startWith` 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*, which is conveniently bundled with ScalaTest traits into `AkkaSpec`:

```
import akka.actor.Props
import scala.collection.immutable

class FSMDocSpec extends MyFavoriteTestFrameWorkPlusAkkaTestKit {

  // fsm code elided ...

  "simple finite state machine" must {

    "demonstrate NullFunction" in {
      class A extends Actor with FSM[Int, Null] {
        val SomeState = 0
        when(SomeState)(FSM.NullFunction)
      }
    }

    "batch correctly" in {
      val buncher = system.actorOf(Props(classOf[Buncher], this))
      buncher ! SetTarget(testActor)
      buncher ! Queue(42)
      buncher ! Queue(43)
```

```
      expectMsg(Batch(immutable.Seq(42, 43)))
      buncher ! Queue(44)
      buncher ! Flush
      buncher ! Queue(45)
      expectMsg(Batch(immutable.Seq(44)))
      expectMsg(Batch(immutable.Seq(45)))
    }

    "not batch if uninitialized" in {
      val buncher = system.actorOf(Props(classOf[Buncher], this))
      buncher ! Queue(42)
      expectNoMsg
    }
  }
}
```

3.7.3 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:

```
class Buncher extends Actor with FSM[State, Data] {

  // fsm body ...

  initialize()
}
```

Note: The FSM trait defines a `receive` method which handles internal messages and passes everything else through to the FSM logic (according to the current state). When overriding the `receive` method, keep in mind that e.g. state timeout handling depends on actually passing the messages through the FSM logic.

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(SetTarget(ref), Uninitialized) =>
    stay using Todo(ref, Vector.empty)
}

when(Active, stateTimeout = 1 second) {
  case Event(Flush | StateTimeout, t: Todo) =>
    goto(Idle) using t.copy(queue = Vector.empty)
}
```

The `Event(msg: Any, data: D)` case class is parameterized with the data type held by the FSM for convenient pattern matching.

Warning: It is required that you define handlers for each of the possible FSM states, otherwise there will be failures when trying to switch to undeclared states.

It is recommended practice to declare the states as objects extending a sealed trait and then verify that there is a `when` clause for each of the states. If you want to leave the handling of a state "unhandled" (more below), it still needs to be declared like this:

```
when(SomeState)(FSM.NullFunction)
```

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("Received unhandled event: " + x)
    stay
  case Event(msg, _) =>
    log.warning("Received unknown event: " + msg)
    goto(Error)
}
```

Within this handler the state of the FSM may be queried using the `stateName` method.

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 modifiers can be chained to achieve a nice and concise description:

```
when(SomeState) {
  case Event(msg, _) =>
    goto(Processing) using (newData) 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", Tick, 1 second, true)
  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 _)

def handler(from: StateType, to: StateType) {
  // handle it here ...
}
```

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.

Stopping a listener without unregistering will not remove the listener from the subscription list; use `UnsubscribeTransitionCallback` before stopping the listener.

Transforming State

The partial functions supplied as argument to the `when()` blocks can be transformed using Scala's full supplement of functional programming tools. In order to retain type inference, there is a helper function which may be used in case some common handling logic shall be applied to different clauses:

```
when(SomeState)(transform {
  case Event(bytes: ByteString, read) => stay using (read + bytes.length)
} using {
  case s @ FSM.State(state, read, timeout, stopReason, replies) if read > 1000 =>
    goto(Processing)
})
```

It goes without saying that the arguments to this method may also be stored, to be used several times, e.g. when applying the same transformation to several `when()` blocks:

```
val processingTrigger: PartialFunction[State, State] = {
  case s @ FSM.State(state, read, timeout, stopReason, replies) if read > 1000 =>
    goto(Processing)
}

when(SomeState)(transform {
  case Event(bytes: ByteString, read) => stay using (read + bytes.length)
} using processingTrigger)
```

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. Any existing timer with the same name will automatically be canceled before adding the new timer.

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

```
isTimerActive(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(Error) {
  case Event("stop", _) =>
    // do cleanup ...
    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(FSM.Normal, state, data)         => // ...
  case StopEvent(FSM.Shutdown, state, data)       => // ...
  case StopEvent(FSM.Failure(cause), state, data) => // ...
}
```

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`.

3.7.4 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 *Configuration* enables logging of an event trace by `LoggingFSM` instances:

```
import akka.actor.LoggingFSM
class MyFSM extends Actor with LoggingFSM[StateType, Data] {
  // body elided ...
}
```

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:

```
import akka.actor.LoggingFSM
class MyFSM extends Actor with LoggingFSM[StateType, Data] {
  override def logDepth = 12
  onTermination {
    case StopEvent(FSM.Failure(_), state, data) =>
      val lastEvents = getLog.mkString("\n\t")
      log.warning("Failure in state " + state + " with data " + data + "\n" +
        "Events leading up to this point:\n\t" + lastEvents)
  }
  // ...
}
```

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.

3.7.5 Examples

A bigger FSM example contrasted with Actor's `become`/`unbecome` can be found in the Typesafe Activator template named Akka FSM in Scala

3.8 Persistence

Akka persistence enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster. The key concept behind Akka persistence is that only changes to an actor's internal state are persisted but never its current state directly (except for optional snapshots). These changes are only ever appended to storage, nothing is ever mutated, which allows for very high transaction rates and efficient replication. Stateful actors are recovered by replaying stored changes to these actors from which they can rebuild internal state. This can be either the full history of changes or starting from a snapshot which can dramatically reduce recovery times. Akka persistence also provides point-to-point communication channels with at-least-once message delivery semantics.

> **Warning:** This module is marked as **"experimental"** as of its introduction in Akka 2.3.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum the binary compatibility guarantee for maintenance releases does not apply to the contents of the `akka.persistence` package.

Akka persistence is inspired by and the official replacement of the eventsourced library. It follows the same concepts and architecture of eventsourced but significantly differs on API and implementation level. See also *Migration Guide Eventsourced to Akka Persistence 2.3.x*

3.8.1 Dependencies

Akka persistence is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-persistence-experimental" % "2.3.2"
```

3.8.2 Architecture

- *Processor*: A processor is a persistent, stateful actor. Messages sent to a processor are written to a journal before its `receive` method is called. When a processor is started or restarted, journaled messages are replayed to that processor, so that it can recover internal state from these messages.
- *View*: A view is a persistent, stateful actor that receives journaled messages that have been written by another processor. A view itself does not journal new messages, instead, it updates internal state only from a processor's replicated message stream.
- *Channel*: Channels are used by processors and views to communicate with other actors. They prevent that replayed messages are redundantly delivered to these actors and provide at-least-once message delivery semantics, also in case of sender and receiver JVM crashes.
- *Journal*: A journal stores the sequence of messages sent to a processor. An application can control which messages are journaled and which are received by the processor without being journaled. The storage backend of a journal is pluggable. The default journal storage plugin writes to the local filesystem, replicated journals are available as Community plugins.
- *Snapshot store*: A snapshot store persists snapshots of a processor's or a view's internal state. Snapshots are used for optimizing recovery times. The storage backend of a snapshot store is pluggable. The default snapshot storage plugin writes to the local filesystem.
- *Event sourcing*. Based on the building blocks described above, Akka persistence provides abstractions for the development of event sourced applications (see section *Event sourcing*)

3.8.3 Processors

A processor can be implemented by extending the `Processor` trait and implementing the `receive` method.

```
import akka.persistence.{ Persistent, PersistenceFailure, Processor }

class MyProcessor extends Processor {
  def receive = {
    case Persistent(payload, sequenceNr) =>
    // message successfully written to journal
    case PersistenceFailure(payload, sequenceNr, cause) =>
    // message failed to be written to journal
    case other =>
    // message not written to journal
  }
}
```

Processors only write messages of type `Persistent` to the journal, others are received without being persisted. When a processor's `receive` method is called with a `Persistent` message it can safely assume that this message has been successfully written to the journal. If a journal fails to write a `Persistent` message then the processor is stopped, by default. If a processor should continue running on persistence failures it must handle `PersistenceFailure` messages. In this case, a processor may want to inform the sender about the failure, so that the sender can re-send the message, if needed.

A `Processor` itself is an `Actor` and can therefore be instantiated with `actorOf`.

```
import akka.actor.Props

val processor = actorOf(Props[MyProcessor], name = "myProcessor")

processor ! Persistent("foo") // will be journaled
processor ! "bar" // will not be journaled
```

Recovery

By default, a processor is automatically recovered on start and on restart by replaying journaled messages. New messages sent to a processor during recovery do not interfere with replayed messages. New messages will only be received by a processor after recovery completes.

Recovery customization

Automated recovery on start can be disabled by overriding `preStart` with an empty implementation.

```
override def preStart() = ()
```

In this case, a processor must be recovered explicitly by sending it a `Recover()` message.

```
processor ! Recover()
```

If not overridden, `preStart` sends a `Recover()` message to `self`. Applications may also override `preStart` to define further `Recover()` parameters such as an upper sequence number bound, for example.

```
override def preStart() {
  self ! Recover(toSequenceNr = 457L)
}
```

Upper sequence number bounds can be used to recover a processor to past state instead of current state. Automated recovery on restart can be disabled by overriding `preRestart` with an empty implementation.

```
override def preRestart(reason: Throwable, message: Option[Any]) = ()
```

Recovery status

A processor can query its own recovery status via the methods

```
def recoveryRunning: Boolean
def recoveryFinished: Boolean
```

Sometimes there is a need for performing additional initialization when the recovery has completed, before processing any other message sent to the processor. The processor can send itself a message from `preStart`. It will be stashed and received after recovery. The mailbox may contain other messages that are queued in front of that message and therefore you need to stash until you receive that message.

```
override def preStart(): Unit = {
  super.preStart()
  self ! "FIRST"
}

def receive = initializing.orElse(active)

def initializing: Receive = {
  case "FIRST" =>
    recoveryCompleted()
    context.become(active)
    unstashAll()
  case other if recoveryFinished =>
    stash()
}

def recoveryCompleted(): Unit = {
  // perform init after recovery, before any other messages
  // ...
}

def active: Receive = {
  case Persistent(msg, _) => //...
}
```

Failure handling

A persistent message that caused an exception will be received again by a processor after restart. To prevent a replay of that message during recovery it can be deleted.

```
override def preRestart(reason: Throwable, message: Option[Any]) {
  message match {
    case Some(p: Persistent) => deleteMessage(p.sequenceNr)
    case _                   =>
  }
  super.preRestart(reason, message)
}
```

Message deletion

A processor can delete a single message by calling the `deleteMessage` method with the sequence number of that message as argument. An optional `permanent` parameter specifies whether the message shall be permanently deleted from the journal or only marked as deleted. In both cases, the message won't be replayed. Later extensions to Akka persistence will allow to replay messages that have been marked as deleted which can be useful for debugging purposes, for example. To delete all messages (journaled by a single processor) up to a specified sequence number, processors should call the `deleteMessages` method.

Identifiers

A processor must have an identifier that doesn't change across different actor incarnations. It defaults to the `String` representation of processor's path without the address part and can be obtained via the `processorId`

method.

```
def processorId: String
```

Applications can customize a processor's id by specifying an actor name during processor creation as shown in section *Processors*. This changes that processor's name in its actor hierarchy and hence influences only part of the processor id. To fully customize a processor's id, the `processorId` method must be overridden.

```
override def processorId = "my-stable-processor-id"
```

Overriding `processorId` is the recommended way to generate stable identifiers.

3.8.4 Views

Views can be implemented by extending the `View` trait and implementing the `receive` and the `processorId` methods.

```
class MyView extends View {
  def processorId: String = "some-processor-id"

  def receive: Actor.Receive = {
    case Persistent(payload, sequenceNr) => // ...
  }
}
```

The `processorId` identifies the processor from which the view receives journaled messages. It is not necessary the referenced processor is actually running. Views read messages from a processor's journal directly. When a processor is started later and begins to write new messages, the corresponding view is updated automatically, by default.

Updates

The default update interval of all views of an actor system is configurable:

```
akka.persistence.view.auto-update-interval = 5s
```

`View` implementation classes may also override the `autoUpdateInterval` method to return a custom update interval for a specific view class or view instance. Applications may also trigger additional updates at any time by sending a view an `Update` message.

```
val view = system.actorOf(Props[MyView])
view ! Update(await = true)
```

If the `await` parameter is set to `true`, messages that follow the `Update` request are processed when the incremental message replay, triggered by that update request, completed. If set to `false` (default), messages following the update request may interleave with the replayed message stream. Automated updates always run with `await = false`.

Automated updates of all views of an actor system can be turned off by configuration:

```
akka.persistence.view.auto-update = off
```

Implementation classes may override the configured default value by overriding the `autoUpdate` method. To limit the number of replayed messages per update request, applications can configure a custom `akka.persistence.view.auto-update-replay-max` value or override the `autoUpdateReplayMax` method. The number of replayed messages for manual updates can be limited with the `replayMax` parameter of the `Update` message.

Recovery

Initial recovery of views works in the very same way as for *Processors* (i.e. by sending a `Recover` message to self). The maximum number of replayed messages during initial recovery is determined by `autoUpdateReplayMax`. Further possibilities to customize initial recovery are explained in section *Processors*.

Identifiers

A view must have an identifier that doesn't change across different actor incarnations. It defaults to the `String` representation of the actor path without the address part and can be obtained via the `viewId` method.

Applications can customize a view's id by specifying an actor name during view creation. This changes that view's name in its actor hierarchy and hence influences only part of the view id. To fully customize a view's id, the `viewId` method must be overridden. Overriding `viewId` is the recommended way to generate stable identifiers.

The `viewId` must differ from the referenced `processorId`, unless *Snapshots* of a view and its processor shall be shared (which is what applications usually do not want).

3.8.5 Channels

Channels are special actors that are used by processors or views to communicate with other actors (channel destinations). The following discusses channels in context of processors but this is also applicable to views.

Channels prevent redundant delivery of replayed messages to destinations during processor recovery. A replayed message is retained by a channel if its delivery has been confirmed by a destination.

```
import akka.actor.{ Actor, Props }
import akka.persistence.{ Channel, Deliver, Persistent, Processor }

class MyProcessor extends Processor {
  val destination = context.actorOf(Props[MyDestination])
  val channel = context.actorOf(Channel.props(), name = "myChannel")

  def receive = {
    case p @ Persistent(payload, _) =>
      channel ! Deliver(p.withPayload(s"processed ${payload}"), destination.path)
  }
}

class MyDestination extends Actor {
  def receive = {
    case p @ ConfirmablePersistent(payload, sequenceNr, redeliveries) =>
      // ...
      p.confirm()
  }
}
```

A channel is ready to use once it has been created, no recovery or further activation is needed. A `Deliver` request instructs a channel to send a `Persistent` message to a destination. A destination is provided as `ActorPath` and messages are sent by the channel via that path's `ActorSelection`. Sender references are preserved by a channel, therefore, a destination can reply to the sender of a `Deliver` request.

Note: Sending via a channel has at-least-once delivery semantics—by virtue of either the sending actor or the channel being persistent—which means that the semantics do not match those of a normal `ActorRef` send operation:

- it is not at-most-once delivery
- message order for the same sender–receiver pair is not retained due to possible resends

- after a crash and restart of the destination messages are still delivered—to the new actor incarnation

These semantics match precisely what an `ActorPath` represents (see *Actor Lifecycle*), therefore you need to supply a path and not a reference when constructing `Deliver` messages.

If a processor wants to reply to a `Persistent` message sender it should use the `sender` path as channel destination.

```
channel ! Deliver(p.withPayload(s"processed ${payload}"), sender.path)
```

Persistent messages delivered by a channel are of type `ConfirmablePersistent`. `ConfirmablePersistent` extends `Persistent` by adding the methods `confirm` and `redeliveries` (see also *Message re-delivery*). A channel destination confirms the delivery of a `ConfirmablePersistent` message by calling `confirm()` on that message. This asynchronously writes a confirmation entry to the journal. Replayed messages internally contain confirmation entries which allows a channel to decide if it should retain these messages or not.

A `Processor` can also be used as channel destination i.e. it can persist `ConfirmablePersistent` messages too.

Message re-delivery

Channels re-deliver messages to destinations if they do not confirm delivery within a configurable timeout. This timeout can be specified as `redeliverInterval` when creating a channel, optionally together with the maximum number of re-deliveries a channel should attempt for each unconfirmed message. The number of re-delivery attempts can be obtained via the `redeliveries` method on `ConfirmablePersistent` or by pattern matching.

```
context.actorOf(Channel.props(
  ChannelSettings(redeliverInterval = 30 seconds, redeliverMax = 15)),
  name = "myChannel")
```

A channel keeps messages in memory until their successful delivery has been confirmed or the maximum number of re-deliveries is reached. To be notified about messages that have reached the maximum number of re-deliveries, applications can register a listener at channel creation.

```
class MyListener extends Actor {
  def receive = {
    case RedeliverFailure(messages) => // ...
  }
}

val myListener = context.actorOf(Props[MyListener])
val myChannel = context.actorOf(Channel.props(
  ChannelSettings(redeliverFailureListener = Some(myListener))))
```

A listener receives `RedeliverFailure` notifications containing all messages that could not be delivered. On receiving a `RedeliverFailure` message, an application may decide to restart the sending processor to enforce a re-send of these messages to the channel or confirm these messages to prevent further re-sends. The sending processor can also be restarted any time later to re-send unconfirmed messages.

This combination of

- message persistence by sending processors
- message replays by sending processors
- message re-deliveries by channels and
- application-level confirmations (acknowledgements) by destinations

enables channels to provide at-least-once message delivery semantics. Possible duplicates can be detected by destinations by tracking message sequence numbers. Message sequence numbers are generated per sending pro-

cessor. Depending on how a processor routes outbound messages to destinations, they may either see a contiguous message sequence or a sequence with gaps.

> **Warning:** If a processor emits more than one outbound message per inbound `Persistent` message it **must** use a separate channel for each outbound message to ensure that confirmations are uniquely identifiable, otherwise, at-least-once message delivery semantics do not apply. This rule has been introduced to avoid writing additional outbound message identifiers to the journal which would decrease the overall throughput. It is furthermore recommended to collapse multiple outbound messages to the same destination into a single outbound message, otherwise, if sent via multiple channels, their ordering is not defined.

If an application wants to have more control how sequence numbers are assigned to messages it should use an application-specific sequence number generator and include the generated sequence numbers into the `payload` of `Persistent` messages.

Persistent channels

Channels created with `Channel.props` do not persist messages. These channels are usually used in combination with a sending processor that takes care of persistence, hence, channel-specific persistence is not necessary in this case. They are referred to as transient channels in the following.

Persistent channels are like transient channels but additionally persist messages before delivering them. Messages that have been persisted by a persistent channel are deleted when destinations confirm their delivery. A persistent channel can be created with `PersistentChannel.props` and configured with a `PersistentChannelSettings` object.

```
val channel = context.actorOf(PersistentChannel.props(
  PersistentChannelSettings(redeliverInterval = 30 seconds, redeliverMax = 15)),
  name = "myPersistentChannel")

channel ! Deliver(Persistent("example"), destination.path)
```

A persistent channel is useful for delivery of messages to slow destinations or destinations that are unavailable for a long time. It can constrain the number of pending confirmations based on the `pendingConfirmationsMax` and `pendingConfirmationsMin` parameters of `PersistentChannelSettings`.

```
PersistentChannelSettings(
  pendingConfirmationsMax = 10000,
  pendingConfirmationsMin = 2000)
```

It suspends delivery when the number of pending confirmations reaches `pendingConfirmationsMax` and resumes delivery again when this number falls below `pendingConfirmationsMin`. This prevents both, flooding destinations with more messages than they can process and unlimited memory consumption by the channel. A persistent channel continues to persist new messages even when message delivery is temporarily suspended.

Standalone usage

Applications may also use channels standalone. Transient channels can be used standalone if re-delivery attempts to destinations are required but message loss in case of a sender JVM crash is not an issue. If message loss in case of a sender JVM crash is an issue, persistent channels should be used. In this case, applications may want to receive replies from the channel whether messages have been successfully persisted or not. This can be enabled by creating the channel with the `replyPersistent` configuration parameter set to `true`:

```
PersistentChannelSettings(replyPersistent = true)
```

With this setting, either the successfully persisted message is replied to the sender or a `PersistenceFailure` message. In case the latter case, the sender should re-send the message.

Identifiers

In the same way as *Processors* and *Views*, channels also have an identifier that defaults to a channel's path. A channel identifier can therefore be customized by using a custom actor name at channel creation. This changes that channel's name in its actor hierarchy and hence influences only part of the channel identifier. To fully customize a channel identifier, it should be provided as argument `Channel.props(String)` or `PersistentChannel.props(String)` (recommended to generate stable identifiers).

```
context.actorOf(Channel.props("my-stable-channel-id"))
```

3.8.6 Persistent messages

Payload

The payload of a `Persistent` message can be obtained via its

```
def payload: Any
```

method or by pattern matching

```
case Persistent(payload, _) =>
```

Inside processors, new persistent messages are derived from the current persistent message before sending them via a channel, either by calling `p.withPayload(...)` or `Persistent(...)` where the latter uses the implicit `currentPersistentMessage` made available by `Processor`.

```
implicit def currentPersistentMessage: Option[Persistent]
```

This is necessary for delivery confirmations to work properly. Both ways are equivalent but we recommend using `p.withPayload(...)` for clarity.

Sequence number

The sequence number of a `Persistent` message can be obtained via its

```
def sequenceNr: Long
```

method or by pattern matching

```
case Persistent(_, sequenceNr) =>
```

Persistent messages are assigned sequence numbers on a per-processor basis (or per channel basis if used standalone). A sequence starts at `1L` and doesn't contain gaps unless a processor deletes messages.

3.8.7 Snapshots

Snapshots can dramatically reduce recovery times of processors and views. The following discusses snapshots in context of processors but this is also applicable to views.

Processors can save snapshots of internal state by calling the `saveSnapshot` method. If saving of a snapshot succeeds, the processor receives a `SaveSnapshotSuccess` message, otherwise a `SaveSnapshotFailure` message

```
class MyProcessor extends Processor {
  var state: Any = _

  def receive = {
    case "snap"                                => saveSnapshot(state)
    case SaveSnapshotSuccess(metadata)         => // ...
    case SaveSnapshotFailure(metadata, reason) => // ...
```

```
  }
}
```

where `metadata` is of type `SnapshotMetadata`:

```
case class SnapshotMetadata(processorId: String, sequenceNr: Long, timestamp: Long = 0L)
```

During recovery, the processor is offered a previously saved snapshot via a `SnapshotOffer` message from which it can initialize internal state.

```
class MyProcessor extends Processor {
  var state: Any = _

  def receive = {
    case SnapshotOffer(metadata, offeredSnapshot) => state = offeredSnapshot
    case Persistent(payload, sequenceNr)          => // ...
  }
}
```

The replayed messages that follow the `SnapshotOffer` message, if any, are younger than the offered snapshot. They finally recover the processor to its current (i.e. latest) state.

In general, a processor is only offered a snapshot if that processor has previously saved one or more snapshots and at least one of these snapshots matches the `SnapshotSelectionCriteria` that can be specified for recovery.

```
processor ! Recover(fromSnapshot = SnapshotSelectionCriteria(
  maxSequenceNr = 457L,
  maxTimestamp = System.currentTimeMillis))
```

If not specified, they default to `SnapshotSelectionCriteria.Latest` which selects the latest (= youngest) snapshot. To disable snapshot-based recovery, applications should use `SnapshotSelectionCriteria.None`. A recovery where no saved snapshot matches the specified `SnapshotSelectionCriteria` will replay all journaled messages.

Snapshot deletion

A processor can delete individual snapshots by calling the `deleteSnapshot` method with the sequence number and the timestamp of a snapshot as argument. To bulk-delete snapshots matching `SnapshotSelectionCriteria`, processors should use the `deleteSnapshots` method.

3.8.8 Event sourcing

In all the examples so far, messages that change a processor's state have been sent as `Persistent` messages by an application, so that they can be replayed during recovery. From this point of view, the journal acts as a write-ahead-log for whatever `Persistent` messages a processor receives. This is also known as *command sourcing*. Commands, however, may fail and some applications cannot tolerate command failures during recovery.

For these applications Event Sourcing is a better choice. Applied to Akka persistence, the basic idea behind event sourcing is quite simple. A processor receives a (non-persistent) command which is first validated if it can be applied to the current state. Here, validation can mean anything, from simple inspection of a command message's fields up to a conversation with several external services, for example. If validation succeeds, events are generated from the command, representing the effect of the command. These events are then persisted and, after successful persistence, used to change a processor's state. When the processor needs to be recovered, only the persisted events are replayed of which we know that they can be successfully applied. In other words, events cannot fail when being replayed to a processor, in contrast to commands. Eventsourced processors may of course also process commands that do not change application state, such as query commands, for example.

Akka persistence supports event sourcing with the `EventsourcedProcessor` trait (which implements event sourcing as a pattern on top of command sourcing). A processor that extends this trait does not handle `Persistent` messages directly but uses the `persist` method to persist and handle events. The behavior

of an `EventsourcedProcessor` is defined by implementing `receiveRecover` and `receiveCommand`. This is demonstrated in the following example.

```
import akka.actor._
import akka.persistence._

case class Cmd(data: String)
case class Evt(data: String)

case class ExampleState(events: List[String] = Nil) {
  def update(evt: Evt) = copy(evt.data :: events)
  def size = events.length
  override def toString: String = events.reverse.toString
}

class ExampleProcessor extends EventsourcedProcessor {
  var state = ExampleState()

  def updateState(event: Evt): Unit =
    state = state.update(event)

  def numEvents =
    state.size

  val receiveRecover: Receive = {
    case evt: Evt                                 => updateState(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  }

  val receiveCommand: Receive = {
    case Cmd(data) =>
      persist(Evt(s"${data}-${numEvents}"))(updateState)
      persist(Evt(s"${data}-${numEvents + 1}")) { event =>
        updateState(event)
        context.system.eventStream.publish(event)
      }
    case "snap"  => saveSnapshot(state)
    case "print" => println(state)
  }

}
```

The example defines two data types, `Cmd` and `Evt` to represent commands and events, respectively. The `state` of the `ExampleProcessor` is a list of persisted event data contained in `ExampleState`.

The processor's `receiveRecover` method defines how `state` is updated during recovery by handling `Evt` and `SnapshotOffer` messages. The processor's `receiveCommand` method is a command handler. In this example, a command is handled by generating two events which are then persisted and handled. Events are persisted by calling `persist` with an event (or a sequence of events) as first argument and an event handler as second argument.

The `persist` method persists events asynchronously and the event handler is executed for successfully persisted events. Successfully persisted events are internally sent back to the processor as individual messages that trigger event handler executions. An event handler may close over processor state and mutate it. The sender of a persisted event is the sender of the corresponding command. This allows event handlers to reply to the sender of a command (not shown).

The main responsibility of an event handler is changing processor state using event data and notifying others about successful state changes by publishing events.

When persisting events with `persist` it is guaranteed that the processor will not receive further commands between the `persist` call and the execution(s) of the associated event handler. This also holds for multiple `persist` calls in context of a single command.

The easiest way to run this example yourself is to download Typesafe Activator and open the tutorial named Akka Persistence Samples with Scala. It contains instructions on how to run the `EventsourcedExample`.

Note: It's also possible to switch between different command handlers during normal processing and recovery with `context.become()` and `context.unbecome()`. To get the actor into the same state after recovery you need to take special care to perform the same state transitions with `become` and `unbecome` in the `receiveRecover` method as you would have done in the command handler.

Reliable event delivery

Sending events from an event handler to another actor has at-most-once delivery semantics. For at-least-once delivery, *Channels* must be used. In this case, also replayed events (received by `receiveRecover`) must be sent to a channel, as shown in the following example:

```
class MyEventsourcedProcessor(destination: ActorRef) extends EventsourcedProcessor {
  val channel = context.actorOf(Channel.props("channel"))

  def handleEvent(event: String) = {
    // update state
    // ...
    // reliably deliver events
    channel ! Deliver(Persistent(event), destination.path)
  }

  def receiveRecover: Receive = {
    case event: String => handleEvent(event)
  }

  def receiveCommand: Receive = {
    case "cmd" => {
      // ...
      persist("evt")(handleEvent)
    }
  }
}
```

In larger integration scenarios, channel destinations may be actors that submit received events to an external message broker, for example. After having successfully submitted an event, they should call `confirm()` on the received `ConfirmablePersistent` message.

3.8.9 Batch writes

To optimize throughput, a `Processor` internally batches received `Persistent` messages under high load before writing them to the journal (as a single batch). The batch size dynamically grows from 1 under low and moderate loads to a configurable maximum size (default is `200`) under high load.

```
akka.persistence.journal.max-message-batch-size = 200
```

A new batch write is triggered by a processor as soon as a batch reaches the maximum size or if the journal completed writing the previous batch. Batch writes are never timer-based which keeps latencies at a minimum.

Applications that want to have more explicit control over batch writes and batch sizes can send processors `PersistentBatch` messages.

```
class MyProcessor extends Processor {
  def receive = {
    case Persistent("a", _) => // ...
    case Persistent("b", _) => // ...
  }
```

```
}

val system = ActorSystem("example")
val processor = system.actorOf(Props[MyProcessor])

processor ! PersistentBatch(List(Persistent("a"), Persistent("b")))
```

`Persistent` messages contained in a `PersistentBatch` are always written atomically, even if the batch size is greater than `max-message-batch-size`. Also, a `PersistentBatch` is written isolated from other batches. `Persistent` messages contained in a `PersistentBatch` are received individually by a processor.

`PersistentBatch` messages, for example, are used internally by an `EventsourcedProcessor` to ensure atomic writes of events. All events that are persisted in context of a single command are written as a single batch to the journal (even if `persist` is called multiple times per command). The recovery of an `EventsourcedProcessor` will therefore never be done partially (with only a subset of events persisted by a single command).

Confirmation and deletion operations performed by *Channels* are also batched. The maximum confirmation and deletion batch sizes are configurable with `akka.persistence.journal.max-confirmation-batch-size` and `akka.persistence.journal.max-deletion-batch-size`, respectively.

3.8.10 Storage plugins

Storage backends for journals and snapshot stores are pluggable in Akka persistence. The default journal plugin writes messages to LevelDB (see *Local LevelDB journal*). The default snapshot store plugin writes snapshots as individual files to the local filesystem (see *Local snapshot store*). Applications can provide their own plugins by implementing a plugin API and activate them by configuration. Plugin development requires the following imports:

```
import scala.concurrent.Future
import scala.collection.immutable.Seq
import akka.persistence._
import akka.persistence.journal._
import akka.persistence.snapshot._
```

Journal plugin API

A journal plugin either extends `SyncWriteJournal` or `AsyncWriteJournal`. `SyncWriteJournal` is an actor that should be extended when the storage backend API only supports synchronous, blocking writes. In this case, the methods to be implemented are:

```
/**
 * Plugin API: synchronously writes a batch of persistent messages to the journal.
 * The batch write must be atomic i.e. either all persistent messages in the batch
 * are written or none.
 */
def writeMessages(messages: immutable.Seq[PersistentRepr]): Unit

/**
 * Plugin API: synchronously writes a batch of delivery confirmations to the journal.
 */
def writeConfirmations(confirmations: immutable.Seq[PersistentConfirmation]): Unit

/**
 * Plugin API: synchronously deletes messages identified by `messageIds` from the
 * journal. If `permanent` is set to `false`, the persistent messages are marked as
 * deleted, otherwise they are permanently deleted.
 */
def deleteMessages(messageIds: immutable.Seq[PersistentId], permanent: Boolean): Unit
```

```
/**
 * Plugin API: synchronously deletes all persistent messages up to `toSequenceNr`
 * (inclusive). If `permanent` is set to `false`, the persistent messages are marked
 * as deleted, otherwise they are permanently deleted.
 */
def deleteMessagesTo(processorId: String, toSequenceNr: Long, permanent: Boolean): Unit
```

`AsyncWriteJournal` is an actor that should be extended if the storage backend API supports asynchronous, non-blocking writes. In this case, the methods to be implemented are:

```
/**
 * Plugin API: asynchronously writes a batch of persistent messages to the journal.
 * The batch write must be atomic i.e. either all persistent messages in the batch
 * are written or none.
 */
def asyncWriteMessages(messages: immutable.Seq[PersistentRepr]): Future[Unit]

/**
 * Plugin API: asynchronously writes a batch of delivery confirmations to the journal.
 */
def asyncWriteConfirmations(confirmations: immutable.Seq[PersistentConfirmation]): Future[Unit]

/**
 * Plugin API: asynchronously deletes messages identified by `messageIds` from the
 * journal. If `permanent` is set to `false`, the persistent messages are marked as
 * deleted, otherwise they are permanently deleted.
 */
def asyncDeleteMessages(messageIds: immutable.Seq[PersistentId], permanent: Boolean): Future[Unit]

/**
 * Plugin API: asynchronously deletes all persistent messages up to `toSequenceNr`
 * (inclusive). If `permanent` is set to `false`, the persistent messages are marked
 * as deleted, otherwise they are permanently deleted.
 */
def asyncDeleteMessagesTo(processorId: String, toSequenceNr: Long, permanent: Boolean): Future[Uni
```

Message replays and sequence number recovery are always asynchronous, therefore, any journal plugin must implement:

```
/**
 * Plugin API: asynchronously replays persistent messages. Implementations replay
 * a message by calling `replayCallback`. The returned future must be completed
 * when all messages (matching the sequence number bounds) have been replayed.
 * The future must be completed with a failure if any of the persistent messages
 * could not be replayed.
 *
 * The `replayCallback` must also be called with messages that have been marked
 * as deleted. In this case a replayed message's `deleted` method must return
 * `true`.
 *
 * The channel ids of delivery confirmations that are available for a replayed
 * message must be contained in that message's `confirms` sequence.
 *
 * @param processorId processor id.
 * @param fromSequenceNr sequence number where replay should start (inclusive).
 * @param toSequenceNr sequence number where replay should end (inclusive).
 * @param max maximum number of messages to be replayed.
 * @param replayCallback called to replay a single message. Can be called from any
 *                       thread.
 *
 * @see [[AsyncWriteJournal]]
 * @see [[SyncWriteJournal]]
 */
```

```
def asyncReplayMessages(processorId: String, fromSequenceNr: Long, toSequenceNr: Long, max: Long)

/**
 * Plugin API: asynchronously reads the highest stored sequence number for the
 * given `processorId`.
 *
 * @param processorId processor id.
 * @param fromSequenceNr hint where to start searching for the highest sequence
 *                       number.
 */
def asyncReadHighestSequenceNr(processorId: String, fromSequenceNr: Long): Future[Long]
```

A journal plugin can be activated with the following minimal configuration:

```
# Path to the journal plugin to be used
akka.persistence.journal.plugin = "my-journal"

# My custom journal plugin
my-journal {
  # Class name of the plugin.
  class = "docs.persistence.MyJournal"
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.actor.default-dispatcher"
}
```

The specified plugin `class` must have a no-arg constructor. The `plugin-dispatcher` is the dispatcher used for the plugin actor. If not specified, it defaults to `akka.persistence.dispatchers.default-plugin-dispatcher` for `SyncWriteJournal` plugins and `akka.actor.default-dispatcher` for `AsyncWriteJournal` plugins.

Snapshot store plugin API

A snapshot store plugin must extend the `SnapshotStore` actor and implement the following methods:

```
/**
 * Plugin API: asynchronously loads a snapshot.
 *
 * @param processorId processor id.
 * @param criteria selection criteria for loading.
 */
def loadAsync(processorId: String, criteria: SnapshotSelectionCriteria): Future[Option[SelectedSna

/**
 * Plugin API: asynchronously saves a snapshot.
 *
 * @param metadata snapshot metadata.
 * @param snapshot snapshot.
 */
def saveAsync(metadata: SnapshotMetadata, snapshot: Any): Future[Unit]

/**
 * Plugin API: called after successful saving of a snapshot.
 *
 * @param metadata snapshot metadata.
 */
def saved(metadata: SnapshotMetadata)

/**
 * Plugin API: deletes the snapshot identified by `metadata`.
 *
 * @param metadata snapshot metadata.
 */
```

```
def delete(metadata: SnapshotMetadata)

/**
 * Plugin API: deletes all snapshots matching `criteria`.
 *
 * @param processorId processor id.
 * @param criteria selection criteria for deleting.
 */
def delete(processorId: String, criteria: SnapshotSelectionCriteria)
```

A snapshot store plugin can be activated with the following minimal configuration:

```
# Path to the snapshot store plugin to be used
akka.persistence.snapshot-store.plugin = "my-snapshot-store"

# My custom snapshot store plugin
my-snapshot-store {
  # Class name of the plugin.
  class = "docs.persistence.MySnapshotStore"
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"
}
```

The specified plugin `class` must have a no-arg constructor. The `plugin-dispatcher` is the dispatcher used for the plugin actor. If not specified, it defaults to `akka.persistence.dispatchers.default-plugin-dispatcher`.

3.8.11 Pre-packaged plugins

Local LevelDB journal

The default journal plugin is `akka.persistence.journal.leveldb` which writes messages to a local LevelDB instance. The default location of the LevelDB files is a directory named `journal` in the current working directory. This location can be changed by configuration where the specified path can be relative or absolute:

```
akka.persistence.journal.leveldb.dir = "target/journal"
```

With this plugin, each actor system runs its own private LevelDB instance.

Shared LevelDB journal

A LevelDB instance can also be shared by multiple actor systems (on the same or on different nodes). This, for example, allows processors to failover to a backup node and continue using the shared journal instance from the backup node.

> **Warning:** A shared LevelDB instance is a single point of failure and should therefore only be used for testing purposes. Highly-available, replicated journal are available as Community plugins.

A shared LevelDB instance is started by instantiating the `SharedLeveldbStore` actor.

```
import akka.persistence.journal.leveldb.SharedLeveldbStore

val store = system.actorOf(Props[SharedLeveldbStore], "store")
```

By default, the shared instance writes journaled messages to a local directory named `journal` in the current working directory. The storage location can be changed by configuration:

```
akka.persistence.journal.leveldb-shared.store.dir = "target/shared"
```

Actor systems that use a shared LevelDB store must activate the `akka.persistence.journal.leveldb-shared` plugin.

```
akka.persistence.journal.plugin = "akka.persistence.journal.leveldb-shared"
```

This plugin must be initialized by injecting the (remote) `SharedLeveldbStore` actor reference. Injection is done by calling the `SharedLeveldbJournal.setStore` method with the actor reference as argument.

```
trait SharedStoreUsage extends Actor {
  override def preStart(): Unit = {
    context.actorSelection("akka.tcp://example@127.0.0.1:2552/user/store") ! Identify(1)
  }

  def receive = {
    case ActorIdentity(1, Some(store)) =>
      SharedLeveldbJournal.setStore(store, context.system)
  }
}
```

Internal journal commands (sent by processors) are buffered until injection completes. Injection is idempotent i.e. only the first injection is used.

Local snapshot store

The default snapshot store plugin is `akka.persistence.snapshot-store.local`. It writes snapshot files to the local filesystem. The default storage location is a directory named `snapshots` in the current working directory. This can be changed by configuration where the specified path can be relative or absolute:

```
akka.persistence.snapshot-store.local.dir = "target/snapshots"
```

3.8.12 Custom serialization

Serialization of snapshots and payloads of `Persistent` messages is configurable with Akka's *Serialization* infrastructure. For example, if an application wants to serialize

- payloads of type `MyPayload` with a custom `MyPayloadSerializer` and
- snapshots of type `MySnapshot` with a custom `MySnapshotSerializer`

it must add

```
akka.actor {
  serializers {
    my-payload = "docs.persistence.MyPayloadSerializer"
    my-snapshot = "docs.persistence.MySnapshotSerializer"
  }
  serialization-bindings {
    "docs.persistence.MyPayload" = my-payload
    "docs.persistence.MySnapshot" = my-snapshot
  }
}
```

to the application configuration. If not specified, a default serializer is used.

3.8.13 Testing

When running tests with LevelDB default settings in `sbt`, make sure to set `fork := true` in your sbt project otherwise, you'll see an `UnsatisfiedLinkError`. Alternatively, you can switch to a LevelDB Java port by setting

```
akka.persistence.journal.leveldb.native = off
```

or

```
akka.persistence.journal.leveldb-shared.store.native = off
```

in your Akka configuration. The LevelDB Java port is for testing purposes only.

3.8.14 Miscellaneous

State machines

State machines can be persisted by mixing in the `FSM` trait into processors.

```
import akka.actor.FSM
import akka.persistence.{ Processor, Persistent }

class PersistentDoor extends Processor with FSM[String, Int] {
  startWith("closed", 0)

  when("closed") {
    case Event(Persistent("open", _), counter) =>
      goto("open") using (counter + 1) replying (counter)
  }

  when("open") {
    case Event(Persistent("close", _), counter) =>
      goto("closed") using (counter + 1) replying (counter)
  }
}
```

3.8.15 Configuration

There are several configuration properties for the persistence module, please refer to the *reference configuration*.

3.9 Testing Actor Systems

3.9.1 TestKit Example

Ray Roestenburg's example code from his blog adapted to work with Akka 2.x.

```
import scala.util.Random

import org.scalatest.BeforeAndAfterAll
import org.scalatest.WordSpecLike
import org.scalatest.Matchers

import com.typesafe.config.ConfigFactory

import akka.actor.Actor
import akka.actor.ActorRef
import akka.actor.ActorSystem
import akka.actor.Props
import akka.testkit.DefaultTimeout
import akka.testkit.ImplicitSender
import akka.testkit.TestKit
import scala.concurrent.duration._
import scala.collection.immutable
```

```
/**
 * a Test to show some TestKit examples
 */
class TestKitUsageSpec
  extends TestKit(ActorSystem("TestKitUsageSpec",
    ConfigFactory.parseString(TestKitUsageSpec.config)))
  with DefaultTimeout with ImplicitSender
  with WordSpecLike with Matchers with BeforeAndAfterAll {
  import TestKitUsageSpec._

  val echoRef = system.actorOf(Props[EchoActor])
  val forwardRef = system.actorOf(Props(classOf[ForwardingActor], testActor))
  val filterRef = system.actorOf(Props(classOf[FilteringActor], testActor))
  val randomHead = Random.nextInt(6)
  val randomTail = Random.nextInt(10)
  val headList = immutable.Seq().padTo(randomHead, "0")
  val tailList = immutable.Seq().padTo(randomTail, "1")
  val seqRef =
    system.actorOf(Props(classOf[SequencingActor], testActor, headList, tailList))

  override def afterAll {
    shutdown()
  }

  "An EchoActor" should {
    "Respond with the same message it receives" in {
      within(500 millis) {
        echoRef ! "test"
        expectMsg("test")
      }
    }
  }
  "A ForwardingActor" should {
    "Forward a message it receives" in {
      within(500 millis) {
        forwardRef ! "test"
        expectMsg("test")
      }
    }
  }
  "A FilteringActor" should {
    "Filter all messages, except expected messagetypes it receives" in {
      var messages = Seq[String]()
      within(500 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(Seq("some", "more", "text"))
    }
  }
```

```
  "A SequencingActor" should {
    "receive an interesting message at some point " in {
      within(500 millis) {
        ignoreMsg {
          case msg: String => msg != "something"
        }
        seqRef ! "something"
        expectMsg("something")
        ignoreMsg {
          case msg: String => msg == "1"
        }
        expectNoMsg
        ignoreNoMsg
      }
    }
  }
}

object TestKitUsageSpec {
  // Define your test specific configuration here
  val config = """
    akka {
      loglevel = "WARNING"
    }
    """

  /**
   * An Actor that echoes everything you send to it
   */
  class EchoActor extends Actor {
    def receive = {
      case msg => sender() ! 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 random tail list. The idea is that you would
   * like to test that the interesting value is received and that you cant
   * be bothered with the rest
   */
  class SequencingActor(next: ActorRef, head: immutable.Seq[String],
                        tail: immutable.Seq[String]) extends Actor {
    def receive = {
```

```
      case msg => {
        head foreach { next ! _ }
        next ! msg
        tail foreach { 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.

3.9.2 Synchronous Unit Testing with `TestActorRef`

Testing the business logic inside `Actor` classes can be divided into two parts: first, each atomic operation must work in isolation, then sequences of incoming events must be processed correctly, even in the presence of some possible variability in the ordering of events. The former is the primary use case for single-threaded unit testing, while the latter can only be verified in integration tests.

Normally, the `ActorRef` shields the underlying `Actor` instance from the outside, the only communications channel is the actor's mailbox. This restriction is an impediment to unit testing, which led to the inception of the `TestActorRef`. This special type of reference is designed specifically for test purposes and allows access to the actor in two ways: either by obtaining a reference to the underlying actor instance, or by invoking or querying the actor's behaviour (`receive`). Each one warrants its own section below.

Obtaining a Reference to an `Actor`

Having access to the actual `Actor` object allows application of all traditional unit testing techniques on the contained methods. Obtaining a reference is done like this:

```
import akka.testkit.TestActorRef

val actorRef = TestActorRef[MyActor]
val actor = actorRef.underlyingActor
```

Since `TestActorRef` is generic in the actor type it returns the underlying actor with its proper static type. From this point on you may bring any unit testing tool to bear on your actor as usual.

Testing Finite State Machines

If your actor under test is a `FSM`, you may use the special `TestFSMRef` which offers all features of a normal `TestActorRef` and in addition allows access to the internal state:

```
import akka.testkit.TestFSMRef
import akka.actor.FSM
import scala.concurrent.duration._

val fsm = TestFSMRef(new TestFsmActor)

val mustBeTypedProperly: TestActorRef[TestFsmActor] = fsm

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.isTimerActive("test") == false)
fsm.setTimer("test", 12, 10 millis, true)
assert(fsm.isTimerActive("test") == true)
fsm.cancelTimer("test")
assert(fsm.isTimerActive("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 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 (see CallingThreadDispatcher); this dispatcher is set implicitly for any actor instantiated into a `TestActorRef`.

```
import akka.testkit.TestActorRef
import scala.concurrent.duration._
import scala.concurrent.Await
import akka.pattern.ask

val actorRef = TestActorRef(new MyActor)
// hypothetical message stimulating a '42' answer
val future = actorRef ? Say42
val Success(result: Int) = future.value.get
result should 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 sophisti-

cated 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.

Note: To summarize: `TestActorRef` overwrites two fields: it sets the dispatcher to `CallingThreadDispatcher.global` and it sets the `receiveTimeout` to None.

The Way In-Between: Expecting Exceptions

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 on `TestActorRef`, which will be forwarded to the underlying actor:

```
import akka.testkit.TestActorRef

val actorRef = TestActorRef(new Actor {
  def receive = {
    case "hello" => throw new IllegalArgumentException("boom")
  }
})
intercept[IllegalArgumentException] { actorRef.receive("hello") }
```

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.

3.9.3 Asynchronous 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.WordSpecLike
import org.scalatest.Matchers
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 WordSpecLike with Matchers with BeforeAndAfterAll {

  def this() = this(ActorSystem("MySpec"))

  import MySpec._

  override def afterAll {
    TestKit.shutdownActorSystem(system)
  }

  "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.

The ActorSystem passed in to the constructor of TestKit is accessible via the `system` member. 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(implicitly[ClassTag[T]].runtimeClass)`.

- `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[An`

  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.

- `awaitAssert(a: => Any, max: Duration, interval: Duration)`

  Poll the given assert function every `interval` until it does not throw an exception or the `max` duration is used up. If the timeout expires the last exception is thrown. The interval defaults to 100 ms and the maximum defaults to the time remaining in the innermost enclosing *within* block.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 Log Messages

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.loggers = ["akka.testkit.TestEventListener"]
  """))
try {
  val actor = system.actorOf(Props.empty)
  EventFilter[ActorKilledException](occurrences = 1) intercept {
    actor ! Kill
  }
} finally {
  shutdown(system)
}
```

If a number of occurrences is specific—as demonstrated above—then `intercept` will block until that number of matching messages have been received or the timeout configured in `akka.test.filter-leeway` is used up (time starts counting after the passed-in block of code returns). In case of a timeout the test fails.

Note: Be sure to exchange the default logger with the `TestEventListener` in your `application.conf` to enable this function:

```
akka.loggers = [akka.testkit.TestEventListener]
```

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, it is inherited from the innermost enclosing `within` block.

It should be noted that if the last message-receiving assertion of the block is `expectNoMsg` or `receiveWhile`, the final check of the `within` is skipped in order to avoid false positives due to wake-up latencies. This means that while individual contained assertions still use the maximum time bound, the overall block may take arbitrarily longer in this case.

```
import akka.actor.Props
import scala.concurrent.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 scala.concurrent.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 WordSpecLike with Matchers 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 scala.concurrent.duration._
import akka.actor._
import scala.concurrent.Future
```

```
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!

Warning: Any message send from a `TestProbe` to another actor which runs on the CallingThreadDispatcher runs the risk of dead-lock, if that other actor might also send to this probe. The implementation of `TestProbe.watch` and `TestProbe.unwatch` will also send a message to the watchee, which means that it is dangerous to try watching e.g. `TestActorRef` from a `TestProbe`.

Watching Other Actors from Probes

A `TestProbe` can register itself for DeathWatch of any other actor:

```
val probe = TestProbe()
probe watch target
target ! PoisonPill
probe.expectTerminated(target)
```

Replying to Messages Received by Probes

The probes keep track of the communications channel for replies, if possible, so they can also reply:

```
val probe = TestProbe()
val future = probe.ref ? "hello"
probe.expectMsg(0 millis, "hello") // TestActor runs on CallingThreadDispatcher
probe.reply("world")
assert(future.isCompleted && future.value == Some(Success("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(classOf[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): TestActor.AutoPilot =
    msg match {
      case "stop" ⇒ TestActor.NoAutoPilot
      case x      ⇒ testActor.tell(x, sender); TestActor.KeepRunning
    }
})
```

The `run` method must return the auto-pilot for the next message, which may be `KeepRunning` to retain the current one or `NoAutoPilot` to switch it off.

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:

```
val probe = TestProbe()
within(1 second) {
  probe.expectMsg("hello")
}
```

Here, the `expectMsg` call will use the default timeout.

3.9.4 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

Warning: In case the CallingThreadDispatcher is used for top-level actors, but without going through TestActorRef, then there is a time window during which the actor is awaiting construction by the user guardian actor. Sending messages to the actor during this time period will result in them being enqueued and then executed on the guardian's thread instead of the caller's thread. To avoid this, use TestActorRef.

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.

Thread Interruptions

If the CallingThreadDispatcher sees that the current thread has its `isInterrupted()` flag set when message processing returns, it will throw an `InterruptedException` after finishing all its processing (i.e. all messages which need processing as described above are processed before this happens). As `tell` cannot throw exceptions due to its contract, this exception will then be caught and logged, and the thread's interrupted status will be set again.

If during message processing an `InterruptedException` is thrown then it will be caught inside the CallingThreadDispatcher's message handling loop, the thread's interrupted flag will be set and processing continues normally.

Note: The summary of these two paragraphs is that if the current thread is interrupted while doing work under the CallingThreadDispatcher, then that will result in the `isInterrupted` flag to be `true` when the message send returns and no `InterruptedException` will be thrown.

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

3.9.5 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 ...
}
def otherState: Receive = LoggingReceive.withLabel("other") {
  case msg => // Do something else ...
}
```

. 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 event bus logger 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
    }
  }
}
```

3.9.6 Different Testing Frameworks

Akka's own test suite is written using ScalaTest, which also shines through in documentation examples. However, the TestKit and its facilities do not depend on that framework, you can essentially use whichever suits your development style best.

This section contains a collection of known gotchas with some other frameworks, which is by no means exhaustive and does not imply endorsement or special support.

When you need it to be a trait

If for some reason it is a problem to inherit from `TestKit` due to it being a concrete class instead of a trait, there's `TestKitBase`:

```
import akka.testkit.TestKitBase

class MyTest extends TestKitBase {
  implicit lazy val system = ActorSystem()

  // put your test code here ...

  shutdown(system)
}
```

The `implicit lazy val system` must be declared exactly like that (you can of course pass arguments to the actor system factory as needed) because trait `TestKitBase` needs the system during its construction.

Warning: Use of the trait is discouraged because of potential issues with binary backwards compatibility in the future, use at own risk.

Specs2

Some Specs2 users have contributed examples of how to work around some clashes which may arise:

- Mixing TestKit into `org.specs2.mutable.Specification` results in a name clash involving the `end` method (which is a private variable in TestKit and an abstract method in Specification); if mixing in TestKit first, the code may compile but might then fail at runtime. The work-around—which is actually beneficial also for the third point—is to apply the TestKit together with `org.specs2.specification.Scope`.
- The Specification traits provide a `Duration` DSL which uses partly the same method names as `scala.concurrent.duration.Duration`, resulting in ambiguous implicits if `scala.concurrent.duration._` is imported. There are two work-arounds:
  - either use the Specification variant of Duration and supply an implicit conversion to the Akka Duration. This conversion is not supplied with the Akka distribution because that would mean that our JAR files would dependon Specs2, which is not justified by this little feature.
  - or mix `org.specs2.time.NoTimeConversions` into the Specification.
- Specifications are by default executed concurrently, which requires some care when writing the tests or alternatively the `sequential` keyword.

3.9.7 Configuration

There are several configuration properties for the TestKit module, please refer to the *reference configuration*.

3.10 Actor DSL

3.10.1 The Actor DSL

Simple actors—for example one-off workers or even when trying things out in the REPL—can be created more concisely using the `Act` trait. The supporting infrastructure is bundled in the following import:

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem

implicit val system = ActorSystem("demo")
```

This import is assumed for all code samples throughout this section. The implicit actor system serves as `ActorRefFactory` for all examples below. To define a simple actor, the following is sufficient:

```
val a = actor(new Act {
  become {
    case "hello" ⇒ sender() ! "hi"
  }
})
```

Here, `actor` takes the role of either `system.actorOf` or `context.actorOf`, depending on which context it is called in: it takes an implicit `ActorRefFactory`, which within an actor is available in the form of the `implicit val context: ActorContext`. Outside of an actor, you'll have to either declare an implicit `ActorSystem`, or you can give the factory explicitly (see further below).

The two possible ways of issuing a `context.become` (replacing or adding the new behavior) are offered separately to enable a clutter-free notation of nested receives:

```
val a = actor(new Act {
  become { // this will replace the initial (empty) behavior
    case "info" ⇒ sender() ! "A"
    case "switch" ⇒
      becomeStacked { // this will stack upon the "A" behavior
        case "info"   ⇒ sender() ! "B"
        case "switch" ⇒ unbecome() // return to the "A" behavior
      }
    case "lobotomize" ⇒ unbecome() // OH NOES: Actor.emptyBehavior
  }
})
```

Please note that calling `unbecome` more often than `becomeStacked` results in the original behavior being installed, which in case of the `Act` trait is the empty behavior (the outer `become` just replaces it during construction).

Life-cycle management

Life-cycle hooks are also exposed as DSL elements (see *Start Hook* and *Stop Hook*), where later invocations of the methods shown below will replace the contents of the respective hooks:

```
val a = actor(new Act {
  whenStarting { testActor ! "started" }
  whenStopping { testActor ! "stopped" }
})
```

The above is enough if the logical life-cycle of the actor matches the restart cycles (i.e. `whenStopping` is executed before a restart and `whenStarting` afterwards). If that is not desired, use the following two hooks (see *Restart Hooks*):

```
val a = actor(new Act {
  become {
    case "die" ⇒ throw new Exception
  }
  whenFailing { case m @ (cause, msg) ⇒ testActor ! m }
  whenRestarted { cause ⇒ testActor ! cause }
})
```

It is also possible to create nested actors, i.e. grand-children, like this:

```
// here we pass in the ActorRefFactory explicitly as an example
val a = actor(system, "fred")(new Act {
  val b = actor("barney")(new Act {
    whenStarting { context.parent ! ("hello from " + self.path) }
  })
  become {
    case x ⇒ testActor ! x
  }
})
```

Note: In some cases it will be necessary to explicitly pass the `ActorRefFactory` to the `actor` method (you will notice when the compiler tells you about ambiguous implicits).

The grand-child will be supervised by the child; the supervisor strategy for this relationship can also be configured using a DSL element (supervision directives are part of the `Act` trait):

```
superviseWith(OneForOneStrategy() {
  case e: Exception if e.getMessage == "hello" ⇒ Stop
  case _: Exception                            ⇒ Resume
})
```

Actor with `Stash`

Last but not least there is a little bit of convenience magic built-in, which detects if the runtime class of the statically given actor subtype extends the `RequiresMessageQueue` trait via the `Stash` trait (this is a complicated way of saying that `new Act with Stash` would not work because its runtime erased type is just an anonymous subtype of `Act`). The purpose is to automatically use the appropriate deque-based mailbox type required by `Stash`. If you want to use this magic, simply extend `ActWithStash`:

```
val a = actor(new ActWithStash {
  become {
    case 1 ⇒ stash()
    case 2 ⇒
      testActor ! 2; unstashAll(); becomeStacked {
        case 1 ⇒ testActor ! 1; unbecome()
      }
  }
})
```

CHAPTER

FOUR

FUTURES AND AGENTS

4.1 Futures

4.1.1 Introduction

In the Scala Standard Library, a Future is a data structure used to retrieve the result of some concurrent operation. This result can be accessed synchronously (blocking) or asynchronously (non-blocking).

4.1.2 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 scala.concurrent.{ 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()
```

Within Actors

Each actor is configured to be run on a `MessageDispatcher`, and that dispatcher doubles as an `ExecutionContext`. If the nature of the Future calls invoked by the actor matches or is compatible with the activities of that actor (e.g. all CPU bound and no latency requirements), then it may be easiest to reuse the dispatcher for running the Futures by importing `context.dispatcher`.

```
class A extends Actor {
  import context.dispatcher
  val f = Future("hello")
  def receive = {
    // receive omitted ...
  }
}
```

4.1.3 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`:

```
import scala.concurrent.Await
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.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 scala.concurrent.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.

To send the result of a `Future` to an `Actor`, you can use the `pipe` construct:

```
import akka.pattern.pipe
future pipeTo actor
```

4.1.4 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 scala.concurrent.Await
import scala.concurrent.Future
import scala.concurrent.duration._

val future = Future {
  "Hello" + "World"
}
future foreach println
```

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 `Future` companion, which can be either successes:

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

Or failures:

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

It is also possible to create an empty `Promise`, to be filled later, and obtain the corresponding `Future`:

```
val promise = Promise[String]()
val theFuture = promise.future
promise.success("hello")
```

4.1.5 Functional Futures

Scala'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
}
f2 foreach println
```

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 = Future.successful(3)
val f3 = f1 map { x =>
  f2 map { y =>
    x.length * y
  }
}
f3 foreach println
```

`f3` is a `Future[Future[Int]]` instead of the desired `Future[Int]`. Instead, the `flatMap` method should be used:

```
val f1 = Future {
  "Hello" + "World"
}
val f2 = Future.successful(3)
val f3 = f1 flatMap { x =>
  f2 map { y =>
    x.length * y
  }
}
f3 foreach println
```

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 = Future.successful(4)
val future2 = future1.filter(_ % 2 == 0)

future2 foreach println

val failedFilter = future1.filter(_ % 2 == 1).recover {
  // When filter fails, it will have a java.util.NoSuchElementException
  case m: NoSuchElementException => 0
}

failedFilter foreach println
```

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.

f foreach println
```

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, 3 seconds).asInstanceOf[Int]
val b = Await.result(f2, 3 seconds).asInstanceOf[Int]

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

val result = Await.result(f3, 3 seconds).asInstanceOf[Int]
```

> **Warning:** `Await.result` and `Await.ready` are provided for exceptional situations where you **must** block, a good rule of thumb is to only use them if you know why you **must** block. For all other cases, use asynchronous composition as described below.

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

f3 foreach println
```

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

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

```
// oddActor returns odd numbers sequentially from 1 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 = futureList.map(_.sum)
oddSum foreach println
```

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 = futureList.map(_.sum)
oddSum foreach println
```

This is the same result as this example:

```
val futureList = Future.sequence((1 to 100).toList.map(x => Future(x * 2 - 1)))
val oddSum = futureList.map(_.sum)
oddSum foreach println
```

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.

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

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:

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

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.

4.1.6 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 Scala 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 Success(result)  => doSomethingOnSuccess(result)
  case Failure(failure) => doSomethingOnFailure(failure)
}
```

4.1.7 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 Failure(exception) => log(exception)
} andThen {
  case _ => watchSomeTV()
}
result foreach println
```

4.1.8 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
future4 foreach println
```

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 }
future3 foreach println
```

4.1.9 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
}
future foreach println
```

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:

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

4.1.10 After

`akka.pattern.after` makes it easy to complete a `Future` with a value or exception after a timeout.

```
// TODO after is unfortunately shadowed by ScalaTest, fix as part of #3759
// import akka.pattern.after

val delayed = akka.pattern.after(200 millis, using = system.scheduler)(Future.failed(
  new IllegalStateException("OHNOES")))
val future = Future { Thread.sleep(1000); "foo" }
val result = Future firstCompletedOf Seq(future, delayed)
```

4.2 Agents

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 an `ExecutionContext`. 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 threads.

Note: Agents are local to the node on which they are created. This implies that you should generally not include them in messages that may be passed to remote Actors or as constructor parameters for remote Actors; those remote Actors will not be able to read or update the Agent.

4.2.1 Creating Agents

Agents are created by invoking `Agent(value)` passing in the Agent's initial value and providing an implicit `ExecutionContext` to be used for it, for these examples we're going to use the default global one, but YMMV:

```
import scala.concurrent.ExecutionContext.Implicits.global
import akka.agent.Agent
val agent = Agent(5)
```

4.2.2 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.

4.2.3 Updating Agents (send & alter)

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, enqueues this change
// of the value of the Agent
agent send 7

// send a function, enqueues this change
// to the value of the Agent
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.

```
// the ExecutionContext you want to run the function on
implicit val ec = someExecutionContext()
// sendOff a function
agent sendOff longRunningOrBlockingFunction
```

All `send` methods also have a corresponding `alter` method that returns a `Future`. See *Futures* for more information on `Futures`.

```
// alter a value
val f1: Future[Int] = agent alter 7

// alter a function
val f2: Future[Int] = agent alter (_ + 1)
val f3: Future[Int] = agent alter (_ * 2)
```

```
// the ExecutionContext you want to run the function on
implicit val ec = someExecutionContext()
// alterOff a function
val f4: Future[Int] = agent alterOff longRunningOrBlockingFunction
```

4.2.4 Awaiting an Agent's value

You can also get a `Future` to the Agents value, that will be completed after the currently queued updates have completed:

```
val future = agent.future
```

See *Futures* for more information on `Futures`.

4.2.5 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:

```
import scala.concurrent.ExecutionContext.Implicits.global
val agent1 = Agent(3)
val agent2 = Agent(5)

// uses foreach
for (value <- agent1)
  println(value)

// 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
```

4.2.6 Configuration

There are several configuration properties for the agents module, please refer to the *reference configuration*.

4.2.7 Deprecated Transactional Agents

Agents participating in enclosing STM transaction is a deprecated feature in 2.3.

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 scala.concurrent.ExecutionContext.Implicits.global
import akka.agent.Agent
import scala.concurrent.duration._
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)

val fromValue = from.future // -> 50
val toValue = to.future // -> 70
```

CHAPTER

FIVE

NETWORKING

5.1 Cluster Specification

Note: This document describes the design concepts of the clustering. It is divided into two parts, where the first part describes what is currently implemented and the second part describes what is planned as future enhancements/additions. References to unimplemented parts have been marked with the footnote *[*]*

5.1.1 The Current Cluster

Intro

Akka Cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck. It does this using gossip protocols and an automatic failure detector.

Terms

node A logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a *hostname:port:uid* tuple.

cluster A set of nodes joined together through the membership service.

leader A single node in the cluster that acts as the leader. Managing cluster convergence, partitions *[*]*, fail-over *[*]*, rebalancing *[*]* etc.

Membership

A cluster is made up of a set of member nodes. The identifier for each node is a `hostname:port:uid` tuple. An Akka application can be 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. Joining a cluster is initiated by issuing a `Join` command to one of the nodes in the cluster to join.

The node identifier internally also contains a UID that uniquely identifies this actor system instance at that `hostname:port`. Akka uses the UID to be able to reliably trigger remote death watch. This means that the same actor system can never join a cluster again once it's been removed from that cluster. To re-join an actor system with the same `hostname:port` to a cluster you have to stop the actor system and start a new one with the same `hotname:port` which will then receive a different UID.

The cluster membership state is a specialized CRDT, which means that it has a monotonic merge function. When concurrent changes occur on different nodes the updates can always be merged and converge to the same end result.

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, with preference to members that have not seen the latest version.

Vector Clocks Vector clocks are a type of data structure and algorithm for generating a partial ordering of events in a distributed system and detecting causality violations.

We use vector clocks 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.

Gossip Convergence Information about the cluster converges locally at a node at certain points in time. This is when a node can prove that the cluster state he is observing has been observed by all other nodes in the cluster. Convergence is implemented by passing a set of nodes that have seen current state version during gossip. This information is referred to as the seen set in the gossip overview. When all nodes are included in the seen set there is convergence.

Gossip convergence cannot occur while any nodes are `unreachable`. The nodes need to become `reachable` again, or moved to the `down` and `removed` states (see the Membership Lifecycle section below). This only blocks the leader from performing its cluster membership management and does not influence the application running on top of the cluster. For example this means that during a network partition it is not possible to add more nodes to the cluster. The nodes can join, but they will not be moved to the `up` state until the partition has healed or the unreachable nodes have been downed.

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 other nodes, 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.

In a cluster each node is monitored by a few (default maximum 5) other nodes, and when any of these detects the node as `unreachable` that information will spread to the rest of the cluster through the gossip. In other words, only one node needs to mark a node `unreachable` to have the rest of the cluster mark that node `unreachable`.

The nodes to monitor are picked out of neighbors in a hashed ordered node ring. This is to increase the likelihood to monitor across racks and data centers, but the order is the same on all nodes, which ensures full coverage.

Heartbeats are sent out every second and every heartbeat is performed in a request/reply handshake with the replies used as input to the failure detector.

The failure detector will also detect if the node becomes `reachable` again. When all nodes that monitored the `unreachable` node detects it as `reachable` again the cluster, after gossip dissemination, will consider it as `reachable`.

If system messages cannot be delivered to a node it will be quarantined and then it cannot come back from `unreachable`. This can happen if the there are too many unacknowledged system messages (e.g. watch, Terminated, remote actor deployment, failures of actors supervised by remote parent). Then the node needs to be

moved to the `down` or `removed` states (see the Membership Lifecycle section below) and the actor system must be restarted before it can join the cluster again.

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 just a role, any node can be the leader and it can change between convergence rounds. The `leader` is simply the first node in sorted order that is able to take the leadership role, where the preferred member states for a `leader` are `up` and `leaving` (see the Membership Lifecycle section 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. Currently `leader` actions are only triggered by receiving a new cluster state with gossip convergence.

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 after a configured time of unreachability.

Seed Nodes The seed nodes are configured contact points for new nodes joining the cluster. When a new node is started it sends a message to all seed nodes and then sends a join command to the seed node that answers first.

The seed nodes configuration value does not have any influence on the running cluster itself, it is only relevant for new nodes joining the cluster as it helps them to find contact points to send the join command to; a new member can send this command to any current member of the cluster, not only to the seed nodes.

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 this version to only push the actual state as needed.

Periodically, the default is every 1 second, each node chooses another random node to initiate a round of gossip with. If less than ½ of the nodes resides in the seen set (have seen the new state) then the cluster gossips 3 times instead of once every second. This adjusted gossip interval is a way to speed up the convergence process in the early dissemination phase after a state change.

The choice of node to gossip with is random but it is biased to towards nodes that might not have seen the current state version. During each round of gossip exchange when no convergence it uses a probability of 0.8 (configurable) to gossip to a node not part of the seen set, i.e. that probably has an older version of the state. Otherwise gossip to any random live node.

This biased selection is a way to speed up the convergence process in the late dissemination phase after a state change.

For clusters larger than 400 nodes (configurable, and suggested by empirical evidence) the 0.8 probability is gradually reduced to avoid overwhelming single stragglers with too many concurrent gossip requests. The gossip receiver also has a mechanism to protect itself from too many simultaneous gossip messages by dropping messages that have been enqueued in the mailbox for too long time.

While the cluster is in a converged state the gossiper only sends a small gossip status message containing the gossip version to the chosen node. As soon as there is a change to the cluster (meaning non-convergence) then it goes back to biased gossip again.

The recipient of the gossip state or the gossip status can use the gossip version (vector clock) to determine whether:

1. it has a newer version of the gossip state, in which case it sends that back to the gossiper
2. it has an outdated version of the state, in which case the recipient requests the current state from the gossiper by sending back its version of the gossip state
3. it has conflicting gossip versions, in which case the different versions are merged and sent back

If the recipient and the gossip have the same version then the gossip state is not sent or requested.

The periodic nature of the gossip has a nice batching effect of state changes, e.g. joining several nodes quickly after each other to one node will result in only one state change to be spread to other members in the cluster.

The gossip messages are serialized with protobuf and also gzipped to reduce payload size.

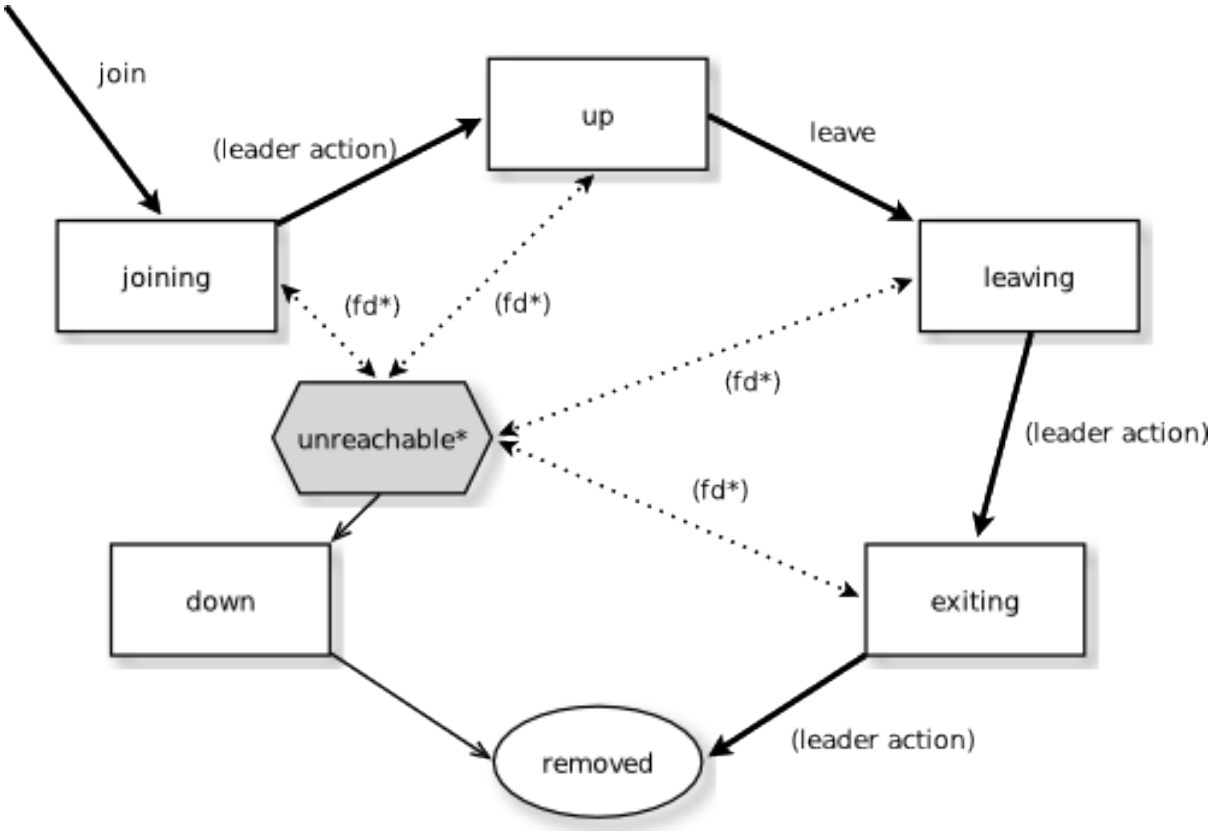
Membership Lifecycle

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`.

If a node is leaving the cluster in a safe, expected manner then it switches to the `leaving` state. Once the leader sees the convergence on the node in the `leaving` state, the leader will then move it to `exiting`. Once all nodes have seen the exiting state (convergence) the `leader` will remove the node from the cluster, marking it as `removed`.

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). To be able to move forward the state of the `unreachable` nodes must be changed. It must become `reachable` again or marked as `down`. If the node is to join the cluster again the actor system must be restarted and go through the joining process again. The cluster can, through the leader, also *auto-down* a node after a configured time of unreachability..

Note: If you have *auto-down* enabled and the failure detector triggers, you can over time end up with a lot of single node clusters if you don't put measures in place to shut down nodes that have become `unreachable`. This follows from the fact that the `unreachable` node will likely see the rest of the cluster as `unreachable`, become its own leader and form its own cluster.



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
- **down** marked as down (no longer part of cluster decisions)
- **removed** tombstone state (no longer a member)

User Actions

- **join** join a single node to a cluster - can be explicit or automatic on startup if a node to join have been specified in the configuration
- **leave** tell a node to leave the cluster gracefully
- **down** mark a node as down

Leader Actions The `leader` has the following duties:

- shifting members in and out of the cluster
  - joining -> up
  - exiting -> removed

Failure Detection and Unreachability

- **fd*** the failure detector of one of the monitoring nodes has triggered causing the monitored node to be marked as unreachable
- **unreachable*** unreachable is not a real member states but more of a flag in addition to the state signaling that the cluster is unable to talk to this node, after beeing unreachable the failure detector may detect it as reachable again and thereby remove the flag

5.1.2 Future Cluster Enhancements and Additions

Goal

In addition to membership also provide automatic partitioning *[*]*, handoff *[*]*, and cluster rebalancing *[*]* of actors.

Additional Terms

These additional terms are used in this section.

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).

Partitioning *[*]*

Note: Actor partitioning is not implemented yet.

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.

Additional Leader Responsibilities

After moving a member from joining to up, the leader can start assigning partitions *[*]* to the new node, and when a node is `leaving` 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.

On convergence the leader can schedule rebalancing across the cluster, 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.

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 persistent (durable) state 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.

Stateful Actor Replication *[*]*

Note: Stateful actor replication is not implemented yet.

[*] Not Implemented Yet

- Actor partitioning
- Actor handoff
- Actor rebalancing
- Stateful actor replication

5.2 Cluster Usage

For introduction to the Akka Cluster concepts please see *Cluster Specification*.

5.2.1 Preparing Your Project for Clustering

The Akka cluster is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-cluster" % "2.3.2"
```

5.2.2 A Simple Cluster Example

The following configuration enables the `Cluster` extension to be used. It joins the cluster and an actor subscribes to cluster membership events and logs them.

The `application.conf` configuration looks like this:

```
akka {
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
  }
  remote {
    log-remote-lifecycle-events = off
    netty.tcp {
      hostname = "127.0.0.1"
      port = 0
    }
  }

  cluster {
    seed-nodes = [
      "akka.tcp://ClusterSystem@127.0.0.1:2551",
      "akka.tcp://ClusterSystem@127.0.0.1:2552"]

    auto-down-unreachable-after = 10s
  }
}
```

To enable cluster capabilities in your Akka project you should, at a minimum, add the *Remoting* settings, but with `akka.cluster.ClusterActorRefProvider`. The `akka.cluster.seed-nodes` should normally also be added to your `application.conf` file.

The seed nodes are configured contact points for initial, automatic, join of the cluster.

Note that if you are going to start the nodes on different machines you need to specify the ip-addresses or host names of the machines in `application.conf` instead of `127.0.0.1`

An actor that uses the cluster extension may look like this:

```
package sample.cluster.simple

import akka.cluster.Cluster
import akka.cluster.ClusterEvent._
import akka.actor.ActorLogging
import akka.actor.Actor

class SimpleClusterListener extends Actor with ActorLogging {

  val cluster = Cluster(context.system)

  // subscribe to cluster changes, re-subscribe when restart
  override def preStart(): Unit = {
    //#subscribe
    cluster.subscribe(self, initialStateMode = InitialStateAsEvents,
      classOf[MemberEvent], classOf[UnreachableMember])
    //#subscribe
  }
  override def postStop(): Unit = cluster.unsubscribe(self)

  def receive = {
    case MemberUp(member) =>
      log.info("Member is Up: {}", member.address)
    case UnreachableMember(member) =>
      log.info("Member detected as unreachable: {}", member)
    case MemberRemoved(member, previousStatus) =>
      log.info("Member is Removed: {} after {}",
        member.address, previousStatus)
    case _: MemberEvent => // ignore
  }
}
```

The actor registers itself as subscriber of certain cluster events. It receives events corresponding to the current state of the cluster when the subscription starts and then it receives events for changes that happen in the cluster.

The easiest way to run this example yourself is to download Typesafe Activator and open the tutorial named Akka Cluster Samples with Scala. It contains instructions of how to run the `SimpleClusterApp`.

5.2.3 Joining to Seed Nodes

You may decide if joining to the cluster should be done manually or automatically to configured initial contact points, so-called seed nodes. When a new node is started it sends a message to all seed nodes and then sends join command to the one that answers first. If no one of the seed nodes replied (might not be started yet) it retries this procedure until successful or shutdown.

You define the seed nodes in the *Configuration* file (application.conf):

```
akka.cluster.seed-nodes = [
  "akka.tcp://ClusterSystem@host1:2552",
  "akka.tcp://ClusterSystem@host2:2552"]
```

This can also be defined as Java system properties when starting the JVM using the following syntax:

```
-Dakka.cluster.seed-nodes.0=akka.tcp://ClusterSystem@host1:2552
-Dakka.cluster.seed-nodes.1=akka.tcp://ClusterSystem@host2:2552
```

The seed nodes can be started in any order and it is not necessary to have all seed nodes running, but the node configured as the first element in the `seed-nodes` configuration list must be started when initially starting a cluster, otherwise the other seed-nodes will not become initialized and no other node can join the cluster. The reason for the special first seed node is to avoid forming separated islands when starting from an empty cluster. It is quickest to start all configured seed nodes at the same time (order doesn't matter), otherwise it can take up to the configured `seed-node-timeout` until the nodes can join.

Once more than two seed nodes have been started it is no problem to shut down the first seed node. If the first seed node is restarted it will first try join the other seed nodes in the existing cluster.

If you don't configure the seed nodes you need to join manually, using *JMX* or *Command Line Management*. You can join to any node in the cluster. It doesn't have to be configured as a seed node.

Joining can also be performed programatically with `Cluster(system).join`. Note that you can only join to an existing cluster member, which means that for bootstrapping some node must join itself.

You may also use `Cluster(system).joinSeedNodes`, which is attractive when dynamically discovering other nodes at startup by using some external tool or API. When using `joinSeedNodes` you should not include the node itself except for the node that is supposed to be the first seed node, and that should be placed first in parameter to `joinSeedNodes`.

Unsuccessful join attempts are automatically retried after the time period defined in configuration property `retry-unsuccessful-join-after`. When using `seed-nodes` this means that a new seed node is picked. When joining manually or programatically this means that the last join request is retried. Retries can be disabled by setting the property to `off`.

An actor system can only join a cluster once. Additional attempts will be ignored. When it has successfully joined it must be restarted to be able to join another cluster or to join the same cluster again. It can use the same host name and port after the restart, but it must have been removed from the cluster before the join request is accepted.

5.2.4 Automatic vs. Manual Downing

When a member is considered by the failure detector to be unreachable the leader is not allowed to perform its duties, such as changing status of new joining members to 'Up'. The node must first become reachable again, or the status of the unreachable member must be changed to 'Down'. Changing status to 'Down' can be performed automatically or manually. By default it must be done manually, using *JMX* or *Command Line Management*.

It can also be performed programatically with `Cluster(system).down(address)`.

You can enable automatic downing with configuration:

```
akka.cluster.auto-down-unreachable-after = 120s
```

This means that the cluster leader member will change the `unreachable` node status to `down` automatically after the configured time of unreachability.

Be aware of that using auto-down implies that two separate clusters will automatically be formed in case of network partition. That might be desired by some applications but not by others.

Note: If you have *auto-down* enabled and the failure detector triggers, you can over time end up with a lot of single node clusters if you don't put measures in place to shut down nodes that have become `unreachable`. This follows from the fact that the `unreachable` node will likely see the rest of the cluster as `unreachable`, become its own leader and form its own cluster.

5.2.5 Leaving

There are two ways to remove a member from the cluster.

You can just stop the actor system (or the JVM process). It will be detected as unreachable and removed after the automatic or manual downing as described above.

A more graceful exit can be performed if you tell the cluster that a node shall leave. This can be performed using *JMX* or *Command Line Management*. It can also be performed programatically with `Cluster(system).leave(address)`.

Note that this command can be issued to any member in the cluster, not necessarily the one that is leaving. The cluster extension, but not the actor system or JVM, of the leaving member will be shutdown after the leader has changed status of the member to *Exiting*. Thereafter the member will be removed from the cluster. Normally this

is handled automatically, but in case of network failures during this process it might still be necessary to set the node's status to `Down` in order to complete the removal.

5.2.6 Subscribe to Cluster Events

You can subscribe to change notifications of the cluster membership by using `Cluster(system).subscribe`.

```
cluster.subscribe(self, classOf[MemberEvent], classOf[UnreachableMember])
```

A snapshot of the full state, `akka.cluster.ClusterEvent.CurrentClusterState`, is sent to the subscriber as the first message, followed by events for incremental updates.

Note that you may receive an empty `CurrentClusterState`, containing no members, if you start the subscription before the initial join procedure has completed. This is expected behavior. When the node has been accepted in the cluster you will receive `MemberUp` for that node, and other nodes.

If you find it inconvenient to handle the `CurrentClusterState` you can use `ClusterEvent.InitialStateAsEvents` as parameter to `subscribe`. That means that instead of receiving `CurrentClusterState` as the first message you will receive the events corresponding to the current state to mimic what you would have seen if you were listening to the events when they occurred in the past. Note that those initial events only correspond to the current state and it is not the full history of all changes that actually has occurred in the cluster.

```
cluster.subscribe(self, initialStateMode = InitialStateAsEvents,
  classOf[MemberEvent], classOf[UnreachableMember])
```

The events to track the life-cycle of members are:

- `ClusterEvent.MemberUp` - A new member has joined the cluster and its status has been changed to `Up`.
- `ClusterEvent.MemberExited` - A member is leaving the cluster and its status has been changed to `Exiting` Note that the node might already have been shutdown when this event is published on another node.
- `ClusterEvent.MemberRemoved` - Member completely removed from the cluster.
- `ClusterEvent.UnreachableMember` - A member is considered as unreachable, detected by the failure detector of at least one other node.
- `ClusterEvent.ReachableMember` - A member is considered as reachable again, after having been unreachable. All nodes that previously detected it as unreachable has detected it as reachable again.

There are more types of change events, consult the API documentation of classes that extends `akka.cluster.ClusterEvent.ClusterDomainEvent` for details about the events.

Instead of subscribing to cluster events it can sometimes be convenient to only get the full membership state with `Cluster(system).state`. Note that this state is not necessarily in sync with the events published to a cluster subscription.

Worker Dial-in Example

Let's take a look at an example that illustrates how workers, here named *backend*, can detect and register to new master nodes, here named *frontend*.

The example application provides a service to transform text. When some text is sent to one of the frontend services, it will be delegated to one of the backend workers, which performs the transformation job, and sends the result back to the original client. New backend nodes, as well as new frontend nodes, can be added or removed to the cluster dynamically.

Messages:

```
case class TransformationJob(text: String)
case class TransformationResult(text: String)
case class JobFailed(reason: String, job: TransformationJob)
case object BackendRegistration
```

The backend worker that performs the transformation job:

```
class TransformationBackend extends Actor {

  val cluster = Cluster(context.system)

  // subscribe to cluster changes, MemberUp
  // re-subscribe when restart
  override def preStart(): Unit = cluster.subscribe(self, classOf[MemberUp])
  override def postStop(): Unit = cluster.unsubscribe(self)

  def receive = {
    case TransformationJob(text) => sender() ! TransformationResult(text.toUpperCase)
    case state: CurrentClusterState =>
      state.members.filter(_.status == MemberStatus.Up) foreach register
    case MemberUp(m) => register(m)
  }

  def register(member: Member): Unit =
    if (member.hasRole("frontend"))
      context.actorSelection(RootActorPath(member.address) / "user" / "frontend") !
        BackendRegistration
}
```

Note that the `TransformationBackend` actor subscribes to cluster events to detect new, potential, frontend nodes, and send them a registration message so that they know that they can use the backend worker.

The frontend that receives user jobs and delegates to one of the registered backend workers:

```
class TransformationFrontend extends Actor {

  var backends = IndexedSeq.empty[ActorRef]
  var jobCounter = 0

  def receive = {
    case job: TransformationJob if backends.isEmpty =>
      sender() ! JobFailed("Service unavailable, try again later", job)

    case job: TransformationJob =>
      jobCounter += 1
      backends(jobCounter % backends.size) forward job

    case BackendRegistration if !backends.contains(sender()) =>
      context watch sender()
      backends = backends :+ sender()

    case Terminated(a) =>
      backends = backends.filterNot(_ == a)
  }
}
```

Note that the `TransformationFrontend` actor watch the registered backend to be able to remove it from its list of available backend workers. Death watch uses the cluster failure detector for nodes in the cluster, i.e. it detects network failures and JVM crashes, in addition to graceful termination of watched actor. Death watch generates the `Terminated` message to the watching actor when the unreachable cluster node has been downed and removed.

The Typesafe Activator tutorial named Akka Cluster Samples with Scala. contains the full source code and instructions of how to run the **Worker Dial-in Example**.

5.2.7 Node Roles

Not all nodes of a cluster need to perform the same function: there might be one sub-set which runs the web front-end, one which runs the data access layer and one for the number-crunching. Deployment of actors—for example by cluster-aware routers—can take node roles into account to achieve this distribution of responsibilities.

The roles of a node is defined in the configuration property named `akka.cluster.roles` and it is typically defined in the start script as a system property or environment variable.

The roles of the nodes is part of the membership information in `MemberEvent` that you can subscribe to.

5.2.8 How To Startup when Cluster Size Reached

A common use case is to start actors after the cluster has been initialized, members have joined, and the cluster has reached a certain size.

With a configuration option you can define required number of members before the leader changes member status of 'Joining' members to 'Up'.

```
akka.cluster.min-nr-of-members = 3
```

In a similar way you can define required number of members of a certain role before the leader changes member status of 'Joining' members to 'Up'.

```
akka.cluster.role {
  frontend.min-nr-of-members = 1
  backend.min-nr-of-members = 2
}
```

You can start the actors in a `registerOnMemberUp` callback, which will be invoked when the current member status is changed tp 'Up', i.e. the cluster has at least the defined number of members.

```
Cluster(system) registerOnMemberUp {
  system.actorOf(Props(classOf[FactorialFrontend], upToN, true),
    name = "factorialFrontend")
}
```

This callback can be used for other things than starting actors.

5.2.9 Cluster Singleton

For some use cases it is convenient and sometimes also mandatory to ensure that you have exactly one actor of a certain type running somewhere in the cluster.

This can be implemented by subscribing to member events, but there are several corner cases to consider. Therefore, this specific use case is made easily accessible by the *Cluster Singleton* in the contrib module.

5.2.10 Cluster Sharding

Distributes actors across several nodes in the cluster and supports interaction with the actors using their logical identifier, but without having to care about their physical location in the cluster.

See *Cluster Sharding* in the contrib module.

5.2.11 Distributed Publish Subscribe

Publish-subscribe messaging between actors in the cluster, and point-to-point messaging using the logical path of the actors, i.e. the sender does not have to know on which node the destination actor is running.

See *Distributed Publish Subscribe in Cluster* in the contrib module.

5.2.12 Cluster Client

Communication from an actor system that is not part of the cluster to actors running somewhere in the cluster. The client does not have to know on which node the destination actor is running.

See *Cluster Client* in the contrib module.

5.2.13 Failure Detector

In a cluster each node is monitored by a few (default maximum 5) other nodes, and when any of these detects the node as `unreachable` that information will spread to the rest of the cluster through the gossip. In other words, only one node needs to mark a node `unreachable` to have the rest of the cluster mark that node `unreachable`.

The failure detector will also detect if the node becomes `reachable` again. When all nodes that monitored the `unreachable` node detects it as `reachable` again the cluster, after gossip dissemination, will consider it as `reachable`.

If system messages cannot be delivered to a node it will be quarantined and then it cannot come back from `unreachable`. This can happen if the there are too many unacknowledged system messages (e.g. watch, Terminated, remote actor deployment, failures of actors supervised by remote parent). Then the node needs to be moved to the `down` or `removed` states and the actor system must be restarted before it can join the cluster again.

The nodes in the cluster monitor each other by sending heartbeats to detect if a node is unreachable from the rest of the cluster. The heartbeat arrival times is interpreted by an implementation of The Phi Accrual Failure Detector.

The suspicion level of failure is given by a value called *phi*. The basic idea of the phi failure detector is to express the value of *phi* on a scale that is dynamically adjusted to reflect current network conditions.

The value of *phi* is calculated as:

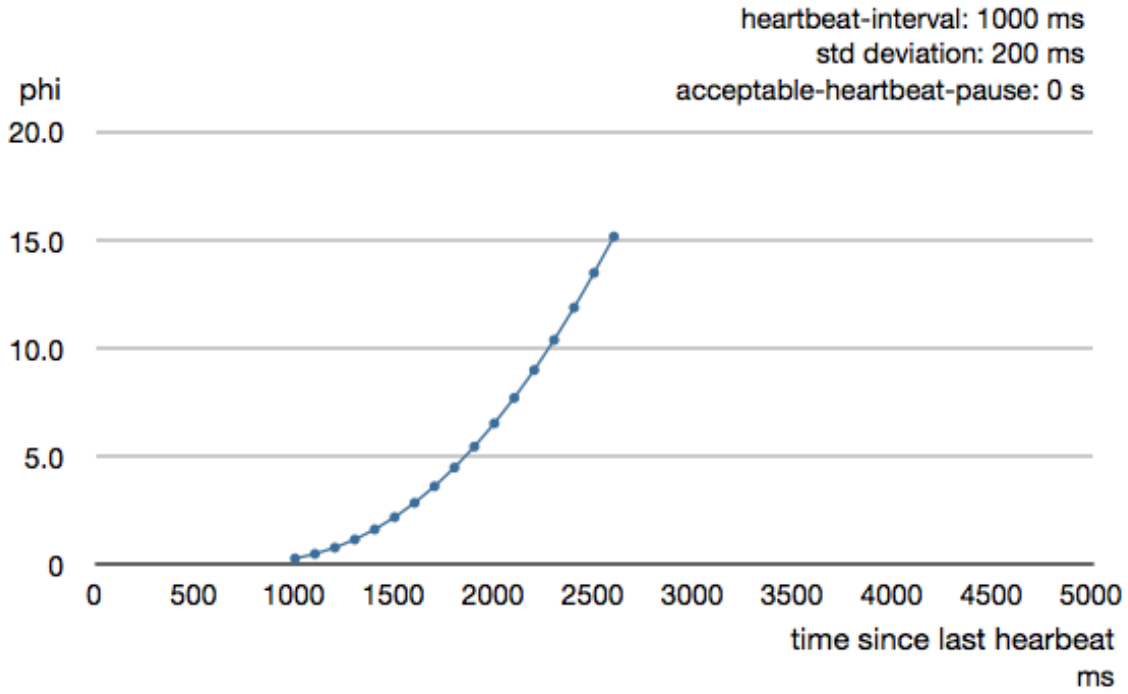
```
phi = -log10(1 - F(timeSinceLastHeartbeat))
```

where F is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat inter-arrival times.

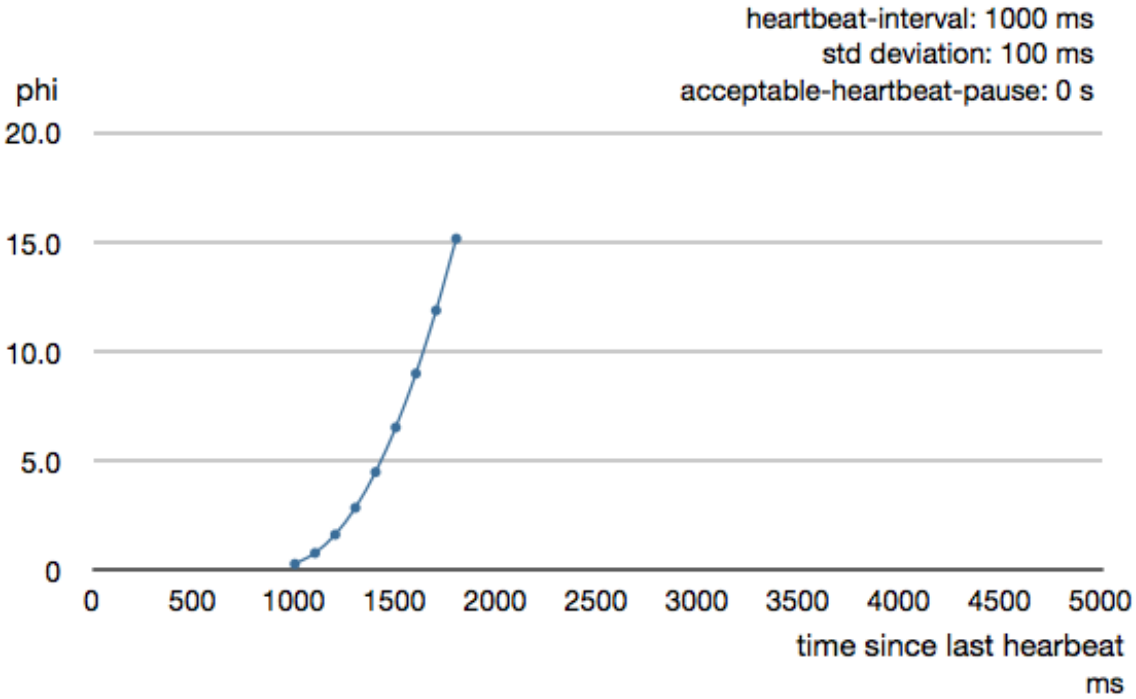
In the *Configuration* you can adjust the `akka.cluster.failure-detector.threshold` to define when a *phi* value is considered to be a failure.

A low `threshold` is prone to generate many false positives 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.

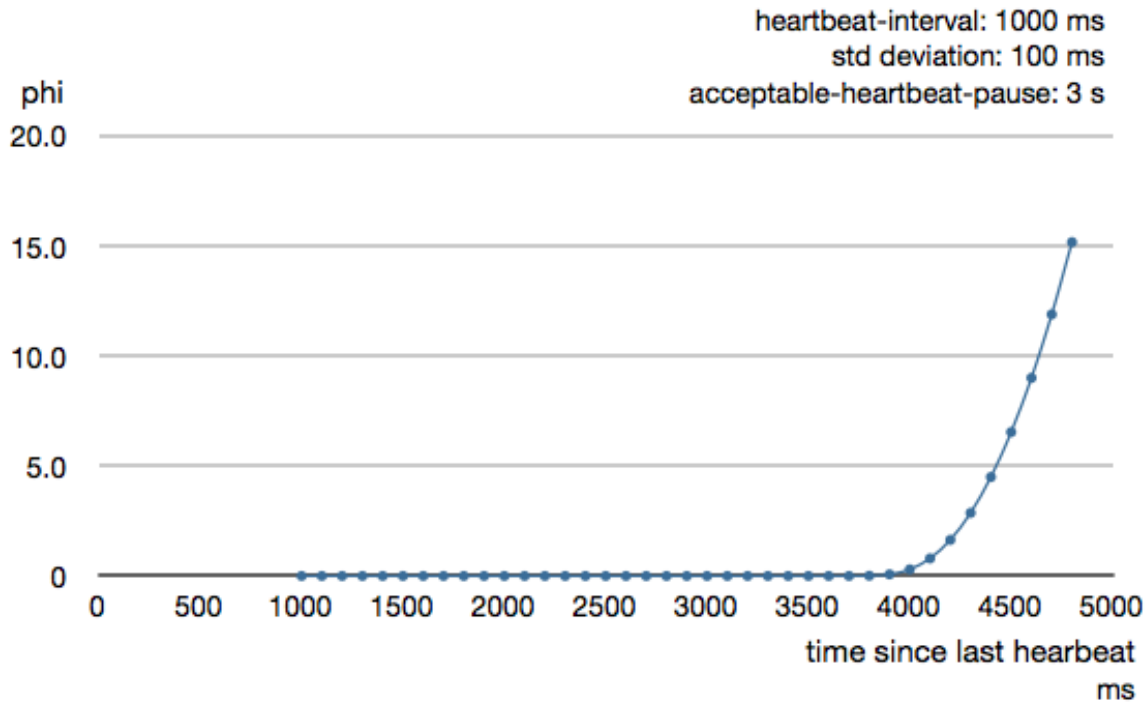
The following chart illustrates how *phi* increase with increasing time since the previous heartbeat.



Phi is calculated from the mean and standard deviation of historical inter arrival times. The previous chart is an example for standard deviation of 200 ms. If the heartbeats arrive with less deviation the curve becomes steeper, i.e. it is possible to determine failure more quickly. The curve looks like this for a standard deviation of 100 ms.



To be able to survive sudden abnormalities, such as garbage collection pauses and transient network failures the failure detector is configured with a margin, `akka.cluster.failure-detector.acceptable-heartbeat-pause`. You may want to adjust the *Configuration* of this depending on you environment. This is how the curve looks like for `acceptable-heartbeat-pause` configured to 3 seconds.



Death watch uses the cluster failure detector for nodes in the cluster, i.e. it detects network failures and JVM crashes, in addition to graceful termination of watched actor. Death watch generates the `Terminated` message to the watching actor when the unreachable cluster node has been downed and removed.

If you encounter suspicious false positives when the system is under load you should define a separate dispatcher for the cluster actors as described in *Cluster Dispatcher*.

5.2.14 Cluster Aware Routers

All *routers* can be made aware of member nodes in the cluster, i.e. deploying new routees or looking up routees on nodes in the cluster. When a node becomes unreachable or leaves the cluster the routees of that node are automatically unregistered from the router. When new nodes join the cluster additional routees are added to the router, according to the configuration. Routees are also added when a node becomes reachable again, after having been unreachable.

There are two distinct types of routers.

- **Group - router that sends messages to the specified path using actor selection** The routees can be shared between routers running on different nodes in the cluster. One example of a use case for this type of router is a service running on some backend nodes in the cluster and used by routers running on front-end nodes in the cluster.
- **Pool - router that creates routees as child actors and deploys them on remote nodes.** Each router will have its own routee instances. For example, if you start a router on 3 nodes in a 10 nodes cluster you will have 30 routee actors in total if the router is configured to use one inctance per node. The routees created by the different routers will not be shared between the routers. One example of a use case for this type of router is a single master that coordinate jobs and delegates the actual work to routees running on other nodes in the cluster.

Router with Group of Routees

When using a `Group` you must start the routee actors on the cluster member nodes. That is not done by the router. The configuration for a group looks like this:

```
akka.actor.deployment {
  /statsService/workerRouter {
      router = consistent-hashing-group
      nr-of-instances = 100
      routees.paths = ["/user/statsWorker"]
      cluster {
        enabled = on
        allow-local-routees = on
        use-role = compute
      }
    }
}
```

Note: The routee actors should be started as early as possible when starting the actor system, because the router will try to use them as soon as the member status is changed to 'Up'. If it is not available at that point it will be removed from the router and it will only re-try when the cluster members are changed.

It is the relative actor paths defined in `routees.paths` that identify what actor to lookup. It is possible to limit the lookup of routees to member nodes tagged with a certain role by specifying `use-role`.

`nr-of-instances` defines total number of routees in the cluster. Setting `nr-of-instances` to a high value will result in new routees added to the router when nodes join the cluster.

The same type of router could also have been defined in code:

```
import akka.cluster.routing.ClusterRouterGroup
import akka.cluster.routing.ClusterRouterGroupSettings
import akka.routing.ConsistentHashingGroup

val workerRouter = context.actorOf(
  ClusterRouterGroup(ConsistentHashingGroup(Nil), ClusterRouterGroupSettings(
    totalInstances = 100, routeesPaths = List("/user/statsWorker"),
    allowLocalRoutees = true, useRole = Some("compute"))).props(),
  name = "workerRouter2")
```

See *Configuration* section for further descriptions of the settings.

Router Example with Group of Routees

Let's take a look at how to use a cluster aware router with a group of routees, i.e. router sending to the paths of the routees.

The example application provides a service to calculate statistics for a text. When some text is sent to the service it splits it into words, and delegates the task to count number of characters in each word to a separate worker, a routee of a router. The character count for each word is sent back to an aggregator that calculates the average number of characters per word when all results have been collected.

Messages:

```
case class StatsJob(text: String)
case class StatsResult(meanWordLength: Double)
case class JobFailed(reason: String)
```

The worker that counts number of characters in each word:

```
class StatsWorker extends Actor {
  var cache = Map.empty[String, Int]
  def receive = {
    case word: String =>
      val length = cache.get(word) match {
        case Some(x) => x
        case None =>
```

```
          val x = word.length
          cache += (word -> x)
          x
      }

      sender() ! length
  }
}
```

The service that receives text from users and splits it up into words, delegates to workers and aggregates:

```
class StatsService extends Actor {
  // This router is used both with lookup and deploy of routees. If you
  // have a router with only lookup of routees you can use Props.empty
  // instead of Props[StatsWorker.class].
  val workerRouter = context.actorOf(FromConfig.props(Props[StatsWorker]),
    name = "workerRouter")

  def receive = {
    case StatsJob(text) if text != "" =>
      val words = text.split(" ")
      val replyTo = sender() // important to not close over sender()
      // create actor that collects replies from workers
      val aggregator = context.actorOf(Props(
        classOf[StatsAggregator], words.size, replyTo))
      words foreach { word =>
        workerRouter.tell(
          ConsistentHashableEnvelope(word, word), aggregator)
      }
  }
}

class StatsAggregator(expectedResults: Int, replyTo: ActorRef) extends Actor {
  var results = IndexedSeq.empty[Int]
  context.setReceiveTimeout(3.seconds)

  def receive = {
    case wordCount: Int =>
      results = results :+ wordCount
      if (results.size == expectedResults) {
        val meanWordLength = results.sum.toDouble / results.size
        replyTo ! StatsResult(meanWordLength)
        context.stop(self)
      }
    case ReceiveTimeout =>
      replyTo ! JobFailed("Service unavailable, try again later")
      context.stop(self)
  }
}
```

Note, nothing cluster specific so far, just plain actors.

All nodes start `StatsService` and `StatsWorker` actors. Remember, routees are the workers in this case. The router is configured with `routees.paths`:

```
akka.actor.deployment {
  /statsService/workerRouter {
    router = consistent-hashing-group
    nr-of-instances = 100
    routees.paths = ["/user/statsWorker"]
    cluster {
      enabled = on
      allow-local-routees = on
      use-role = compute
```

```
    }
  }
}
```

This means that user requests can be sent to `StatsService` on any node and it will use `StatsWorker` on all nodes.

The Typesafe Activator tutorial named Akka Cluster Samples with Scala. contains the full source code and instructions of how to run the **Router Example with Group of Routees**.

Router with Pool of Remote Deployed Routees

When using a `Pool` with routees created and deployed on the cluster member nodes the configuration for a router looks like this:

```
akka.actor.deployment {
  /singleton/statsService/workerRouter {
      router = consistent-hashing-pool
      nr-of-instances = 100
      cluster {
        enabled = on
        max-nr-of-instances-per-node = 3
        allow-local-routees = on
        use-role = compute
      }
    }
}
```

It is possible to limit the deployment of routees to member nodes tagged with a certain role by specifying `use-role`.

`nr-of-instances` defines total number of routees in the cluster, but the number of routees per node, `max-nr-of-instances-per-node`, will not be exceeded. Setting `nr-of-instances` to a high value will result in creating and deploying additional routees when new nodes join the cluster.

The same type of router could also have been defined in code:

```
import akka.cluster.routing.ClusterRouterPool
import akka.cluster.routing.ClusterRouterPoolSettings
import akka.routing.ConsistentHashingPool

val workerRouter = context.actorOf(
  ClusterRouterPool(ConsistentHashingPool(0), ClusterRouterPoolSettings(
    totalInstances = 100, maxInstancesPerNode = 3,
    allowLocalRoutees = false, useRole = None)).props(Props[StatsWorker]),
  name = "workerRouter3")
```

See *Configuration* section for further descriptions of the settings.

Router Example with Pool of Remote Deployed Routees

Let's take a look at how to use a cluster aware router on single master node that creates and deploys workers. To keep track of a single master we use the *Cluster Singleton* in the contrib module. The `ClusterSingletonManager` is started on each node.

```
system.actorOf(ClusterSingletonManager.props(
  singletonProps = Props[StatsService], singletonName = "statsService",
  terminationMessage = PoisonPill, role = Some("compute")),
  name = "singleton")
```

We also need an actor on each node that keeps track of where current single master exists and delegates jobs to the `StatsService`. That is provided by the `ClusterSingletonProxy`.

```
system.actorOf(ClusterSingletonProxy.props(singletonPath = "/user/singleton/statsService",
  role = Some("compute")), name = "statsServiceProxy")
```

The `ClusterSingletonProxy` receives text from users and delegates to the current `StatsService`, the single master. It listens to cluster events to lookup the `StatsService` on the oldest node.

All nodes start `ClusterSingletonProxy` and the `ClusterSingletonManager`. The router is now configured like this:

```
akka.actor.deployment {
  /singleton/statsService/workerRouter {
    router = consistent-hashing-pool
    nr-of-instances = 100
    cluster {
      enabled = on
      max-nr-of-instances-per-node = 3
      allow-local-routees = on
      use-role = compute
    }
  }
}
```

The Typesafe Activator tutorial named Akka Cluster Samples with Scala. contains the full source code and instructions of how to run the **Router Example with Pool of Remote Deployed Routees**.

5.2.15 Cluster Metrics

The member nodes of the cluster collects system health metrics and publishes that to other nodes and to registered subscribers. This information is primarily used for load-balancing routers.

Hyperic Sigar

The built-in metrics is gathered from JMX MBeans, and optionally you can use Hyperic Sigar for a wider and more accurate range of metrics compared to what can be retrieved from ordinary MBeans. Sigar is using a native OS library. To enable usage of Sigar you need to add the directory of the native library to `-Djava.libarary.path=<path_of_sigar_libs>` add the following dependency:

```
"org.fusesource" % "sigar" % "1.6.4"
```

Download the native Sigar libraries from Maven Central

Adaptive Load Balancing

The `AdaptiveLoadBalancingPool` / `AdaptiveLoadBalancingGroup` performs load balancing of messages to cluster nodes based on the cluster metrics data. It uses random selection of routees with probabilities derived from the remaining capacity of the corresponding node. It can be configured to use a specific MetricsSelector to produce the probabilities, a.k.a. weights:

- `heap` / `HeapMetricsSelector` - Used and max JVM heap memory. Weights based on remaining heap capacity; (max - used) / max
- `load` / `SystemLoadAverageMetricsSelector` - System load average for the past 1 minute, corresponding value can be found in `top` of Linux systems. The system is possibly nearing a bottleneck if the system load average is nearing number of cpus/cores. Weights based on remaining load capacity; 1 - (load / processors)
- `cpu` / `CpuMetricsSelector` - CPU utilization in percentage, sum of User + Sys + Nice + Wait. Weights based on remaining cpu capacity; 1 - utilization
- `mix` / `MixMetricsSelector` - Combines heap, cpu and load. Weights based on mean of remaining capacity of the combined selectors.

- Any custom implementation of `akka.cluster.routing.MetricsSelector`

The collected metrics values are smoothed with exponential weighted moving average. In the *Configuration* you can adjust how quickly past data is decayed compared to new data.

Let's take a look at this router in action. What can be more demanding than calculating factorials?

The backend worker that performs the factorial calculation:

```
class FactorialBackend extends Actor with ActorLogging {

  import context.dispatcher

  def receive = {
    case (n: Int) =>
      Future(factorial(n)) map { result => (n, result) } pipeTo sender()
  }

  def factorial(n: Int): BigInt = {
    @tailrec def factorialAcc(acc: BigInt, n: Int): BigInt = {
      if (n <= 1) acc
      else factorialAcc(acc * n, n - 1)
    }
    factorialAcc(BigInt(1), n)
  }

}
```

The frontend that receives user jobs and delegates to the backends via the router:

```
class FactorialFrontend(upToN: Int, repeat: Boolean) extends Actor with ActorLogging {

  val backend = context.actorOf(FromConfig.props(),
    name = "factorialBackendRouter")

  override def preStart(): Unit = {
    sendJobs()
    if (repeat) {
      context.setReceiveTimeout(10.seconds)
    }
  }

  def receive = {
    case (n: Int, factorial: BigInt) =>
      if (n == upToN) {
        log.debug("{}! = {}", n, factorial)
        if (repeat) sendJobs()
        else context.stop(self)
      }
    case ReceiveTimeout =>
      log.info("Timeout")
      sendJobs()
  }

  def sendJobs(): Unit = {
    log.info("Starting batch of factorials up to [{}]", upToN)
    1 to upToN foreach { backend ! _ }
  }
}
```

As you can see, the router is defined in the same way as other routers, and in this case it is configured as follows:

```
akka.actor.deployment {
  /factorialFrontend/factorialBackendRouter = {
    router = adaptive-group
```

```
    # metrics-selector = heap
    # metrics-selector = load
    # metrics-selector = cpu
    metrics-selector = mix
    nr-of-instances = 100
    routees.paths = ["/user/factorialBackend"]
    cluster {
      enabled = on
      use-role = backend
      allow-local-routees = off
    }
  }
}
```

It is only router type `adaptive` and the `metrics-selector` that is specific to this router, other things work in the same way as other routers.

The same type of router could also have been defined in code:

```
import akka.cluster.routing.ClusterRouterGroup
import akka.cluster.routing.ClusterRouterGroupSettings
import akka.cluster.routing.AdaptiveLoadBalancingGroup
import akka.cluster.routing.HeapMetricsSelector

val backend = context.actorOf(
  ClusterRouterGroup(AdaptiveLoadBalancingGroup(HeapMetricsSelector),
    ClusterRouterGroupSettings(
      totalInstances = 100, routeesPaths = List("/user/factorialBackend"),
      allowLocalRoutees = true, useRole = Some("backend"))).props(),
  name = "factorialBackendRouter2")
```

```
import akka.cluster.routing.ClusterRouterPool
import akka.cluster.routing.ClusterRouterPoolSettings
import akka.cluster.routing.AdaptiveLoadBalancingPool
import akka.cluster.routing.SystemLoadAverageMetricsSelector

val backend = context.actorOf(
  ClusterRouterPool(AdaptiveLoadBalancingPool(
    SystemLoadAverageMetricsSelector), ClusterRouterPoolSettings(
    totalInstances = 100, maxInstancesPerNode = 3,
    allowLocalRoutees = false, useRole = Some("backend"))).props(Props[FactorialBackend]),
  name = "factorialBackendRouter3")
```

The Typesafe Activator tutorial named Akka Cluster Samples with Scala. contains the full source code and instructions of how to run the **Adaptive Load Balancing** sample.

Subscribe to Metrics Events

It is possible to subscribe to the metrics events directly to implement other functionality.

```
import akka.cluster.Cluster
import akka.cluster.ClusterEvent.ClusterMetricsChanged
import akka.cluster.ClusterEvent.CurrentClusterState
import akka.cluster.NodeMetrics
import akka.cluster.StandardMetrics.HeapMemory
import akka.cluster.StandardMetrics.Cpu

class MetricsListener extends Actor with ActorLogging {
  val selfAddress = Cluster(context.system).selfAddress

  // subscribe to ClusterMetricsChanged
  // re-subscribe when restart
```

```
  override def preStart(): Unit =
    Cluster(context.system).subscribe(self, classOf[ClusterMetricsChanged])
  override def postStop(): Unit =
    Cluster(context.system).unsubscribe(self)

  def receive = {
    case ClusterMetricsChanged(clusterMetrics) =>
      clusterMetrics.filter(_.address == selfAddress) foreach { nodeMetrics =>
        logHeap(nodeMetrics)
        logCpu(nodeMetrics)
      }
    case state: CurrentClusterState => // ignore
  }

  def logHeap(nodeMetrics: NodeMetrics): Unit = nodeMetrics match {
    case HeapMemory(address, timestamp, used, committed, max) =>
      log.info("Used heap: {} MB", used.doubleValue / 1024 / 1024)
    case _ => // no heap info
  }

  def logCpu(nodeMetrics: NodeMetrics): Unit = nodeMetrics match {
    case Cpu(address, timestamp, Some(systemLoadAverage), cpuCombined, processors) =>
      log.info("Load: {} ({} processors)", systemLoadAverage, processors)
    case _ => // no cpu info
  }
}
```

Custom Metrics Collector

You can plug-in your own metrics collector instead of `akka.cluster.SigarMetricsCollector` or `akka.cluster.JmxMetricsCollector`. Look at those two implementations for inspiration. The implementation class can be defined in the *Configuration*.

5.2.16 How to Test

Multi Node Testing is useful for testing cluster applications.

Set up your project according to the instructions in *Multi Node Testing* and *Multi JVM Testing*, i.e. add the `sbt-multi-jvm` plugin and the dependency to `akka-multi-node-testkit`.

First, as described in *Multi Node Testing*, we need some scaffolding to configure the `MultiNodeSpec`. Define the participating roles and their *Configuration* in an object extending `MultiNodeConfig`:

```
import akka.remote.testkit.MultiNodeConfig
import com.typesafe.config.ConfigFactory

object StatsSampleSpecConfig extends MultiNodeConfig {
  // register the named roles (nodes) of the test
  val first = role("first")
  val second = role("second")
  val third = role("thrid")

  // this configuration will be used for all nodes
  // note that no fixed host names and ports are used
  commonConfig(ConfigFactory.parseString("""
    akka.actor.provider = "akka.cluster.ClusterActorRefProvider"
    akka.remote.log-remote-lifecycle-events = off
    akka.cluster.roles = [compute]
    # don't use sigar for tests, native lib not in path
    akka.cluster.metrics.collector-class = akka.cluster.JmxMetricsCollector
     // router lookup config ...
```

```
    """))

}
```

Define one concrete test class for each role/node. These will be instantiated on the different nodes (JVMs). They can be implemented differently, but often they are the same and extend an abstract test class, as illustrated here.

```
// need one concrete test class per node
class StatsSampleSpecMultiJvmNode1 extends StatsSampleSpec
class StatsSampleSpecMultiJvmNode2 extends StatsSampleSpec
class StatsSampleSpecMultiJvmNode3 extends StatsSampleSpec
```

Note the naming convention of these classes. The name of the classes must end with `MultiJvmNode1`, `MultiJvmNode2` and so on. It is possible to define another suffix to be used by the `sbt-multi-jvm`, but the default should be fine in most cases.

Then the abstract `MultiNodeSpec`, which takes the `MultiNodeConfig` as constructor parameter.

```
import org.scalatest.BeforeAndAfterAll
import org.scalatest.WordSpecLike
import org.scalatest.Matchers
import akka.remote.testkit.MultiNodeSpec
import akka.testkit.ImplicitSender

abstract class StatsSampleSpec extends MultiNodeSpec(StatsSampleSpecConfig)
  with WordSpecLike with Matchers with BeforeAndAfterAll
  with ImplicitSender {

  import StatsSampleSpecConfig._

  override def initialParticipants = roles.size

  override def beforeAll() = multiNodeSpecBeforeAll()

  override def afterAll() = multiNodeSpecAfterAll()
```

Most of this can of course be extracted to a separate trait to avoid repeating this in all your tests.

Typically you begin your test by starting up the cluster and let the members join, and create some actors. That can be done like this:

```
"illustrate how to startup cluster" in within(15 seconds) {
  Cluster(system).subscribe(testActor, classOf[MemberUp])
  expectMsgClass(classOf[CurrentClusterState])

  val firstAddress = node(first).address
  val secondAddress = node(second).address
  val thirdAddress = node(third).address

  Cluster(system) join firstAddress

  system.actorOf(Props[StatsWorker], "statsWorker")
  system.actorOf(Props[StatsService], "statsService")

  receiveN(3).collect { case MemberUp(m) => m.address }.toSet should be(
    Set(firstAddress, secondAddress, thirdAddress))

  Cluster(system).unsubscribe(testActor)

  testConductor.enter("all-up")
}
```

From the test you interact with the cluster using the `Cluster` extension, e.g. `join`.

```
Cluster(system) join firstAddress
```

Notice how the *testActor* from *testkit* is added as *subscriber* to cluster changes and then waiting for certain events, such as in this case all members becoming 'Up'.

The above code was running for all roles (JVMs). `runOn` is a convenient utility to declare that a certain block of code should only run for a specific role.

```
"show usage of the statsService from one node" in within(15 seconds) {
  runOn(second) {
    assertServiceOk()
  }

  testConductor.enter("done-2")
}

def assertServiceOk(): Unit = {
  val service = system.actorSelection(node(third) / "user" / "statsService")
  // eventually the service should be ok,
  // first attempts might fail because worker actors not started yet
  awaitAssert {
    service ! StatsJob("this is the text that will be analyzed")
    expectMsgType[StatsResult](1.second).meanWordLength should be(
      3.875 +- 0.001)
  }

}
```

Once again we take advantage of the facilities in *testkit* to verify expected behavior. Here using `testActor` as sender (via `ImplicitSender`) and verifing the reply with `expectMsgPF`.

In the above code you can see `node(third)`, which is useful facility to get the root actor reference of the actor system for a specific role. This can also be used to grab the `akka.actor.Address` of that node.

```
val firstAddress = node(first).address
val secondAddress = node(second).address
val thirdAddress = node(third).address
```

5.2.17 JMX

Information and management of the cluster is available as JMX MBeans with the root name `akka.Cluster`. The JMX information can be displayed with an ordinary JMX console such as JConsole or JVisualVM.

From JMX you can:

- see what members that are part of the cluster
- see status of this node
- join this node to another node in cluster
- mark any node in the cluster as down
- tell any node in the cluster to leave

Member nodes are identified by their address, in format *akka.<protocol>://<actor-system-name>@<hostname>:<port>*.

5.2.18 Command Line Management

The cluster can be managed with the script *bin/akka-cluster* provided in the Akka distribution.

Run it without parameters to see instructions about how to use the script:

```
Usage: bin/akka-cluster <node-hostname> <jmx-port> <command> ...

Supported commands are:
           join <node-url> - Sends request a JOIN node with the specified URL
          leave <node-url> - Sends a request for node with URL to LEAVE the cluster
           down <node-url> - Sends a request for marking node with URL as DOWN
             member-status - Asks the member node for its current status
                   members - Asks the cluster for addresses of current members
               unreachable - Asks the cluster for addresses of unreachable members
            cluster-status - Asks the cluster for its current status (member ring,
                             unavailable nodes, meta data etc.)
                    leader - Asks the cluster who the current leader is
              is-singleton - Checks if the cluster is a singleton cluster (single
                             node cluster)
             is-available - Checks if the member node is available
Where the <node-url> should be on the format of
  'akka.<protocol>://<actor-system-name>@<hostname>:<port>'

Examples: bin/akka-cluster localhost 9999 is-available
          bin/akka-cluster localhost 9999 join akka.tcp://MySystem@darkstar:2552
          bin/akka-cluster localhost 9999 cluster-status
```

To be able to use the script you must enable remote monitoring and management when starting the JVMs of the cluster nodes, as described in Monitoring and Management Using JMX Technology

Example of system properties to enable remote monitoring and management:

```
java -Dcom.sun.management.jmxremote.port=9999 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false
```

5.2.19 Configuration

There are several configuration properties for the cluster. We refer to the *reference configuration* for more information.

Cluster Info Logging

You can silence the logging of cluster events at info level with configuration property:

```
akka.cluster.log-info = off
```

Cluster Dispatcher

Under the hood the cluster extension is implemented with actors and it can be necessary to create a bulkhead for those actors to avoid disturbance from other actors. Especially the heartbeating actors that is used for failure detection can generate false positives if they are not given a chance to run at regular intervals. For this purpose you can define a separate dispatcher to be used for the cluster actors:

```
akka.cluster.use-dispatcher = cluster-dispatcher

cluster-dispatcher {
  type = "Dispatcher"
  executor = "fork-join-executor"
  fork-join-executor {
    parallelism-min = 2
    parallelism-max = 4
  }
}
```

5.3 Remoting

For an introduction of remoting capabilities of Akka please see *Location Transparency*.

Note: As explained in that chapter Akka remoting is designed for communication in a peer-to-peer fashion and it has limitations for client-server setups. In particular Akka Remoting does not work with Network Address Translation and Load Balancers, among others.

5.3.1 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" % "2.3.2"
```

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 {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      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 networking 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. All settings are described in *Remote Configuration*.

5.3.2 Types of Remote Interaction

Akka has two ways of using remoting:

- Lookup : used to look up an actor on a remote node with `actorSelection(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.

5.3.3 Looking up Remote Actors

`actorSelection(path)` will obtain an `ActorSelection` to an Actor on a remote node, e.g.:

```
val selection =
  context.actorSelection("akka.tcp://actorSystemName@10.0.0.1:2552/user/actorName")
```

As you can see from the example above the following pattern is used to find an actor on a remote node:

```
akka.<protocol>://<actor system>@<hostname>:<port>/<actor path>
```

Once you obtained a selection to the actor you can interact with it they same way you would with a local actor, e.g.:

```
selection ! "Pretty awesome feature"
```

To acquire an `ActorRef` for an `ActorSelection` you need to send a message to the selection and use the `sender` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `ActorRef`. This can also be done with the `resolveOne` method of the `ActorSelection`, which returns a `Future` of the matching `ActorRef`.

Note: For more details on how actor addresses and paths are formed and used, please refer to *Actor References, Paths and Addresses*.

5.3.4 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.tcp://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:

```
val actor = system.actorOf(Props[SampleActor], "sampleActor")
actor ! "Pretty slick"
```

The actor class `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.

Note: In order to ensure serializability of `Props` when passing constructor arguments to the actor being created, do not make the factory an inner class: this will inherently capture a reference to its enclosing object, which in most cases is not serializable. It is best to create a factory method in the companion object of the actor's class.

Serializability of all Props can be tested by setting the configuration item `akka.actor.serialize-creators=on`. Only Props whose `deploy` has `LocalScope` are exempt from this check.

Note: You can use asterisks as wildcard matches for the actor paths, so you could specify: `/*/sampleActor` and that would match all `sampleActor` on that level in the hierarchy. You can also use wildcard in the last position to match all actors at a certain level: `/someParent/*`. Non-wildcard matches always have higher priority to match than wildcards, so: `/foo/bar` is considered **more specific** than `/foo/*` and only the highest priority match is used. Please note that it **cannot** be used to partially match section, like this: `/foo*/bar`, `/f*o/bar` etc.

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
```

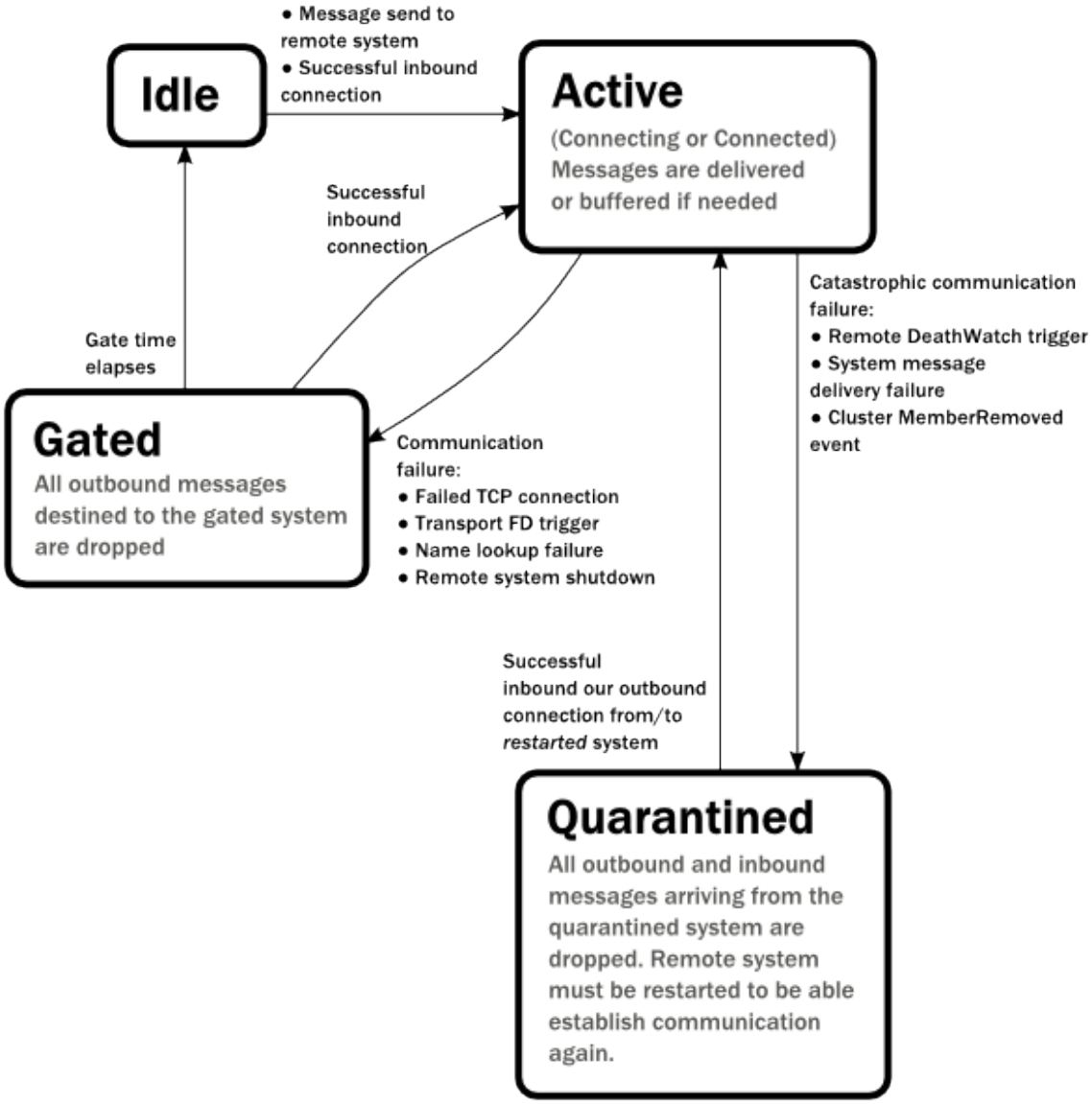
and a remote address like this:

```
val one = AddressFromURIString("akka.tcp://sys@host:1234")
val two = Address("akka.tcp", "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[SampleActor].
  withDeploy(Deploy(scope = RemoteScope(address))))
```

5.3.5 Lifecycle and Failure Recovery Model



Each link with link with a remote system can be in one of the four states as illustrated above. Before any communication happens with a remote system at a given `Address` the state of the association is `Idle`. The first time a message is attempted to be sent to the remote system or an inbound connection is accepted the state of the link transitions to `Active` denoting that the two systems has messages to send or receive and no failures were encountered so far. When a communication failure happens and the connection is lost between the two systems the link becomes `Gated`.

In this state the system will not attempt to connect to the remote host and all outbound messages will be dropped. The time while the link is in the `Gated` state is controlled by the setting `akka.remote.retry-gate-closed-for`: after this time elapses the link state transitions to `Idle` again. `Gate` is one-sided in the sense that whenever a successful *inbound* connection is accepted from a remote system during `Gate` it automatically transitions to `Active` and communication resumes immediately.

In the face of communication failures that are unrecoverable because the state of the participating systems are inconsistent, the remote system becomes `Quarantined`. Unlike `Gate`, quarantining is permanent and lasts until one of the systems is restarted. After a restart communication can be resumed again and the link can become `Active` again.

5.3.6 Watching Remote Actors

Watching a remote actor is not different than watching a local actor, as described in *Lifecycle Monitoring aka DeathWatch*.

Warning: *Caveat:* Watching an `ActorRef` acquired with `actorFor` does not trigger `Terminated` for lost connections. `actorFor` is deprecated in favor of `actorSelection`. Acquire the `ActorRef` to watch with `Identify` and `ActorIdentity` as described in *Identifying Actors via Actor Selection*.

Failure Detector

Under the hood remote death watch uses heartbeat messages and a failure detector to generate `Terminated` message from network failures and JVM crashes, in addition to graceful termination of watched actor.

The heartbeat arrival times is interpreted by an implementation of The Phi Accrual Failure Detector.

The suspicion level of failure is given by a value called *phi*. The basic idea of the phi failure detector is to express the value of *phi* on a scale that is dynamically adjusted to reflect current network conditions.

The value of *phi* is calculated as:

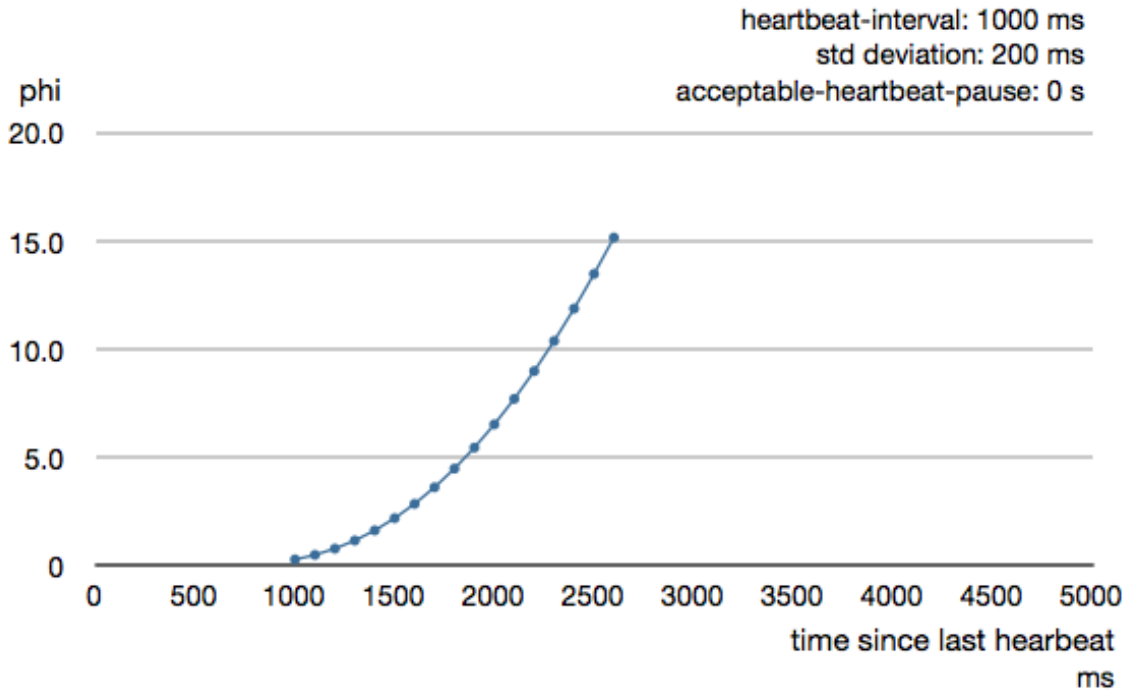
```
phi = -log10(1 - F(timeSinceLastHeartbeat))
```

where F is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat inter-arrival times.

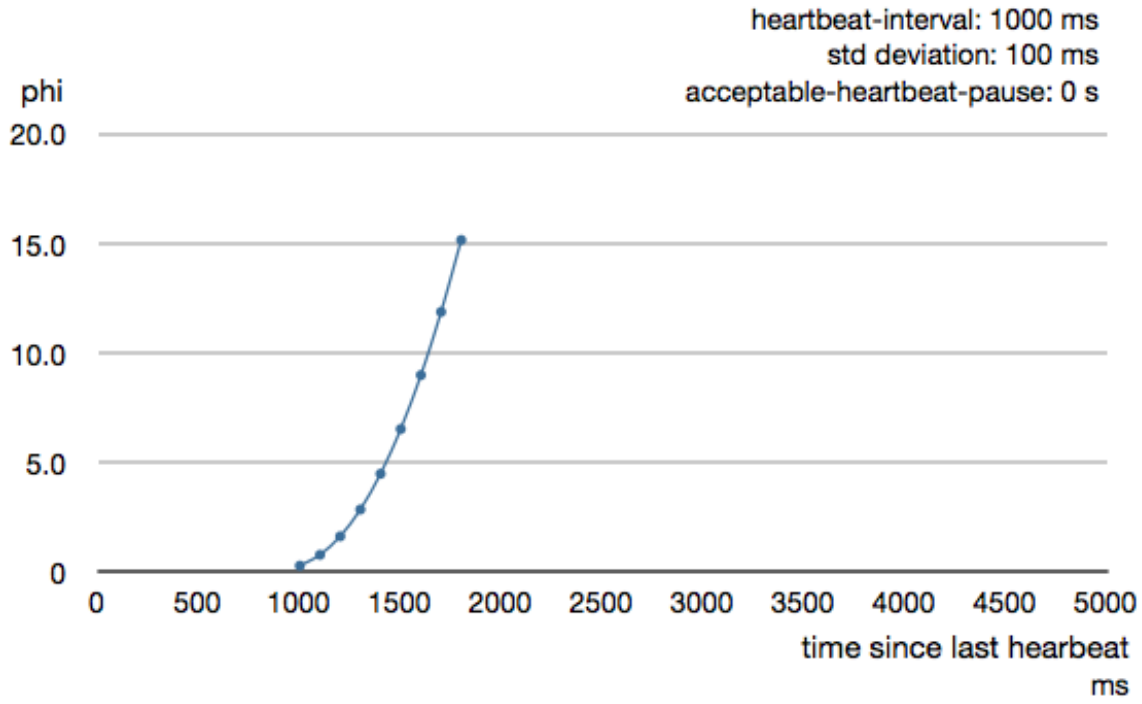
In the *Remote Configuration* you can adjust the `akka.remote.watch-failure-detector.threshold` to define when a *phi* value is considered to be a failure.

A low `threshold` is prone to generate many false positives 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 10 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.

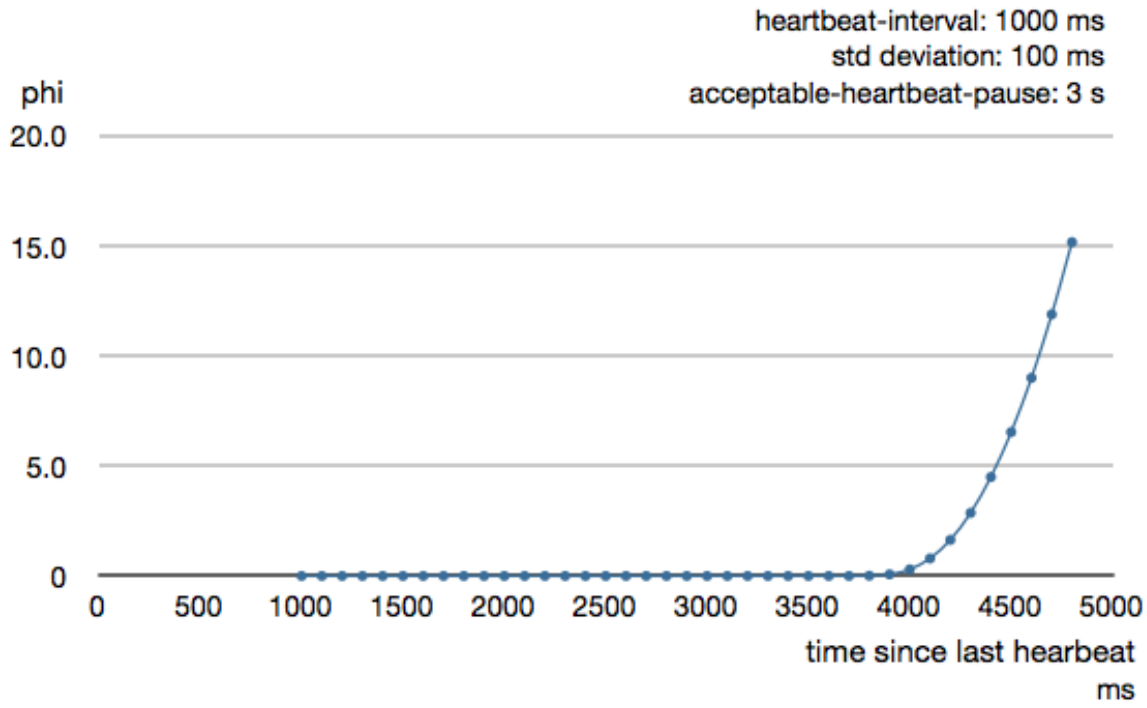
The following chart illustrates how *phi* increase with increasing time since the previous heartbeat.



Phi is calculated from the mean and standard deviation of historical inter arrival times. The previous chart is an example for standard deviation of 200 ms. If the heartbeats arrive with less deviation the curve becomes steeper, i.e. it is possible to determine failure more quickly. The curve looks like this for a standard deviation of 100 ms.



To be able to survive sudden abnormalities, such as garbage collection pauses and transient network failures the failure detector is configured with a margin, `akka.remote.watch-failure-detector.acceptable-heartbeat-pause`. You may want to adjust the *Remote Configuration* of this depending on you environment. This is how the curve looks like for `acceptable-heartbeat-pause` configured to 3 seconds.



5.3.7 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*

5.3.8 Routers with Remote Destinations

It is absolutely feasible to combine remoting with *Routing*.

A pool of remote deployed routees can be configured as:

```
akka.actor.deployment {
  /parent/remotePool {
    router = round-robin-pool
    nr-of-instances = 10
    target.nodes = ["akka.tcp://app@10.0.0.2:2552", "akka://app@10.0.0.3:2552"]
  }
}
```

This configuration setting will clone the actor defined in the `Props` of the `remotePool` 10 times and deploy it evenly distributed across the two given target nodes.

A group of remote actors can be configured as:

```
akka.actor.deployment {
  /parent/remoteGroup {
    router = round-robin-group
    routees.paths = [
      "akka.tcp://app@10.0.0.1:2552/user/workers/w1",
      "akka.tcp://app@10.0.0.2:2552/user/workers/w1",
      "akka.tcp://app@10.0.0.3:2552/user/workers/w1"]
  }
}
```

This configuration setting will send messages to the defined remote actor paths. It requires that you create the destination actors on the remote nodes with matching paths. That is not done by the router.

5.3.9 Remoting Sample

There is a more extensive remote example that comes with Typesafe Activator. The tutorial named Akka Remote Samples with Scala demonstrates both remote deployment and look-up of remote actors.

Pluggable transport support

Akka can be configured to use various transports to communicate with remote systems. The core component of this feature is the `akka.remote.Transport` SPI. Transport implementations must extend this trait. Transports can be loaded by setting the `akka.remote.enabled-transports` configuration key to point to one or more configuration sections containing driver descriptions.

An example of setting up the default Netty based SSL driver as default:

```
akka {
  remote {
    enabled-transports = [akka.remote.netty.ssl]

    netty.ssl.security {
      key-store = "mykeystore"
      trust-store = "mytruststore"
      key-store-password = "changeme"
```

```
      key-password = "changeme"
      trust-store-password = "changeme"
      protocol = "TLSv1"
      random-number-generator = "AES128CounterSecureRNG"
      enabled-algorithms = [TLS_RSA_WITH_AES_128_CBC_SHA]
    }
  }
}
```

An example of setting up a custom transport implementation:

```
akka {
  remote {
    applied-transports = ["akka.remote.mytransport"]

    mytransport {
      # The transport-class configuration entry is required, and
      # it must contain the fully qualified name of the transport
      # implementation
      transport-class = "my.package.MyTransport"

      # It is possible to decorate Transports with additional services.
      # Adapters should be registered in the "adapters" sections to
      # be able to apply them to transports
      applied-adapters = []

      # Driver specific configuration options has to be in the same
      # section:
      some-config = foo
      another-config = bar
    }
```

Remote Events

It is possible to listen to events that occur in Akka Remote, and to subscribe/unsubscribe to these events you simply register as listener to the below described types in on the `ActorSystem.eventStream`.

Note: To subscribe to any remote event, subscribe to `RemotingLifecycleEvent`. To subscribe to events related only to the lifecycle of associations, subscribe to `akka.remote.AssociationEvent`.

Note: The use of term "Association" instead of "Connection" reflects that the remoting subsystem may use connectionless transports, but an association similar to transport layer connections is maintained between endpoints by the Akka protocol.

By default an event listener is registered which logs all of the events described below. This default was chosen to help setting up a system, but it is quite common to switch this logging off once that phase of the project is finished.

Note: In order to switch off the logging, set `akka.remote.log-remote-lifecycle-events = off` in your `application.conf`.

To be notified when an association is over ("disconnected") listen to `DisassociatedEvent` which holds the direction of the association (inbound or outbound) and the addresses of the involved parties.

To be notified when an association is successfully established ("connected") listen to `AssociatedEvent` which holds the direction of the association (inbound or outbound) and the addresses of the involved parties.

To intercept errors directly related to associations, listen to `AssociationErrorEvent` which holds the direction of the association (inbound or outbound), the addresses of the involved parties and the `Throwable` cause.

To be notified when the remoting subsystem is ready to accept associations, listen to `RemotingListenEvent` which contains the addresses the remoting listens on.

To be notified when the remoting subsystem has been shut down, listen to `RemotingShutdownEvent`.

To intercept generic remoting related errors, listen to `RemotingErrorEvent` which holds the `Throwable` cause.

5.3.10 Remote Security

Akka provides a couple of ways to enhance security between remote nodes (client/server):

- Untrusted Mode
- Security Cookie Handshake

Untrusted Mode

As soon as an actor system can connect to another remotely, it may in principle send any possible message to any actor contained within that remote system. One example may be sending a `PoisonPill` to the system guardian, shutting that system down. This is not always desired, and it can be disabled with the following setting:

```
akka.remote.untrusted-mode = on
```

This disallows sending of system messages (actor life-cycle commands, DeathWatch, etc.) and any message extending `PossiblyHarmful` to the system on which this flag is set. Should a client send them nonetheless they are dropped and logged (at DEBUG level in order to reduce the possibilities for a denial of service attack). `PossiblyHarmful` covers the predefined messages like `PoisonPill` and `Kill`, but it can also be added as a marker trait to user-defined messages.

Messages sent with actor selection are by default discarded in untrusted mode, but permission to receive actor selection messages can be granted to specific actors defined in configuration:

```
akka.remote.trusted-selection-paths = ["/user/receptionist", "/user/namingService"]
```

The actual message must still not be of type `PossiblyHarmful`.

In summary, the following operations are ignored by a system configured in untrusted mode when incoming via the remoting layer:

- remote deployment (which also means no remote supervision)
- remote DeathWatch
- `system.stop()`, `PoisonPill`, `Kill`
- sending any message which extends from the `PossiblyHarmful` marker interface, which includes `Terminated`
- messages sent with actor selection, unless destination defined in `trusted-selection-paths`.

Note: Enabling the untrusted mode does not remove the capability of the client to freely choose the target of its message sends, which means that messages not prohibited by the above rules can be sent to any actor in the remote system. It is good practice for a client-facing system to only contain a well-defined set of entry point actors, which then forward requests (possibly after performing validation) to another actor system containing the actual worker actors. If messaging between these two server-side systems is done using local `ActorRef` (they can be exchanged safely between actor systems within the same JVM), you can restrict the messages on this interface by marking them `PossiblyHarmful` so that a client cannot forge them.

Secure Cookie Handshake

Akka remoting also allows you to specify a secure cookie that will be exchanged and ensured to be identical in the connection handshake between the client and the server. If they are not identical then the client will be refused to connect to the server.

The secure cookie can be any kind of string. But the recommended approach is to generate a cryptographically secure cookie using this script `$AKKA_HOME/scripts/generate_config_with_secure_cookie.sh` or from code using the `akka.util.Crypt.generateSecureCookie()` utility method.

You have to ensure that both the connecting client and the server have the same secure cookie as well as the `require-cookie` option turned on.

Here is an example config:

```
akka.remote {
  secure-cookie = "090A030E0F0A05010900000A0C0E0C0B03050D05"
  require-cookie = on
}
```

SSL

SSL can be used as the remote transport by adding `akka.remote.netty.ssl` to the `enabled-transport` configuration section. See a description of the settings in the *Remote Configuration* section.

The SSL support is implemented with Java Secure Socket Extension, please consult the offical Java Secure Socket Extension documentation and related resources for troubleshooting.

Note: When using SHA1PRNG on Linux it's recommended specify `-Djava.security.egd=file:/dev/./urandom` as argument to the JVM to prevent blocking. It is NOT as secure because it reuses the seed. Use '/dev/./urandom', not '/dev/urandom' as that doesn't work according to Bug ID: 6202721.

5.3.11 Remote Configuration

There are lots of configuration properties that are related to remoting in Akka. We refer to the *reference configuration* for more information.

Note: Setting properties like the listening IP and port number programmatically is best done by using something like the following:

```
ConfigFactory.parseString("akka.remote.netty.tcp.hostname=\"1.2.3.4\"")
    .withFallback(ConfigFactory.load());
```

5.4 Serialization

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.

5.4.1 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:

```
akka {
  actor {
    serializers {
      java = "akka.serialization.JavaSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      myown = "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:

```
akka {
  actor {
    serializers {
      java = "akka.serialization.JavaSerializer"
      proto = "akka.remote.serialization.ProtobufSerializer"
      myown = "docs.serialization.MyOwnSerializer"
    }

    serialization-bindings {
      "java.lang.String" = java
      "docs.serialization.Customer" = java
      "com.google.protobuf.Message" = proto
      "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:

```
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:

```
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 should be(original)
```

For more information, have a look at the `ScalaDoc` for `akka.serialization._`

5.4.2 Customization

So, lets say that you want to create your own `Serializer`, you saw the `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. 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 `Serialization.serializedActorPath(actorRef)` like this:

```
import akka.actor.{ ActorRef, ActorSystem }
import akka.serialization._
import com.typesafe.config.ConfigFactory

    // Serialize
    // (beneath toBinary)
    val identifier: String = Serialization.serializedActorPath(theActorRef)

    // Then just serialize the identifier however you like

    // Deserialize
    // (beneath fromBinary)
    val deserializedActorRef = extendedSystem.provider.resolveActorRef(identifier)
    // Then just use the ActorRef
```

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, 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. To find out the appropriate address to use when sending to `remoteAddr` you can use `ActorRefProvider.getExternalAddressFor(remoteAddr)` like this:

```
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.toSerializationFormatWithAddress(ExternalAddress(extendedSystem).
    addressFor(remote))
```

Note: `ActorPath.toSerializationFormatWithAddress` differs from `toString` if the address does not already have `host` and `port` components, i.e. it only inserts address information for local addresses.

`toSerializationFormatWithAddress` also adds the unique id of the actor, which will change when the actor is stopped and then created again with the same name. Sending messages to a reference pointing the old actor will not be delivered to the new actor. If you don't want this behavior, e.g. in case of long term storage of the reference, you can use `toStringWithAddress`, which doesn't include the unique id.

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 also a default remote address which is the one used by cluster support (and typical systems have just this one); you can get it like this:

```
object ExternalAddress extends ExtensionKey[ExternalAddressExt]

class ExternalAddressExt(system: ExtendedActorSystem) extends Extension {
  def addressForAkka: Address = system.provider.getDefaultAddress
}

def serializeAkkaDefault(ref: ActorRef): String =
  ref.path.toSerializationFormatWithAddress(ExternalAddress(theActorSystem).
    addressForAkka)
```

Deep serialization of Actors

The recommended approach to do deep serialization of internal actor state is to use Akka *Persistence*.

5.4.3 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.

5.4.4 External Akka Serializers

Akka-protostuff by Roman Levenstein

Akka-quickser by Roman Levenstein

Akka-kryo by Roman Levenstein

5.5 I/O

5.5.1 Introduction

The `akka.io` package has been developed in collaboration between the Akka and spray.io teams. Its design combines experiences from the `spray-io` module with improvements that were jointly developed for more general consumption as an actor-based service.

The guiding design goal for this I/O implementation was to reach extreme scalability, make no compromises in providing an API correctly matching the underlying transport mechanism and to be fully event-driven, non-blocking and asynchronous. The API is meant to be a solid foundation for the implementation of network protocols and building higher abstractions; it is not meant to be a full-service high-level NIO wrapper for end users.

5.5.2 Terminology, Concepts

The I/O API is completely actor based, meaning that all operations are implemented with message passing instead of direct method calls. Every I/O driver (TCP, UDP) has a special actor, called a *manager* that serves as an entry point for the API. I/O is broken into several drivers. The manager for a particular driver is accessible through the `IO` entry point. For example the following code looks up the TCP manager and returns its `ActorRef`:

```
import akka.io.{ IO, Tcp }
import context.system // implicitly used by IO(Tcp)

val manager = IO(Tcp)
```

The manager receives I/O command messages and instantiates worker actors in response. The worker actors present themselves to the API user in the reply to the command that was sent. For example after a `Connect` command sent to the TCP manager the manager creates an actor representing the TCP connection. All operations related to the given TCP connections can be invoked by sending messages to the connection actor which announces itself by sending a `Connected` message.

DeathWatch and Resource Management

I/O worker actors receive commands and also send out events. They usually need a user-side counterpart actor listening for these events (such events could be inbound connections, incoming bytes or acknowledgements for writes). These worker actors *watch* their listener counterparts. If the listener stops then the worker will automatically release any resources that it holds. This design makes the API more robust against resource leaks.

Thanks to the completely actor based approach of the I/O API the opposite direction works as well: a user actor responsible for handling a connection can watch the connection actor to be notified if it unexpectedly terminates.

Write models (Ack, Nack)

I/O devices have a maximum throughput which limits the frequency and size of writes. When an application tries to push more data than a device can handle, the driver has to buffer bytes until the device is able to write them. With buffering it is possible to handle short bursts of intensive writes — but no buffer is infinite. "Flow control" is needed to avoid overwhelming device buffers.

Akka supports two types of flow control:

- *Ack-based*, where the driver notifies the writer when writes have succeeded.
- *Nack-based*, where the driver notifies the writer when writes have failed.

Each of these models is available in both the TCP and the UDP implementations of Akka I/O.

Individual writes can be acknowledged by providing an ack object in the write message (`Write` in the case of TCP and `Send` for UDP). When the write is complete the worker will send the ack object to the writing actor. This can be used to implement *ack-based* flow control; sending new data only when old data has been acknowledged.

If a write (or any other command) fails, the driver notifies the actor that sent the command with a special message (`CommandFailed` in the case of UDP and TCP). This message will also notify the writer of a failed write, serving as a nack for that write. Please note, that in a nack-based flow-control setting the writer has to be prepared for the fact that the failed write might not be the most recent write it sent. For example, the failure notification for a write `W1` might arrive after additional write commands `W2` and `W3` have been sent. If the writer wants to resend any nacked messages it may need to keep a buffer of pending messages.

Warning: An acknowledged write does not mean acknowledged delivery or storage; receiving an ack for a write simply signals that the I/O driver has successfully processed the write. The Ack/Nack protocol described here is a means of flow control not error handling. In other words, data may still be lost, even if every write is acknowledged.

ByteString

To maintain isolation, actors should communicate with immutable objects only. `ByteString` is an immutable container for bytes. It is used by Akka's I/O system as an efficient, immutable alternative the traditional byte containers used for I/O on the JVM, such as `Array[Byte]` and `ByteBuffer`.

`ByteString` is a rope-like data structure that is immutable and provides fast concatenation and slicing operations (perfect for I/O). When two `ByteString`s 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 `ByteString`s 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. If you require a `ByteString` that only blocks as much memory as necessary for it's content, use the `compact` method to get a `CompactByteString` instance. If the `ByteString` represented only a slice of the original array, this will result in copying all bytes in that slice.

`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.

`ByteString` also comes with its own optimized builder and iterator classes `ByteStringBuilder` and `ByteIterator` which provide extra features in addition to those of normal builders and iterators.

Compatibility with java.io

A `ByteStringBuilder` can be wrapped in a `java.io.OutputStream` via the `asOutputStream` method. Likewise, `ByteIterator` can be wrapped in a `java.io.InputStream` via `asInputStream`. Using these, `akka.io` applications can integrate legacy code based on `java.io` streams.

5.5.3 Architecture in-depth

For further details on the design and internal architecture see *I/O Layer Design*.

5.6 Using TCP

The code snippets through-out this section assume the following imports:

```
import akka.actor.{ Actor, ActorRef, Props }
import akka.io.{ IO, Tcp }
import akka.util.ByteString
import java.net.InetSocketAddress
```

All of the Akka I/O APIs are accessed through manager objects. When using an I/O API, the first step is to acquire a reference to the appropriate manager. The code below shows how to acquire a reference to the `Tcp` manager.

```
import akka.io.{ IO, Tcp }
import context.system // implicitly used by IO(Tcp)

val manager = IO(Tcp)
```

The manager is an actor that handles the underlying low level I/O resources (selectors, channels) and instantiates workers for specific tasks, such as listening to incoming connections.

5.6.1 Connecting

```
object Client {
  def props(remote: InetSocketAddress, replies: ActorRef) =
    Props(classOf[Client], remote, replies)
}

class Client(remote: InetSocketAddress, listener: ActorRef) extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Connect(remote)

  def receive = {
    case CommandFailed(_: Connect) =>
      listener ! "connect failed"
      context stop self

    case c @ Connected(remote, local) =>
      listener ! c
      val connection = sender()
      connection ! Register(self)
      context become {
        case data: ByteString =>
          connection ! Write(data)
        case CommandFailed(w: Write) =>
          // O/S buffer was full
          listener ! "write failed"
        case Received(data) =>
          listener ! data
        case "close" =>
          connection ! Close
        case _: ConnectionClosed =>
          listener ! "connection closed"
          context stop self
      }
  }
}
```

The first step of connecting to a remote address is sending a `Connect` message to the TCP manager; in addition to the simplest form shown above there is also the possibility to specify a local `InetSocketAddress` to bind to and a list of socket options to apply.

Note: The SO_NODELAY (TCP_NODELAY on Windows) socket option defaults to true in Akka, indepen-

dently of the OS default settings. This setting disables Nagle's algorithm, considerably improving latency for most applications. This setting could be overridden by passing `SO.TcpNoDelay(false)` in the list of socket options of the `Connect` message.

The TCP manager will then reply either with a `CommandFailed` or it will spawn an internal actor representing the new connection. This new actor will then send a `Connected` message to the original sender of the `Connect` message.

In order to activate the new connection a `Register` message must be sent to the connection actor, informing that one about who shall receive data from the socket. Before this step is done the connection cannot be used, and there is an internal timeout after which the connection actor will shut itself down if no `Register` message is received.

The connection actor watches the registered handler and closes the connection when that one terminates, thereby cleaning up all internal resources associated with that connection.

The actor in the example above uses `become` to switch from unconnected to connected operation, demonstrating the commands and events which are observed in that state. For a discussion on `CommandFailed` see Throttling Reads and Writes below. `ConnectionClosed` is a trait, which marks the different connection close events. The last line handles all connection close events in the same way. It is possible to listen for more fine-grained connection close events, see Closing Connections below.

5.6.2 Accepting connections

```
class Server extends Actor {

  import Tcp._
  import context.system

  IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 0))

  def receive = {
    case b @ Bound(localAddress) =>
      // do some logging or setup ...

    case CommandFailed(_: Bind) => context stop self

    case c @ Connected(remote, local) =>
      val handler = context.actorOf(Props[SimplisticHandler])
      val connection = sender()
      connection ! Register(handler)
  }

}
```

To create a TCP server and listen for inbound connections, a `Bind` command has to be sent to the TCP manager. This will instruct the TCP manager to listen for TCP connections on a particular `InetSocketAddress`; the port may be specified as `0` in order to bind to a random port.

The actor sending the `Bind` message will receive a `Bound` message signalling that the server is ready to accept incoming connections; this message also contains the `InetSocketAddress` to which the socket was actually bound (i.e. resolved IP address and correct port number).

From this point forward the process of handling connections is the same as for outgoing connections. The example demonstrates that handling the reads from a certain connection can be delegated to another actor by naming it as the handler when sending the `Register` message. Writes can be sent from any actor in the system to the connection actor (i.e. the actor which sent the `Connected` message). The simplistic handler is defined as:

```
class SimplisticHandler extends Actor {
  import Tcp._
  def receive = {
    case Received(data) => sender() ! Write(data)
```

```
    case PeerClosed     => context stop self
  }
}
```

For a more complete sample which also takes into account the possibility of failures when sending please see Throttling Reads and Writes below.

The only difference to outgoing connections is that the internal actor managing the listen port—the sender of the `Bound` message—watches the actor which was named as the recipient for `Connected` messages in the `Bind` message. When that actor terminates the listen port will be closed and all resources associated with it will be released; existing connections will not be terminated at this point.

5.6.3 Closing connections

A connection can be closed by sending one of the commands `Close`, `ConfirmedClose` or `Abort` to the connection actor.

`Close` will close the connection by sending a `FIN` message, but without waiting for confirmation from the remote endpoint. Pending writes will be flushed. If the close is successful, the listener will be notified with `Closed`.

`ConfirmedClose` will close the sending direction of the connection by sending a `FIN` message, but data will continue to be received until the remote endpoint closes the connection, too. Pending writes will be flushed. If the close is successful, the listener will be notified with `ConfirmedClosed`.

`Abort` will immediately terminate the connection by sending a `RST` message to the remote endpoint. Pending writes will be not flushed. If the close is successful, the listener will be notified with `Aborted`.

`PeerClosed` will be sent to the listener if the connection has been closed by the remote endpoint. Per default, the connection will then automatically be closed from this endpoint as well. To support half-closed connections set the `keepOpenOnPeerClosed` member of the `Register` message to `true` in which case the connection stays open until it receives one of the above close commands.

`ErrorClosed` will be sent to the listener whenever an error happened that forced the connection to be closed.

All close notifications are sub-types of `ConnectionClosed` so listeners who do not need fine-grained close events may handle all close events in the same way.

5.6.4 Writing to a connection

Once a connection has been established data can be sent to it from any actor in the form of a `Tcp.WriteCommand`. `Tcp.WriteCommand` is an abstract class with three concrete implementations:

Tcp.Write The simplest `WriteCommand` implementation which wraps a `ByteString` instance and an "ack" event. A `ByteString` (as explained in *this section*) models one or more chunks of immutable in-memory data with a maximum (total) size of 2 GB (2^31 bytes).

Tcp.WriteFile If you want to send "raw" data from a file you can do so efficiently with the `Tcp.WriteFile` command. This allows you do designate a (contiguous) chunk of on-disk bytes for sending across the connection without the need to first load them into the JVM memory. As such `Tcp.WriteFile` can "hold" more than 2GB of data and an "ack" event if required.

Tcp.CompoundWrite Sometimes you might want to group (or interleave) several `Tcp.Write` and/or `Tcp.WriteFile` commands into one atomic write command which gets written to the connection in one go. The `Tcp.CompoundWrite` allows you to do just that and offers three benefits:

1. As explained in the following section the TCP connection actor can only handle one single write command at a time. By combining several writes into one `CompoundWrite` you can have them be sent across the connection with minimum overhead and without the need to spoon feed them to the connection actor via an *ACK-based* message protocol.
2. Because a `WriteCommand` is atomic you can be sure that no other actor can "inject" other writes into your series of writes if you combine them into one single `CompoundWrite`. In scenarios where

several actors write to the same connection this can be an important feature which can be somewhat hard to achieve otherwise.

3. The "sub writes" of a `CompoundWrite` are regular `Write` or `WriteFile` commands that themselves can request "ack" events. These ACKs are sent out as soon as the respective "sub write" has been completed. This allows you to attach more than one ACK to a `Write` or `WriteFile` (by combining it with an empty write that itself requests an ACK) or to have the connection actor acknowledge the progress of transmitting the `CompoundWrite` by sending out intermediate ACKs at arbitrary points.

5.6.5 Throttling Reads and Writes

The basic model of the TCP connection actor is that it has no internal buffering (i.e. it can only process one write at a time, meaning it can buffer one write until it has been passed on to the O/S kernel in full). Congestion needs to be handled at the user level, for both writes and reads.

For back-pressuring writes there are three modes of operation

- *ACK-based:* every `Write` command carries an arbitrary object, and if this object is not `Tcp.NoAck` then it will be returned to the sender of the `Write` upon successfully writing all contained data to the socket. If no other write is initiated before having received this acknowledgement then no failures can happen due to buffer overrun.
- *NACK-based:* every write which arrives while a previous write is not yet completed will be replied to with a `CommandFailed` message containing the failed write. Just relying on this mechanism requires the implemented protocol to tolerate skipping writes (e.g. if each write is a valid message on its own and it is not required that all are delivered). This mode is enabled by setting the `useResumeWriting` flag to `false` within the `Register` message during connection activation.
- *NACK-based with write suspending:* this mode is very similar to the NACK-based one, but once a single write has failed no further writes will succeed until a `ResumeWriting` message is received. This message will be answered with a `WritingResumed` message once the last accepted write has completed. If the actor driving the connection implements buffering and resends the NACK'ed messages after having awaited the `WritingResumed` signal then every message is delivered exactly once to the network socket.

These write back-pressure models (with the exception of the second which is rather specialised) are demonstrated in complete examples below. The full and contiguous source is available on github.

For back-pressuring reads there are two modes of operation

- *Push-reading:* in this mode the connection actor sends the registered reader actor incoming data as soon as available as `Received` events. Whenever the reader actor wants to signal back-pressure to the remote TCP endpoint it can send a `SuspendReading` message to the connection actor to indicate that it wants to suspend the reception of new data. No `Received` events will arrive until a corresponding `ResumeReading` is sent indicating that the receiver actor is ready again.
- *Pull-reading:* after sending a `Received` event the connection actor automatically suspends accepting data from the socket until the reader actor signals with a `ResumeReading` message that it is ready to process more input data. Hence new data is "pulled" from the connection by sending `ResumeReading` messages.

Note: It should be obvious that all these flow control schemes only work between one writer/reader and one connection actor; as soon as multiple actors send write commands to a single connection no consistent result can be achieved.

5.6.6 ACK-Based Write Back-Pressure

For proper function of the following example it is important to configure the connection to remain half-open when the remote side closed its writing end: this allows the example `EchoHandler` to write all outstanding data back to the client before fully closing the connection. This is enabled using a flag upon connection activation (observe the `Register` message):

```
case Connected(remote, local) =>
  log.info("received connection from {}", remote)
  val handler = context.actorOf(Props(handlerClass, sender(), remote))
  sender() ! Register(handler, keepOpenOnPeerClosed = true)
```

With this preparation let us dive into the handler itself:

```
  // storage omitted ...
class SimpleEchoHandler(connection: ActorRef, remote: InetSocketAddress)
  extends Actor with ActorLogging {

  import Tcp._

  // sign death pact: this actor terminates when connection breaks
  context watch connection

  case object Ack extends Event

  def receive = {
    case Received(data) =>
      buffer(data)
      connection ! Write(data, Ack)

      context.become({
        case Received(data) => buffer(data)
        case Ack            => acknowledge()
        case PeerClosed     => closing = true
      }, discardOld = false)

    case PeerClosed => context stop self
  }

  // storage omitted ...
}
```

The principle is simple: when having written a chunk always wait for the `Ack` to come back before sending the next chunk. While waiting we switch behavior such that new incoming data are buffered. The helper functions used are a bit lengthy but not complicated:

```
private def buffer(data: ByteString): Unit = {
  storage :+= data
  stored += data.size

  if (stored > maxStored) {
    log.warning(s"drop connection to [$remote] (buffer overrun)")
    context stop self

  } else if (stored > highWatermark) {
    log.debug(s"suspending reading")
    connection ! SuspendReading
    suspended = true
  }
}

private def acknowledge(): Unit = {
  require(storage.nonEmpty, "storage was empty")

  val size = storage(0).size
  stored -= size
  transferred += size

  storage = storage drop 1
```

```
  if (suspended && stored < lowWatermark) {
    log.debug("resuming reading")
    connection ! ResumeReading
    suspended = false
  }

  if (storage.isEmpty) {
    if (closing) context stop self
    else context.unbecome()
  } else connection ! Write(storage(0), Ack)
}
```

The most interesting part is probably the last: an `Ack` removes the oldest data chunk from the buffer, and if that was the last chunk then we either close the connection (if the peer closed its half already) or return to the idle behavior; otherwise we just send the next buffered chunk and stay waiting for the next `Ack`.

Back-pressure can be propagated also across the reading side back to the writer on the other end of the connection by sending the `SuspendReading` command to the connection actor. This will lead to no data being read from the socket anymore (although this does happen after a delay because it takes some time until the connection actor processes this command, hence appropriate head-room in the buffer should be present), which in turn will lead to the O/S kernel buffer filling up on our end, then the TCP window mechanism will stop the remote side from writing, filling up its write buffer, until finally the writer on the other side cannot push any data into the socket anymore. This is how end-to-end back-pressure is realized across a TCP connection.

5.6.7 NACK-Based Write Back-Pressure with Suspending

```
class EchoHandler(connection: ActorRef, remote: InetSocketAddress)
  extends Actor with ActorLogging {

  import Tcp._

  case class Ack(offset: Int) extends Event

  // sign death pact: this actor terminates when connection breaks
  context watch connection

  // start out in optimistic write-through mode
  def receive = writing

  def writing: Receive = {
    case Received(data) =>
      connection ! Write(data, Ack(currentOffset))
      buffer(data)

    case Ack(ack) =>
      acknowledge(ack)

    case CommandFailed(Write(_, Ack(ack))) =>
      connection ! ResumeWriting
      context become buffering(ack)

    case PeerClosed =>
      if (storage.isEmpty) context stop self
      else context become closing
  }

  // buffering ...

  // closing ...

  override def postStop(): Unit = {
```

```
    log.info(s"transferred $transferred bytes from/to [$remote]")
  }

  // storage omitted ...
}
  // storage omitted ...
```

The principle here is to keep writing until a `CommandFailed` is received, using acknowledgements only to prune the resend buffer. When a such a failure was received, transition into a different state for handling and handle resending of all queued data:

```
def buffering(nack: Int): Receive = {
  var toAck = 10
  var peerClosed = false

  {
    case Received(data)         => buffer(data)
    case WritingResumed         => writeFirst()
    case PeerClosed             => peerClosed = true
    case Ack(ack) if ack < nack => acknowledge(ack)
    case Ack(ack) =>
      acknowledge(ack)
      if (storage.nonEmpty) {
        if (toAck > 0) {
          // stay in ACK-based mode for a while
          writeFirst()
          toAck -= 1
        } else {
          // then return to NACK-based again
          writeAll()
          context become (if (peerClosed) closing else writing)
        }
      } else if (peerClosed) context stop self
      else context become writing
  }
}
```

It should be noted that all writes which are currently buffered have also been sent to the connection actor upon entering this state, which means that the `ResumeWriting` message is enqueued after those writes, leading to the reception of all outstanding `CommandFailed` messages (which are ignored in this state) before receiving the `WritingResumed` signal. That latter message is sent by the connection actor only once the internally queued write has been fully completed, meaning that a subsequent write will not fail. This is exploited by the `EchoHandler` to switch to an ACK-based approach for the first ten writes after a failure before resuming the optimistic write-through behavior.

```
def closing: Receive = {
  case CommandFailed(_: Write) =>
    connection ! ResumeWriting
    context.become({

      case WritingResumed =>
        writeAll()
        context.unbecome()

      case ack: Int => acknowledge(ack)

    }, discardOld = false)

  case Ack(ack) =>
    acknowledge(ack)
    if (storage.isEmpty) context stop self
}
```

Closing the connection while still sending all data is a bit more involved than in the ACK-based approach: the idea is to always send all outstanding messages and acknowledge all successful writes, and if a failure happens then switch behavior to await the `WritingResumed` event and start over.

The helper functions are very similar to the ACK-based case:

```
private def buffer(data: ByteString): Unit = {
  storage :+= data
  stored += data.size

  if (stored > maxStored) {
    log.warning(s"drop connection to [$remote] (buffer overrun)")
    context stop self

  } else if (stored > highWatermark) {
    log.debug(s"suspending reading at $currentOffset")
    connection ! SuspendReading
    suspended = true
  }
}

private def acknowledge(ack: Int): Unit = {
  require(ack == storageOffset, s"received ack $ack at $storageOffset")
  require(storage.nonEmpty, s"storage was empty at ack $ack")

  val size = storage(0).size
  stored -= size
  transferred += size

  storageOffset += 1
  storage = storage drop 1

  if (suspended && stored < lowWatermark) {
    log.debug("resuming reading")
    connection ! ResumeReading
    suspended = false
  }
}
```

5.6.8 Read Back-Pressure with Pull Mode

When using push based reading, data coming from the socket is sent to the actor as soon as it is available. In the case of the previous Echo server example this meant that we needed to maintain a buffer of incoming data to keep it around since the rate of writing might be slower than the rate of the arrival of new data.

With the Pull mode this buffer can be completely eliminated as the following snippet demonstrates:

```
override def preStart: Unit = connection ! ResumeReading

def receive = {
  case Received(data) => connection ! Write(data, Ack)
  case Ack            => connection ! ResumeReading
}
```

The idea here is that reading is not resumed until the previous write has been completely acknowledged by the connection actor. Every pull mode connection actor starts from suspended state. To start the flow of data we send a `ResumeReading` in the `preStart` method to tell the connection actor that we are ready to receive the first chunk of data. Since we only resume reading when the previous data chunk has been completely written there is no need for maintaining a buffer.

To enable pull reading on an outbound connection the `pullMode` parameter of the `Connect` should be set to `true`:

```
IO(Tcp) ! Connect(listenAddress, pullMode = true)
```

Pull Mode Reading for Inbound Connections

The previous section demonstrated how to enable pull reading mode for outbound connections but it is possible to create a listener actor with this mode of reading by setting the `pullMode` parameter of the `Bind` command to `true`:

```
IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 0), pullMode = true)
```

One of the effects of this setting is that all connections accepted by this listener actor will use pull mode reading.

Another effect of this setting is that in addition of setting all inbound connections to pull mode, accepting connections becomes pull based, too. This means that after handling one (or more) `Connected` events the listener actor has to be resumed by sending it a `ResumeAccepting` message.

Listener actors with pull mode start suspended so to start accepting connections a `ResumeAccepting` command has to be sent to the listener actor after binding was successful:

```
case Bound(localAddress) =>
  // Accept connections one by one
  sender ! ResumeAccepting(batchSize = 1)
  context.become(listening(sender))
```

After handling an incoming connection we need to resume accepting again:

```
def listening(listener: ActorRef): Receive = {
  case Connected(remote, local) =>
    val handler = context.actorOf(Props(classOf[PullEcho], sender))
    sender ! Register(handler, keepOpenOnPeerClosed = true)
    listener ! ResumeAccepting(batchSize = 1)
}
```

The `ResumeAccepting` accepts a `batchSize` parameter that specifies how many new connections are accepted before a next `ResumeAccepting` message is needed to resume handling of new connections.

5.7 Using UDP

UDP is a connectionless datagram protocol which offers two different ways of communication on the JDK level:

- sockets which are free to send datagrams to any destination and receive datagrams from any origin
- sockets which are restricted to communication with one specific remote socket address

In the low-level API the distinction is made—confusingly—by whether or not `connect` has been called on the socket (even when connect has been called the protocol is still connectionless). These two forms of UDP usage are offered using distinct IO extensions described below.

5.7.1 Unconnected UDP

Simple Send

```
class SimpleSender(remote: InetSocketAddress) extends Actor {
  import context.system
  IO(Udp) ! Udp.SimpleSender

  def receive = {
    case Udp.SimpleSenderReady =>
      context.become(ready(sender()))
```

```
  }

  def ready(send: ActorRef): Receive = {
    case msg: String =>
      send ! Udp.Send(ByteString(msg), remote)
  }
}
```

The simplest form of UDP usage is to just send datagrams without the need of getting a reply. To this end a "simple sender" facility is provided as demonstrated above. The UDP extension is queried using the `SimpleSender` message, which is answered by a `SimpleSenderReady` notification. The sender of this message is the newly created sender actor which from this point onward can be used to send datagrams to arbitrary destinations; in this example it will just send any UTF-8 encoded `String` it receives to a predefined remote address.

Note: The simple sender will not shut itself down because it cannot know when you are done with it. You will need to send it a `PoisonPill` when you want to close the ephemeral port the sender is bound to.

Bind (and Send)

```
class Listener(nextActor: ActorRef) extends Actor {
  import context.system
  IO(Udp) ! Udp.Bind(self, new InetSocketAddress("localhost", 0))

  def receive = {
    case Udp.Bound(local) =>
      context.become(ready(sender()))
  }

  def ready(socket: ActorRef): Receive = {
    case Udp.Received(data, remote) =>
      val processed = // parse data etc., e.g. using PipelineStage
      socket ! Udp.Send(data, remote) // example server echoes back
      nextActor ! processed
    case Udp.Unbind  => socket ! Udp.Unbind
    case Udp.Unbound => context.stop(self)
  }
}
```

If you want to implement a UDP server which listens on a socket for incoming datagrams then you need to use the `Bind` command as shown above. The local address specified may have a zero port in which case the operating system will automatically choose a free port and assign it to the new socket. Which port was actually bound can be found out by inspecting the `Bound` message.

The sender of the `Bound` message is the actor which manages the new socket. Sending datagrams is achieved by using the `Send` message type and the socket can be closed by sending a `Unbind` command, in which case the socket actor will reply with a `Unbound` notification.

Received datagrams are sent to the actor designated in the `Bind` message, whereas the `Bound` message will be sent to the sender of the `Bind`.

5.7.2 Connected UDP

The service provided by the connection based UDP API is similar to the bind-and-send service we saw earlier, but the main difference is that a connection is only able to send to the `remoteAddress` it was connected to, and will receive datagrams only from that address.

```
class Connected(remote: InetSocketAddress) extends Actor {
  import context.system
  IO(UdpConnected) ! UdpConnected.Connect(self, remote)
```

```
  def receive = {
    case UdpConnected.Connected =>
      context.become(ready(sender()))
  }

  def ready(connection: ActorRef): Receive = {
    case UdpConnected.Received(data) =>
      // process data, send it on, etc.
    case msg: String =>
      connection ! UdpConnected.Send(ByteString(msg))
    case d @ UdpConnected.Disconnect => connection ! d
    case UdpConnected.Disconnected   => context.stop(self)
  }
}
```

Consequently the example shown here looks quite similar to the previous one, the biggest difference is the absence of remote address information in `Send` and `Received` messages.

Note: There is a small performance benefit in using connection based UDP API over the connectionless one. If there is a SecurityManager enabled on the system, every connectionless message send has to go through a security check, while in the case of connection-based UDP the security check is cached after connect, thus writes do not suffer an additional performance penalty.

5.8 ZeroMQ

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.

Note: The currently used version of `zeromq-scala-bindings` is only compatible with zeromq 2; zeromq 3 is not supported.

5.8.1 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://127.0.0.1:21231"))
```

Above examples will create a ZeroMQ Publisher socket that is Bound to the port 21231 on localhost.

Similarly you can create a subscription socket, with a listener, that subscribes to all messages from the publisher using:

```
import akka.zeromq._

class Listener extends Actor {
  def receive: Receive = {
    case Connecting    => //...
    case m: ZMQMessage => //...
    case _             => //...
  }
}

val listener = system.actorOf(Props(classOf[Listener], this))
val subSocket = ZeroMQExtension(system).newSocket(SocketType.Sub,
  Listener(listener), Connect("tcp://127.0.0.1:21231"), SubscribeAll)
```

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 = ZeroMQExtension(system).newSocket(SocketType.Sub,
  Listener(listener), Connect("tcp://127.0.0.1:21231"), 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 `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(ByteString("foo.bar"), ByteString(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 = ZeroMQExtension(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)
  import context.dispatcher

  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)).get
      // the first frame is the topic, second is the message
      pubSocket ! ZMQMessage(ByteString("health.heap"), ByteString(heapPayload))

      // use akka SerializationExtension to convert to bytes
      val loadPayload = ser.serialize(Load(timestamp, os.getSystemLoadAverage)).get
      // the first frame is the topic, second is the message
      pubSocket ! ZMQMessage(ByteString("health.load"), ByteString(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 {

  ZeroMQExtension(context.system).newSocket(SocketType.Sub, Listener(self),
    Connect("tcp://127.0.0.1:1235"), Subscribe("health"))
  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.frames(0).utf8String == "health.heap" =>
      val Heap(timestamp, used, max) = ser.deserialize(m.frames(1).toArray,
        classOf[Heap]).get
      log.info("Used heap {} bytes, at {}", used,
        timestampFormat.format(new Date(timestamp)))

    case m: ZMQMessage if m.frames(0).utf8String == "health.load" =>
      val Load(timestamp, loadAverage) = ser.deserialize(m.frames(1).toArray,
        classOf[Load]).get
```

```
      log.info("Load average {}, at {}", loadAverage,
        timestampFormat.format(new Date(timestamp)))
  }
}

  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 {

  ZeroMQExtension(context.system).newSocket(SocketType.Sub,
    Listener(self), Connect("tcp://127.0.0.1:1235"), Subscribe("health.heap"))
  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.frames(0).utf8String == "health.heap" =>
      val Heap(timestamp, used, max) =
        ser.deserialize(m.frames(1).toArray, classOf[Heap]).get
      if ((used.toDouble / max) > 0.9) count += 1
      else count = 0
      if (count > 10) log.warning("Need more memory, using {} %",
        (100.0 * used / max))
  }
}

  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 = ZeroMQExtension(system).newSocket(
  SocketType.Router,
  Listener(listener),
  Bind("tcp://127.0.0.1:21233"),
  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.

5.8.2 Configuration

There are several configuration properties for the zeromq module, please refer to the *reference configuration*.

5.9 Camel

5.9.1 Introduction

The akka-camel module allows Untyped Actors to receive and send messages over a great variety of protocols and APIs. In addition to the native Scala and Java actor API, actors can now exchange messages with other systems over large number of protocols and APIs such as HTTP, SOAP, TCP, FTP, SMTP or JMS, to mention a few. At the moment, approximately 80 protocols and APIs are supported.

Apache Camel

The akka-camel module is based on Apache Camel, a powerful and light-weight integration framework for the JVM. For an introduction to Apache Camel you may want to read this Apache Camel article. Camel comes with a large number of components that provide bindings to different protocols and APIs. The camel-extra project provides further components.

Consumer

Usage of Camel's integration components in Akka is essentially a one-liner. Here's an example.

```
import akka.camel.{ CamelMessage, Consumer }

class MyEndpoint extends Consumer {
  def endpointUri = "mina2:tcp://localhost:6200?textline=true"

  def receive = {
    case msg: CamelMessage => { /* ... */ }
    case _                 => { /* ... */ }
  }
}

// start and expose actor via tcp
import akka.actor.{ ActorSystem, Props }

val system = ActorSystem("some-system")
val mina = system.actorOf(Props[MyEndpoint])
```

The above example exposes an actor over a TCP endpoint via Apache Camel's Mina component. The actor implements the endpointUri method to define an endpoint from which it can receive messages. After starting the actor, TCP clients can immediately send messages to and receive responses from that actor. If the message exchange should go over HTTP (via Camel's Jetty component), only the actor's endpointUri method must be changed.

```
import akka.camel.{ CamelMessage, Consumer }

class MyEndpoint extends Consumer {
  def endpointUri = "jetty:http://localhost:8877/example"

  def receive = {
    case msg: CamelMessage => { /* ... */ }
    case _                 => { /* ... */ }
  }
}
```

Producer

Actors can also trigger message exchanges with external systems i.e. produce to Camel endpoints.

```
import akka.actor.Actor
import akka.camel.{ Producer, Oneway }
import akka.actor.{ ActorSystem, Props }

class Orders extends Actor with Producer with Oneway {
  def endpointUri = "jms:queue:Orders"
}

val sys = ActorSystem("some-system")
val orders = sys.actorOf(Props[Orders])

orders ! <order amount="100" currency="PLN" itemId="12345"/>
```

In the above example, any message sent to this actor will be sent to the JMS queue `orders`. Producer actors may choose from the same set of Camel components as Consumer actors do.

CamelMessage

The number of Camel components is constantly increasing. The akka-camel module can support these in a plug-and-play manner. Just add them to your application's classpath, define a component-specific endpoint URI and use it to exchange messages over the component-specific protocols or APIs. This is possible because Camel components bind protocol-specific message formats to a Camel-specific normalized message format. The normalized message format hides protocol-specific details from Akka and makes it therefore very easy to support a large number of protocols through a uniform Camel component interface. The akka-camel module further converts mutable Camel messages into immutable representations which are used by Consumer and Producer actors for pattern matching, transformation, serialization or storage. In the above example of the Orders Producer, the XML message is put in the body of a newly created Camel Message with an empty set of headers. You can also create a CamelMessage yourself with the appropriate body and headers as you see fit.

CamelExtension

The akka-camel module is implemented as an Akka Extension, the `CamelExtension` object. Extensions will only be loaded once per `ActorSystem`, which will be managed by Akka. The `CamelExtension` object provides access to the Camel trait. The Camel trait in turn provides access to two important Apache Camel objects, the CamelContext and the ProducerTemplate. Below you can see how you can get access to these Apache Camel objects.

```
val system = ActorSystem("some-system")
val camel = CamelExtension(system)
val camelContext = camel.context
val producerTemplate = camel.template
```

One `CamelExtension` is only loaded once for every one `ActorSystem`, which makes it safe to call the `CamelExtension` at any point in your code to get to the Apache Camel objects associated with it. There is one CamelContext and one ProducerTemplate for every one `ActorSystem` that uses a `CamelExtension`. By Default, a new CamelContext is created when the `CamelExtension` starts. If you want to inject your own context instead, you can extend the ContextProvider trait and add the FQCN of your implementation in the config, as the value of the "akka.camel.context-provider". This interface define a single method `getContext` used to load the CamelContext.

Below an example on how to add the ActiveMQ component to the CamelContext, which is required when you would like to use the ActiveMQ component.

```
// import org.apache.activemq.camel.component.ActiveMQComponent
val system = ActorSystem("some-system")
val camel = CamelExtension(system)
val camelContext = camel.context
// camelContext.addComponent("activemq", ActiveMQComponent.activeMQComponent(
//   "vm://localhost?broker.persistent=false"))
```

The CamelContext joins the lifecycle of the `ActorSystem` and `CamelExtension` it is associated with; the CamelContext is started when the `CamelExtension` is created, and it is shut down when the associated `ActorSystem` is shut down. The same is true for the ProducerTemplate.

The `CamelExtension` is used by both *Producer* and *Consumer* actors to interact with Apache Camel internally. You can access the `CamelExtension` inside a *Producer* or a *Consumer* using the `camel` definition, or get straight at the *CamelContext* using the `camelContext` definition. Actors are created and started asynchronously. When a *Consumer* actor is created, the *Consumer* is published at its Camel endpoint (more precisely, the route is added to the CamelContext from the Endpoint to the actor). When a *Producer* actor is created, a SendProcessor and Endpoint are created so that the Producer can send messages to it. Publication is done asynchronously; setting up an endpoint may still be in progress after you have requested the actor to be created. Some Camel components can take a while to startup, and in some cases you might want to know when the endpoints are activated and ready to be used. The Camel trait allows you to find out when the endpoint is activated or deactivated.

```
import akka.camel.{ CamelMessage, Consumer }
import scala.concurrent.duration._

class MyEndpoint extends Consumer {
  def endpointUri = "mina2:tcp://localhost:6200?textline=true"

  def receive = {
    case msg: CamelMessage => { /* ... */ }
    case _                 => { /* ... */ }
  }
}
val system = ActorSystem("some-system")
val camel = CamelExtension(system)
val actorRef = system.actorOf(Props[MyEndpoint])
// get a future reference to the activation of the endpoint of the Consumer Actor
val activationFuture = camel.activationFutureFor(actorRef)(timeout = 10 seconds,
  executor = system.dispatcher)
```

The above code shows that you can get a `Future` to the activation of the route from the endpoint to the actor, or you can wait in a blocking fashion on the activation of the route. An `ActivationTimeoutException` is thrown if the endpoint could not be activated within the specified timeout. Deactivation works in a similar fashion:

```
system.stop(actorRef)
// get a future reference to the deactivation of the endpoint of the Consumer Actor
val deactivationFuture = camel.deactivationFutureFor(actorRef)(timeout = 10 seconds,
  executor = system.dispatcher)
```

Deactivation of a Consumer or a Producer actor happens when the actor is terminated. For a Consumer, the route to the actor is stopped. For a Producer, the SendProcessor is stopped. A `DeActivationTimeoutException` is thrown if the associated camel objects could not be deactivated within the specified timeout.

5.9.2 Consumer Actors

For objects to receive messages, they must mixin the Consumer trait. For example, the following actor class (Consumer1) implements the endpointUri method, which is declared in the Consumer trait, in order to receive messages from the `file:data/input/actor` Camel endpoint.

```
import akka.camel.{ CamelMessage, Consumer }

class Consumer1 extends Consumer {
  def endpointUri = "file:data/input/actor"

  def receive = {
    case msg: CamelMessage => println("received %s" format msg.bodyAs[String])
  }
}
```

Whenever a file is put into the data/input/actor directory, its content is picked up by the Camel file component and sent as message to the actor. Messages consumed by actors from Camel endpoints are of type CamelMessage. These are immutable representations of Camel messages.

Here's another example that sets the endpointUri to `jetty:http://localhost:8877/camel/default`. It causes Camel's Jetty component to start an embedded Jetty server, accepting HTTP connections from localhost on port 8877.

```
import akka.camel.{ CamelMessage, Consumer }

class Consumer2 extends Consumer {
  def endpointUri = "jetty:http://localhost:8877/camel/default"

  def receive = {
    case msg: CamelMessage => sender() ! ("Hello %s" format msg.bodyAs[String])
  }
}
```

After starting the actor, clients can send messages to that actor by POSTing to `http://localhost:8877/camel/default`. The actor sends a response by using the sender *!* method. For returning a message body and headers to the HTTP client the response type should be CamelMessage. For any other response type, a new CamelMessage object is created by akka-camel with the actor response as message body.

Delivery acknowledgements

With in-out message exchanges, clients usually know that a message exchange is done when they receive a reply from a consumer actor. The reply message can be a CamelMessage (or any object which is then internally converted to a CamelMessage) on success, and a Failure message on failure.

With in-only message exchanges, by default, an exchange is done when a message is added to the consumer actor's mailbox. Any failure or exception that occurs during processing of that message by the consumer actor cannot be reported back to the endpoint in this case. To allow consumer actors to positively or negatively acknowledge the receipt of a message from an in-only message exchange, they need to override the `autoAck` method to return false. In this case, consumer actors must reply either with a special akka.camel.Ack message (positive acknowledgement) or a akka.actor.Status.Failure (negative acknowledgement).

```
import akka.camel.{ CamelMessage, Consumer }
import akka.camel.Ack
import akka.actor.Status.Failure
```

```
class Consumer3 extends Consumer {
  override def autoAck = false

  def endpointUri = "jms:queue:test"

  def receive = {
    case msg: CamelMessage =>
      sender() ! Ack
      // on success
      // ..
      val someException = new Exception("e1")
      // on failure
      sender() ! Failure(someException)
  }
}
```

Consumer timeout

Camel Exchanges (and their corresponding endpoints) that support two-way communications need to wait for a response from an actor before returning it to the initiating client. For some endpoint types, timeout values can be defined in an endpoint-specific way which is described in the documentation of the individual Camel components. Another option is to configure timeouts on the level of consumer actors.

Two-way communications between a Camel endpoint and an actor are initiated by sending the request message to the actor with the ask pattern and the actor replies to the endpoint when the response is ready. The ask request to the actor can timeout, which will result in the Exchange failing with a TimeoutException set on the failure of the Exchange. The timeout on the consumer actor can be overridden with the `replyTimeout`, as shown below.

```
import akka.camel.{ CamelMessage, Consumer }
import scala.concurrent.duration._

class Consumer4 extends Consumer {
  def endpointUri = "jetty:http://localhost:8877/camel/default"
  override def replyTimeout = 500 millis
  def receive = {
    case msg: CamelMessage => sender() ! ("Hello %s" format msg.bodyAs[String])
  }
}
```

5.9.3 Producer Actors

For sending messages to Camel endpoints, actors need to mixin the Producer trait and implement the endpointUri method.

```
import akka.actor.Actor
import akka.actor.{ Props, ActorSystem }
import akka.camel.{ Producer, CamelMessage }
import akka.util.Timeout

class Producer1 extends Actor with Producer {
  def endpointUri = "http://localhost:8080/news"
}
```

Producer1 inherits a default implementation of the receive method from the Producer trait. To customize a producer actor's default behavior you must override the Producer.transformResponse and Producer.transformOutgoingMessage methods. This is explained later in more detail. Producer Actors cannot override the default Producer.receive method.

Any message sent to a Producer actor will be sent to the associated Camel endpoint, in the above example to `http://localhost:8080/news`. The Producer always sends messages asynchronously. Response mes-

sages (if supported by the configured endpoint) will, by default, be returned to the original sender. The following example uses the ask pattern to send a message to a Producer actor and waits for a response.

```
import akka.pattern.ask
import scala.concurrent.duration._
implicit val timeout = Timeout(10 seconds)

val system = ActorSystem("some-system")
val producer = system.actorOf(Props[Producer1])
val future = producer.ask("some request").mapTo[CamelMessage]
```

The future contains the response CamelMessage, or an `AkkaCamelException` when an error occurred, which contains the headers of the response.

Custom Processing

Instead of replying to the initial sender, producer actors can implement custom response processing by overriding the routeResponse method. In the following example, the response message is forwarded to a target actor instead of being replied to the original sender.

```
import akka.actor.{ Actor, ActorRef }
import akka.camel.{ Producer, CamelMessage }
import akka.actor.{ Props, ActorSystem }

class ResponseReceiver extends Actor {
  def receive = {
    case msg: CamelMessage =>
    // do something with the forwarded response
  }
}

class Forwarder(uri: String, target: ActorRef) extends Actor with Producer {
  def endpointUri = uri

  override def routeResponse(msg: Any) { target forward msg }
}
val system = ActorSystem("some-system")
val receiver = system.actorOf(Props[ResponseReceiver])
val forwardResponse = system.actorOf(
  Props(classOf[Forwarder], this, "http://localhost:8080/news/akka", receiver))
// the Forwarder sends out a request to the web page and forwards the response to
// the ResponseReceiver
forwardResponse ! "some request"
```

Before producing messages to endpoints, producer actors can pre-process them by overriding the Producer.transformOutgoingMessage method.

```
import akka.actor.Actor
import akka.camel.{ Producer, CamelMessage }

class Transformer(uri: String) extends Actor with Producer {
  def endpointUri = uri

  def upperCase(msg: CamelMessage) = msg.mapBody {
    body: String => body.toUpperCase
  }

  override def transformOutgoingMessage(msg: Any) = msg match {
    case msg: CamelMessage => upperCase(msg)
  }
}
```

Producer configuration options

The interaction of producer actors with Camel endpoints can be configured to be one-way or two-way (by initiating in-only or in-out message exchanges, respectively). By default, the producer initiates an in-out message exchange with the endpoint. For initiating an in-only exchange, producer actors have to override the oneway method to return true.

```
import akka.actor.{ Actor, Props, ActorSystem }
import akka.camel.Producer

class OnewaySender(uri: String) extends Actor with Producer {
  def endpointUri = uri
  override def oneway: Boolean = true
}

val system = ActorSystem("some-system")
val producer = system.actorOf(Props(classOf[OnewaySender], this, "activemq:FOO.BAR"))
producer ! "Some message"
```

Message correlation

To correlate request with response messages, applications can set the *Message.MessageExchangeId* message header.

```
import akka.camel.{ Producer, CamelMessage }
import akka.actor.Actor
import akka.actor.{ Props, ActorSystem }

class Producer2 extends Actor with Producer {
  def endpointUri = "activemq:FOO.BAR"
}
val system = ActorSystem("some-system")
val producer = system.actorOf(Props[Producer2])

producer ! CamelMessage("bar", Map(CamelMessage.MessageExchangeId -> "123"))
```

ProducerTemplate

The Producer trait is a very convenient way for actors to produce messages to Camel endpoints. Actors may also use a Camel ProducerTemplate for producing messages to endpoints.

```
import akka.actor.Actor
class MyActor extends Actor {
  def receive = {
    case msg =>
      val template = CamelExtension(context.system).template
      template.sendBody("direct:news", msg)
  }
}
```

For initiating a a two-way message exchange, one of the `ProducerTemplate.request*` methods must be used.

```
import akka.actor.Actor
class MyActor extends Actor {
  def receive = {
    case msg =>
      val template = CamelExtension(context.system).template
      sender() ! template.requestBody("direct:news", msg)
  }
}
```

5.9.4 Asynchronous routing

In-out message exchanges between endpoints and actors are designed to be asynchronous. This is the case for both, consumer and producer actors.

- A consumer endpoint sends request messages to its consumer actor using the `!` (tell) operator and the actor returns responses with `sender !` once they are ready.
- A producer actor sends request messages to its endpoint using Camel's asynchronous routing engine. Asynchronous responses are wrapped and added to the producer actor's mailbox for later processing. By default, response messages are returned to the initial sender but this can be overridden by Producer implementations (see also description of the `routeResponse` method in *Custom Processing*).

However, asynchronous two-way message exchanges, without allocating a thread for the full duration of exchange, cannot be generically supported by Camel's asynchronous routing engine alone. This must be supported by the individual Camel components (from which endpoints are created) as well. They must be able to suspend any work started for request processing (thereby freeing threads to do other work) and resume processing when the response is ready. This is currently the case for a subset of components such as the Jetty component. All other Camel components can still be used, of course, but they will cause allocation of a thread for the duration of an in-out message exchange. There's also *Examples* that implements both, an asynchronous consumer and an asynchronous producer, with the jetty component.

If the used Camel component is blocking it might be necessary to use a separate *dispatcher* for the producer. The Camel processor is invoked by a child actor of the producer and the dispatcher can be defined in the deployment section of the configuration. For example, if your producer actor has path `/user/integration/output` the dispatcher of the child actor can be defined with:

```
akka.actor.deployment {
  /integration/output/* {
    dispatcher = my-dispatcher
  }
}
```

5.9.5 Custom Camel routes

In all the examples so far, routes to consumer actors have been automatically constructed by akka-camel, when the actor was started. Although the default route construction templates, used by akka-camel internally, are sufficient for most use cases, some applications may require more specialized routes to actors. The akka-camel module provides two mechanisms for customizing routes to actors, which will be explained in this section. These are:

- Usage of *Akka Camel components* to access actors. Any Camel route can use these components to access Akka actors.
- *Intercepting route construction* to actors. This option gives you the ability to change routes that have already been added to Camel. Consumer actors have a hook into the route definition process which can be used to change the route.

Akka Camel components

Akka actors can be accessed from Camel routes using the actor Camel component. This component can be used to access any Akka actor (not only consumer actors) from Camel routes, as described in the following sections.

Access to actors

To access actors from custom Camel routes, the actor Camel component should be used. It fully supports Camel's asynchronous routing engine.

This component accepts the following endpoint URI format:

- `[<actor-path>]?<options>`

where `<actor-path>` is the `ActorPath` to the actor. The `<options>` are name-value pairs separated by `&` (i.e. `name1=value1&name2=value2&...`).

URI options

The following URI options are supported:

Name	Type	Default	Description
replyTimeout	Duration	false	The reply timeout, specified in the same way that you use the duration in akka, for instance `10 seconds` except that in the url it is handy to use a + between the amount and the unit, like for example `200+millis` See also *Consumer timeout*.
autoAck	Boolean	true	If set to true, in-only message exchanges are auto-acknowledged when the message is added to the actor's mailbox. If set to false, actors must acknowledge the receipt of the message. See also *Delivery acknowledgements*.

Here's an actor endpoint URI example containing an actor path:

```
akka://some-system/user/myconsumer?autoAck=false&replyTimeout=100+millis
```

In the following example, a custom route to an actor is created, using the actor's path. the akka camel package contains an implicit `toActorRouteDefinition` that allows for a route to reference an `ActorRef` directly as shown in the below example, The route starts from a Jetty endpoint and ends at the target actor.

```
import akka.actor.{ Props, ActorSystem, Actor, ActorRef }
import akka.camel.{ CamelMessage, CamelExtension }
import org.apache.camel.builder.RouteBuilder
import akka.camel._
class Responder extends Actor {
  def receive = {
    case msg: CamelMessage =>
      sender() ! (msg.mapBody {
        body: String => "received %s" format body
      })
  }
}

class CustomRouteBuilder(system: ActorSystem, responder: ActorRef)
  extends RouteBuilder {
  def configure {
    from("jetty:http://localhost:8877/camel/custom").to(responder)
  }
}
val system = ActorSystem("some-system")
val camel = CamelExtension(system)
val responder = system.actorOf(Props[Responder], name = "TestResponder")
camel.context.addRoutes(new CustomRouteBuilder(system, responder))
```

When a message is received on the jetty endpoint, it is routed to the Responder actor, which in return replies back to the client of the HTTP request.

Intercepting route construction

The previous section, *Akka Camel components*, explained how to setup a route to an actor manually. It was the application's responsibility to define the route and add it to the current CamelContext. This section explains a more convenient way to define custom routes: akka-camel is still setting up the routes to consumer actors (and adds these routes to the current CamelContext) but applications can define extensions to these routes. Extensions can be defined with Camel's Java DSL or Scala DSL. For example, an extension could be a custom error handler that redelivers messages from an endpoint to an actor's bounded mailbox when the mailbox was full.

The following examples demonstrate how to extend a route to a consumer actor for handling exceptions thrown by that actor.

```
import akka.camel.Consumer

import org.apache.camel.builder.Builder
import org.apache.camel.model.RouteDefinition

class ErrorThrowingConsumer(override val endpointUri: String) extends Consumer {
  def receive = {
    case msg: CamelMessage => throw new Exception("error: %s" format msg.body)
  }
  override def onRouteDefinition = (rd) => rd.onException(classOf[Exception]).
    handled(true).transform(Builder.exceptionMessage).end

  final override def preRestart(reason: Throwable, message: Option[Any]) {
    sender() ! Failure(reason)
  }
}
```

The above ErrorThrowingConsumer sends the Failure back to the sender in preRestart because the Exception that is thrown in the actor would otherwise just crash the actor, by default the actor would be restarted, and the response would never reach the client of the Consumer.

The akka-camel module creates a RouteDefinition instance by calling from(endpointUri) on a Camel RouteBuilder (where endpointUri is the endpoint URI of the consumer actor) and passes that instance as argument to the route definition handler *). The route definition handler then extends the route and returns a ProcessorDefinition (in the above example, the ProcessorDefinition returned by the end method. See the org.apache.camel.model package for details). After executing the route definition handler, akka-camel finally calls a to(targetActorUri) on the returned ProcessorDefinition to complete the route to the consumer actor (where targetActorUri is the actor component URI as described in *Access to actors*). If the actor cannot be found, a *ActorNotRegisteredException* is thrown.

*) Before passing the RouteDefinition instance to the route definition handler, akka-camel may make some further modifications to it.

5.9.6 Examples

The Typesafe Activator tutorial named Akka Camel Samples with Scala contains 3 samples:

- Asynchronous routing and transformation - This example demonstrates how to implement consumer and producer actors that support *Asynchronous routing* with their Camel endpoints.
- Custom Camel route - Demonstrates the combined usage of a `Producer` and a `Consumer` actor as well as the inclusion of a custom Camel route.
- Quartz Scheduler Example - Showing how simple is to implement a cron-style scheduler by using the Camel Quartz component

5.9.7 Configuration

There are several configuration properties for the Camel module, please refer to the *reference configuration*.

5.9.8 Additional Resources

For an introduction to akka-camel 2, see also the Peter Gabryanczyk's talk Migrating akka-camel module to Akka 2.x.

For an introduction to akka-camel 1, see also the Appendix E - Akka and Camel (pdf) of the book Camel in Action.

Other, more advanced external articles (for version 1) are:

- Akka Consumer Actors: New Features and Best Practices
- Akka Producer Actors: New Features and Best Practices

CHAPTER

SIX

UTILITIES

6.1 Event Bus

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:

```
/**
 * Attempts to register the subscriber to the specified Classifier
 * @return true if successful and false if not (because it was already
 *   subscribed to that Classifier, or otherwise)
 */
def subscribe(subscriber: Subscriber, to: Classifier): Boolean

/**
 * Attempts to deregister the subscriber from the specified Classifier
 * @return true if successful and false if not (because it wasn't subscribed
 *   to that Classifier, or otherwise)
 */
def unsubscribe(subscriber: Subscriber, from: Classifier): Boolean

/**
 * Attempts to deregister the subscriber from all Classifiers it may be subscribed to
 */
def unsubscribe(subscriber: Subscriber): Unit

/**
 * Publishes the specified Event to this bus
 */
def publish(event: Event): Unit
```

Note: Please note that the EventBus does not preserve the sender of the published messages. If you need a reference to the original sender you have to provide it inside the message.

This mechanism is used in different places within Akka, e.g. 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.

6.1.1 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 illustrated with the following example:

```
import akka.event.EventBus
import akka.event.LookupClassification

case class MsgEnvelope(topic: String, payload: Any)

/**
 * Publishes the payload of the MsgEnvelope when the topic of the
 * MsgEnvelope equals the String specified when subscribing.
 */
class LookupBusImpl extends EventBus with LookupClassification {
  type Event = MsgEnvelope
  type Classifier = String
  type Subscriber = ActorRef

  // is used for extracting the classifier from the incoming events
  override protected def classify(event: Event): Classifier = event.topic

  // will be invoked for each event for all subscribers which registered themselves
  // for the event's classifier
  override protected def publish(event: Event, subscriber: Subscriber): Unit = {
    subscriber ! event.payload
  }

  // must define a full order over the subscribers, expressed as expected from
  // `java.lang.Comparable.compare`
  override protected def compareSubscribers(a: Subscriber, b: Subscriber): Int =
    a.compareTo(b)

  // determines the initial size of the index data structure
  // used internally (i.e. the expected number of different classifiers)
  override protected def mapSize: Int = 128

}
```

A test for this implementation may look like this:

```
val lookupBus = new LookupBusImpl
lookupBus.subscribe(testActor, "greetings")
lookupBus.publish(MsgEnvelope("time", System.currentTimeMillis()))
lookupBus.publish(MsgEnvelope("greetings", "hello"))
expectMsg("hello")
```

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 necessary methods to be implemented are illustrated with the following example:

```
import akka.util.Subclassification

class StartsWithSubclassification extends Subclassification[String] {
  override def isEqual(x: String, y: String): Boolean =
    x == y

  override def isSubclass(x: String, y: String): Boolean =
    x.startsWith(y)
}

import akka.event.SubchannelClassification

/**
 * Publishes the payload of the MsgEnvelope when the topic of the
 * MsgEnvelope starts with the String specified when subscribing.
 */
class SubchannelBusImpl extends EventBus with SubchannelClassification {
  type Event = MsgEnvelope
  type Classifier = String
  type Subscriber = ActorRef

  // Subclassification is an object providing `isEqual` and `isSubclass`
  // to be consumed by the other methods of this classifier
  override protected val subclassification: Subclassification[Classifier] =
    new StartsWithSubclassification

  // is used for extracting the classifier from the incoming events
  override protected def classify(event: Event): Classifier = event.topic

  // will be invoked for each event for all subscribers which registered
  // themselves for the event's classifier
  override protected def publish(event: Event, subscriber: Subscriber): Unit = {
    subscriber ! event.payload
  }
}
```

A test for this implementation may look like this:

```
val subchannelBus = new SubchannelBusImpl
subchannelBus.subscribe(testActor, "abc")
subchannelBus.publish(MsgEnvelope("xyzabc", "x"))
subchannelBus.publish(MsgEnvelope("bcdef", "b"))
subchannelBus.publish(MsgEnvelope("abc", "c"))
expectMsg("c")
subchannelBus.publish(MsgEnvelope("abcdef", "d"))
expectMsg("d")
```

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 necessary methods to be implemented are illustrated with the following example:

```
import akka.event.ScanningClassification

/**
 * Publishes String messages with length less than or equal to the length
 * specified when subscribing.
 */
class ScanningBusImpl extends EventBus with ScanningClassification {
  type Event = String
  type Classifier = Int
  type Subscriber = ActorRef

  // is needed for determining matching classifiers and storing them in an
  // ordered collection
  override protected def compareClassifiers(a: Classifier, b: Classifier): Int =
    if (a < b) -1 else if (a == b) 0 else 1

  // is needed for storing subscribers in an ordered collection
  override protected def compareSubscribers(a: Subscriber, b: Subscriber): Int =
    a.compareTo(b)

  // 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
  override protected def matches(classifier: Classifier, event: Event): Boolean =
    event.length <= classifier

  // will be invoked for each event for all subscribers which registered themselves
  // for a classifier matching this event
  override protected def publish(event: Event, subscriber: Subscriber): Unit = {
    subscriber ! event
  }
}
```

A test for this implementation may look like this:

```
val scanningBus = new ScanningBusImpl
scanningBus.subscribe(testActor, 3)
scanningBus.publish("xyzabc")
scanningBus.publish("ab")
expectMsg("ab")
scanningBus.publish("abc")
expectMsg("abc")
```

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

Actor Classification

This classification was originally developed specifically for implementing *DeathWatch*: subscribers as well as classifiers are of type `ActorRef`.

The necessary methods to be implemented are illustrated with the following example:

```
import akka.event.ActorEventBus
import akka.event.ActorClassification
import akka.event.ActorClassifier

case class Notification(ref: ActorRef, id: Int)
```

```
class ActorBusImpl extends ActorEventBus with ActorClassifier with ActorClassification {
  type Event = Notification

  // is used for extracting the classifier from the incoming events
  override protected def classify(event: Event): ActorRef = event.ref

  // determines the initial size of the index data structure
  // used internally (i.e. the expected number of different classifiers)
  override protected def mapSize: Int = 128
}
```

A test for this implementation may look like this:

```
val observer1 = TestProbe().ref
val observer2 = TestProbe().ref
val probe1 = TestProbe()
val probe2 = TestProbe()
val subscriber1 = probe1.ref
val subscriber2 = probe2.ref
val actorBus = new ActorBusImpl
actorBus.subscribe(subscriber1, observer1)
actorBus.subscribe(subscriber2, observer1)
actorBus.subscribe(subscriber2, observer2)
actorBus.publish(Notification(observer1, 100))
probe1.expectMsg(Notification(observer1, 100))
probe2.expectMsg(Notification(observer1, 100))
actorBus.publish(Notification(observer2, 101))
probe2.expectMsg(Notification(observer2, 101))
probe1.expectNoMsg(500.millis)
```

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

6.1.2 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 `RemotingLifecycleEvent`). The following example demonstrates how a simple subscription works:

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

class Listener extends Actor {
  def receive = {
    case d: DeadLetter => println(d)
  }
}
val listener = system.actorOf(Props(classOf[Listener], this))
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 {
  loggers = ["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 not 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.

6.2 Logging

Logging in Akka is not tied to a specific logging backend. By default log messages are printed to STDOUT, but you can plug-in a SLF4J logger or your own logger. Logging is performed asynchronously to ensure that logging has minimal performance impact. Logging generally means IO and locks, which can slow down the operations of your code if it was performed synchronously.

6.2.1 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.

Logging of Dead Letters

By default messages sent to dead letters are logged at info level. Existence of dead letters does not necessarily indicate a problem, but it might be, and therefore they are logged by default. After a few messages this logging is turned off, to avoid flooding the logs. You can disable this logging completely or adjust how many dead letters that are logged. During system shutdown it is likely that you see dead letters, since pending messages in the actor mailboxes are sent to dead letters. You can also disable logging of dead letters during shutdown.

```
akka {
  log-dead-letters = 10
  log-dead-letters-during-shutdown = on
}
```

To customize the logging further or take other actions for dead letters you can subscribe to the *Event Stream*.

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 user-level messages then wrap your actors' behaviors with `akka.event.LoggingReceive` and enable the `receive` option:

```
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 et.c.)
      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 not logged
    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 logged
    log-received-messages = on
  }
}
```

If you want to see message types with payload size in bytes larger than a specified limit at INFO log level:

```
akka {
  remote {
    # Logging of message types with payload size in bytes larger than
    # this value. Maximum detected size per message type is logged once,
    # with an increase threshold of 10%.
    # By default this feature is turned off. Activate it by setting the property to
    # a value in bytes, such as 1000b. Note that for all messages larger than this
    # limit there will be extra performance and scalability cost.
    log-frame-size-exceeding = 1000b
  }
}
```

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.

Turn Off Logging

To turn off logging you can configure the log levels to be `OFF` like this.

```
akka {
  stdout-loglevel = "OFF"
  loglevel = "OFF"
}
```

The `stdout-loglevel` is only in effect during system startup and shutdown, and setting it to `OFF` as well, ensures that nothing gets logged during system startup or shutdown.

6.2.2 Loggers

Logging is performed asynchronously through an event bus. Log events are processed by an event handler actor and it will receive the log events in the same order as they were emitted.

You can configure which event handlers are created at system start-up and listen to logging events. That is done using the `loggers` element in the *Configuration*. Here you can also define the log level.

```
akka {
  # Loggers to register at boot time (akka.event.Logging$DefaultLogger logs
  # to STDOUT)
  loggers = ["akka.event.Logging$DefaultLogger"]
  # Options: OFF, 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 *Logging to stdout during startup and shutdown* logger 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)        => // ...
  }
}
```

6.2.3 Logging to stdout during startup and shutdown

When the actor system is starting up and shutting down the configured `loggers` are not used. Instead log messages are printed to stdout (System.out). The default log level for this stdout logger is `WARNING` and it can be silenced completely by setting `akka.stdout-loglevel=OFF`.

6.2.4 SLF4J

Akka provides a logger 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" % "1.0.13"
```

You need to enable the Slf4jLogger in the 'loggers' 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 {
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  loglevel = "DEBUG"
}
```

One gotcha is that the timestamp is attributed in the event handler, not when actually doing the logging.

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 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.

More accurate timestamps for log output in MDC

Akka's logging is asynchronous which means that the timestamp of a log entry is taken from when the underlying logger implementation is called, which can be surprising at first. If you want to more accurately output the timestamp, use the MDC attribute `akkaTimestamp`:

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

MDC values defined by the application

One useful feature available in Slf4j is MDC, Akka has a way for let the application specify custom values, you just need to get a specialized `LoggingAdapter`, the `DiagnosticLoggingAdapter`. In order to get it you will use the factory receiving an Actor as logSource:

```
// Within your Actor
val log: DiagnosticLoggingAdapter = Logging(this);
```

Once you have the logger, you just need to add the custom values before you log something. This way, the values will be put in the SLF4J MDC right before appending the log and removed after.

Note: The cleanup (removal) should be done in the actor at the end, otherwise, next message will log with same mdc values, if it is not set to a new map. Use `log.clearMDC()`.

```
val mdc = Map("requestId" -> 1234, "visitorId" -> 5678)
log.mdc(mdc)

// Log something
log.info("Starting new request")

log.clearMDC()
```

For convenience you can mixin the `log` member into actors, instead of defining it as above. This trait also lets you override `def mdc(msg: Any): MDC` for specifying MDC values depending on current message and lets you forget about the cleanup as well, since it already does it for you.

```
import Logging.MDC

case class Req(work: String, visitorId: Int)

class MdcActorMixin extends Actor with akka.actor.DiagnosticActorLogging {
  var reqId = 0

  override def mdc(currentMessage: Any): MDC = {
    reqId += 1
    val always = Map("requestId" -> reqId)
    val perMessage = currentMessage match {
      case r: Req => Map("visitorId" -> r.visitorId)
      case _      => Map()
    }
    always ++ perMessage
  }

  def receive: Receive = {
    case r: Req => {
      log.info(s"Starting new request: ${r.work}")
    }
  }
}
```

Now, the values will be available in the MDC, so you can use them in the layout pattern:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>
      %-5level %logger{36} [req: %X{requestId}, visitor: %X{visitorId}] - %msg%n
    </pattern>
  </encoder>
</appender>
```

6.3 Scheduler

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.

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 job buckets which are emptied according to a fixed schedule. It does not execute tasks at the exact time, but on every tick, it will run everything that is (over)due. The accuracy of the default Scheduler can be modified by the `akka.scheduler.tick-duration` configuration property.

6.3.1 Some examples

```
import akka.actor.Actor
import akka.actor.Props
import scala.concurrent.duration._

    //Use the system's dispatcher as ExecutionContext
    import system.dispatcher

    //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 a message to the testActor) after 50ms
system.scheduler.scheduleOnce(50 milliseconds) {
  testActor ! System.currentTimeMillis
}
```

```
val Tick = "tick"
class TickActor extends Actor {
  def receive = {
    case Tick => //Do something
  }
}
val tickActor = system.actorOf(Props(classOf[TickActor], this))
//Use system's dispatcher as ExecutionContext
import system.dispatcher

//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()
```

Warning: If you schedule functions or Runnable instances you should be extra careful to not close over unstable references. In practice this means not using `this` inside the closure in the scope of an Actor instance, not accessing `sender()` directly and not calling the methods of the Actor instance directly. If you need to schedule an invocation schedule a message to `self` instead (containing the necessary parameters) and then call the method when the message is received.

6.3.2 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
```

6.3.3 The Scheduler interface

The actual scheduler implementation is loaded reflectively upon `ActorSystem` start-up, which means that it is possible to provide a different one using the `akka.scheduler.implementation` configuration property. The referenced class must implement the following 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.
 *
 * Scheduler implementation are loaded reflectively at ActorSystem start-up
 * with the following constructor arguments:
 *  1) the system's com.typesafe.config.Config (from system.settings.config)
 *  2) a akka.event.LoggingAdapter
 *  3) a java.util.concurrent.ThreadFactory
 */
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
   * interval=Duration(500, TimeUnit.MILLISECONDS)
   *
   * Java & Scala API
   */
  final def schedule(
    initialDelay: FiniteDuration,
    interval: FiniteDuration,
    receiver: ActorRef,
    message: Any)(implicit executor: ExecutionContext,
                  sender: ActorRef = Actor.noSender): Cancellable =
    schedule(initialDelay, interval, new Runnable {
      def run = {
        receiver ! message
        if (receiver.isTerminated)
          throw new SchedulerException("timer active for terminated actor")
```

```
    }
  })

/**
 * 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 interval = Duration(100, TimeUnit.MILLISECONDS)
 *
 * Scala API
 */
final def schedule(
  initialDelay: FiniteDuration,
  interval: FiniteDuration)(f: ⇒ Unit)(
    implicit executor: ExecutionContext): Cancellable =
  schedule(initialDelay, interval, new Runnable { override def run = f })

/**
 * 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 interval = Duration(100, TimeUnit.MILLISECONDS)
 *
 * Java API
 */
def schedule(
  initialDelay: FiniteDuration,
  interval: FiniteDuration,
  runnable: Runnable)(implicit executor: ExecutionContext): 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
 */
final def scheduleOnce(
  delay: FiniteDuration,
  receiver: ActorRef,
  message: Any)(implicit executor: ExecutionContext,
                sender: ActorRef = Actor.noSender): Cancellable =
  scheduleOnce(delay, new Runnable {
    override def run = receiver ! message
  })

/**
 * 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
 */
final def scheduleOnce(delay: FiniteDuration)(f: ⇒ Unit)(
  implicit executor: ExecutionContext): Cancellable =
  scheduleOnce(delay, new Runnable { override def run = f })

/**
 * 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: FiniteDuration,
```

```
    runnable: Runnable)(implicit executor: ExecutionContext): Cancellable

  /**
   * The maximum supported task frequency of this scheduler, i.e. the inverse
   * of the minimum time interval between executions of a recurring task, in Hz.
   */
  def maxFrequency: Double

}
```

6.3.4 The Cancellable interface

Scheduling a task will result in a `Cancellable` (or throw an `IllegalStateException` if attempted after the scheduler's shutdown). 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. Check the return value of `cancel` to detect whether the scheduled task was canceled or will (eventually) have run.

```
/**
 * 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 and returns true if that was successful.
   * If this cancellable was (concurrently) cancelled already, then this method
   * will return false although isCancelled will return true.
   *
   * Java & Scala API
   */
  def cancel(): Boolean

  /**
   * Returns true if and only if this Cancellable has been successfully cancelled
   *
   * Java & Scala API
   */
  def isCancelled: Boolean
}
```

6.4 Duration

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

6.4.1 Finite vs. Infinite

Since trying to convert an infinite duration into a concrete time unit like seconds will throw an exception, there are different types available for distinguishing the two kinds at compile time:

- `FiniteDuration` is guaranteed to be finite, calling `toNanos` and friends is safe
- `Duration` can be finite or infinite, so this type should only be used when finite-ness does not matter; this is a supertype of `FiniteDuration`

6.4.2 Scala

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

```
import scala.concurrent.duration._

val fivesec = 5.seconds
val threemillis = 3.millis
val diff = fivesec - threemillis
assert(diff < fivesec)
val fourmillis = threemillis * 4 / 3 // 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.

6.4.3 Java

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

```
import scala.concurrent.duration.Duration;
import scala.concurrent.duration.Deadline;
```

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

6.4.4 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
val rest = deadline.timeLeft
```

In Java you create these from durations:

```
final Deadline deadline = Duration.create(10, "seconds").fromNow();
final Duration rest = deadline.timeLeft();
```

6.5 Circuit Breaker

6.5.1 Why are they used?

A circuit breaker is used to provide stability and prevent cascading failures in distributed systems. These should be used in conjunction with judicious timeouts at the interfaces between remote systems to prevent the failure of a single component from bringing down all components.

As an example, we have a web application interacting with a remote third party web service. Let's say the third party has oversold their capacity and their database melts down under load. Assume that the database fails in such

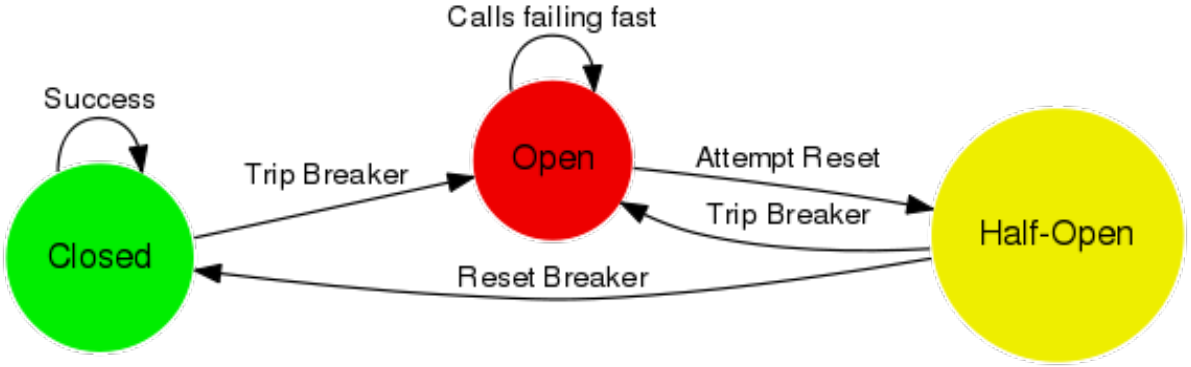
a way that it takes a very long time to hand back an error to the third party web service. This in turn makes calls fail after a long period of time. Back to our web application, the users have noticed that their form submissions take much longer seeming to hang. Well the users do what they know to do which is use the refresh button, adding more requests to their already running requests. This eventually causes the failure of the web application due to resource exhaustion. This will affect all users, even those who are not using functionality dependent on this third party web service.

Introducing circuit breakers on the web service call would cause the requests to begin to fail-fast, letting the user know that something is wrong and that they need not refresh their request. This also confines the failure behavior to only those users that are using functionality dependent on the third party, other users are no longer affected as there is no resource exhaustion. Circuit breakers can also allow savvy developers to mark portions of the site that use the functionality unavailable, or perhaps show some cached content as appropriate while the breaker is open.

The Akka library provides an implementation of a circuit breaker called `akka.pattern.CircuitBreaker` which has the behavior described below.

6.5.2 What do they do?

- **During normal operation, a circuit breaker is in the *Closed* state:**
  - Exceptions or calls exceeding the configured *callTimeout* increment a failure counter
  - Successes reset the failure count to zero
  - When the failure counter reaches a *maxFailures* count, the breaker is tripped into *Open* state
- **While in *Open* state:**
  - All calls fail-fast with a `CircuitBreakerOpenException`
  - After the configured *resetTimeout*, the circuit breaker enters a *Half-Open* state
- **In *Half-Open* state:**
  - The first call attempted is allowed through without failing fast
  - All other calls fail-fast with an exception just as in *Open* state
  - If the first call succeeds, the breaker is reset back to *Closed* state
  - If the first call fails, the breaker is tripped again into the *Open* state for another full *resetTimeout*
- **State transition listeners:**
  - Callbacks can be provided for every state entry via *onOpen*, *onClose*, and *onHalfOpen*
  - These are executed in the `ExecutionContext` provided.



6.5.3 Examples

Initialization

Here's how a `CircuitBreaker` would be configured for:

- 5 maximum failures
- a call timeout of 10 seconds
- a reset timeout of 1 minute

Scala

```
import scala.concurrent.duration._
import akka.pattern.CircuitBreaker
import akka.pattern.pipe
import akka.actor.Actor
import akka.actor.ActorLogging
import scala.concurrent.Future
import akka.event.Logging

class DangerousActor extends Actor with ActorLogging {
  import context.dispatcher

  val breaker =
    new CircuitBreaker(context.system.scheduler,
      maxFailures = 5,
      callTimeout = 10.seconds,
      resetTimeout = 1.minute).onOpen(notifyMeOnOpen())

  def notifyMeOnOpen(): Unit =
    log.warning("My CircuitBreaker is now open, and will not close for one minute")
```

Java

```
import akka.actor.UntypedActor;
import scala.concurrent.Future;
import akka.event.LoggingAdapter;
import scala.concurrent.duration.Duration;
import akka.pattern.CircuitBreaker;
import akka.event.Logging;

import static akka.pattern.Patterns.pipe;
import static akka.dispatch.Futures.future;

import java.util.concurrent.Callable;

public class DangerousJavaActor extends UntypedActor {

  private final CircuitBreaker breaker;
  private final LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  public DangerousJavaActor() {
    this.breaker = new CircuitBreaker(
      getContext().dispatcher(), getContext().system().scheduler(),
      5, Duration.create(10, "s"), Duration.create(1, "m"))
      .onOpen(new Runnable() {
        public void run() {
          notifyMeOnOpen();
```

```
        }
      });
  }

  public void notifyMeOnOpen() {
    log.warning("My CircuitBreaker is now open, and will not close for one minute");
  }
```

Call Protection

Here's how the `CircuitBreaker` would be used to protect an asynchronous call as well as a synchronous one:

Scala

```
def dangerousCall: String = "This really isn't that dangerous of a call after all"

def receive = {
  case "is my middle name" =>
    breaker.withCircuitBreaker(Future(dangerousCall)) pipeTo sender()
  case "block for me" =>
    sender() ! breaker.withSyncCircuitBreaker(dangerousCall)
}
```

Java

```
public String dangerousCall() {
  return "This really isn't that dangerous of a call after all";
}

@Override
public void onReceive(Object message) {
  if (message instanceof String) {
    String m = (String) message;
    if ("is my middle name".equals(m)) {
      final Future<String> f = future(
        new Callable<String>() {
          public String call() {
            return dangerousCall();
          }
        }, getContext().dispatcher());

      pipe(breaker.callWithCircuitBreaker(
        new Callable<Future<String>>() {
          public Future<String> call() throws Exception {
            return f;
          }
        }), getContext().dispatcher()).to(getSender());
    }
    if ("block for me".equals(m)) {
      getSender().tell(breaker
        .callWithSyncCircuitBreaker(
          new Callable<String>() {
            @Override
            public String call() throws Exception {
              return dangerousCall();
            }
          }), getSelf());
    }
```

```
  }
}
```

Note: Using the `CircuitBreaker` companion object's *apply* or *create* methods will return a `CircuitBreaker` where callbacks are executed in the caller's thread. This can be useful if the asynchronous `Future` behavior is unnecessary, for example invoking a synchronous-only API.

6.6 Akka Extensions

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.

6.6.1 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.ActorSystem
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

  /**
```

```
   * Java API: retrieve the Count extension for the given system.
   */
  override def get(system: ActorSystem): CountExtensionImpl = super.get(system)
}
```

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!

6.6.2 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 = ["docs.extension.CountExtension"]
}
```

6.6.3 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.ActorSystem
import akka.actor.Extension
import akka.actor.ExtensionId
import akka.actor.ExtensionIdProvider
import akka.actor.ExtendedActorSystem
import scala.concurrent.duration.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.timeout"),
      TimeUnit.MILLISECONDS)
}
object Settings extends ExtensionId[SettingsImpl] with ExtensionIdProvider {

  override def lookup = Settings

  override def createExtension(system: ExtendedActorSystem) =
    new SettingsImpl(system.settings.config)

  /**
   * Java API: retrieve the Settings extension for the given system.
   */
  override def get(system: ActorSystem): SettingsImpl = super.get(system)
}
```

Use it:

```
class MyActor extends Actor {
  val settings = Settings(context.system)
  val connection = connect(settings.DbUri, settings.CircuitBreakerTimeout)
```

6.7 Microkernel

The purpose of the Akka Microkernel is to offer a bundling mechanism so that you can distribute an Akka application as a single payload, without the need to run in a Java Application Server or manually having to create a launcher script.

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 and additional dependencies in the `lib` directory to have them automatically loaded and placed on the classpath.

To start the kernel use the scripts in the `bin` directory, passing the boot classes for your application.

The start script adds `config` directory first in the classpath, followed by `lib/*`. It runs java with main class `akka.kernel.Main` and the supplied Bootable class as argument.

Example 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):

```
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()
  }
}
```

CHAPTER

SEVEN

HOWTO: COMMON PATTERNS

This section lists common actor patterns which have been found to be useful, elegant or instructive. Anything is welcome, example topics being message routing strategies, supervision patterns, restart handling, etc. As a special bonus, additions to this section are marked with the contributor's name, and it would be nice if every Akka user who finds a recurring pattern in his or her code could share it for the profit of all. Where applicable it might also make sense to add to the `akka.pattern` package for creating an OTP-like library.

7.1 Throttling Messages

Contributed by: Kaspar Fischer

"A message throttler that ensures that messages are not sent out at too high a rate."

The pattern is described in Throttling Messages in Akka 2.

7.2 Balancing Workload Across Nodes

Contributed by: Derek Wyatt

"Often times, people want the functionality of the BalancingDispatcher with the stipulation that the Actors doing the work have distinct Mailboxes on remote nodes. In this post we'll explore the implementation of such a concept."

The pattern is described Balancing Workload across Nodes with Akka 2.

7.3 Work Pulling Pattern to throttle and distribute work, and prevent mailbox overflow

Contributed by: Michael Pollmeier

"This pattern ensures that your mailboxes don't overflow if creating work is fast than actually doing it – which can lead to out of memory errors when the mailboxes eventually become too full. It also let's you distribute work around your cluster, scale dynamically scale and is completely non-blocking. This pattern is a specialisation of the above 'Balancing Workload Pattern'."

The pattern is described Work Pulling Pattern to prevent mailbox overflow, throttle and distribute work.

7.4 Ordered Termination

Contributed by: Derek Wyatt

"When an Actor stops, its children stop in an undefined order. Child termination is asynchronous and thus non-deterministic.

If an Actor has children that have order dependencies, then you might need to ensure a particular shutdown order of those children so that their postStop() methods get called in the right order."

The pattern is described An Akka 2 Terminator.

7.5 Akka AMQP Proxies

Contributed by: Fabrice Drouin

""AMQP proxies" is a simple way of integrating AMQP with Akka to distribute jobs across a network of computing nodes. You still write "local" code, have very little to configure, and end up with a distributed, elastic, fault-tolerant grid where computing nodes can be written in nearly every programming language."

The pattern is described Akka AMQP Proxies.

7.6 Shutdown Patterns in Akka 2

Contributed by: Derek Wyatt

"How do you tell Akka to shut down the ActorSystem when everything's finished? It turns out that there's no magical flag for this, no configuration setting, no special callback you can register for, and neither will the illustrious shutdown fairy grace your application with her glorious presence at that perfect moment. She's just plain mean.

In this post, we'll discuss why this is the case and provide you with a simple option for shutting down "at the right time", as well as a not-so-simple-option for doing the exact same thing."

The pattern is described Shutdown Patterns in Akka 2.

7.7 Distributed (in-memory) graph processing with Akka

Contributed by: Adelbert Chang

"Graphs have always been an interesting structure to study in both mathematics and computer science (among other fields), and have become even more interesting in the context of online social networks such as Facebook and Twitter, whose underlying network structures are nicely represented by graphs."

The pattern is described Distributed In-Memory Graph Processing with Akka.

7.8 Case Study: An Auto-Updating Cache Using Actors

Contributed by: Eric Pederson

"We recently needed to build a caching system in front of a slow backend system with the following requirements:

The data in the backend system is constantly being updated so the caches need to be updated every N minutes. Requests to the backend system need to be throttled. The caching system we built used Akka actors and Scala's support for functions as first class objects."

The pattern is described Case Study: An Auto-Updating Cache using Actors.

7.9 Discovering message flows in actor systems with the Spider Pattern

Contributed by: Raymond Roestenburg

"Building actor systems is fun but debugging them can be difficult, you mostly end up browsing through many log files on several machines to find out what's going on. I'm sure you have browsed through logs and thought, "Hey, where did that message go?", "Why did this message cause that effect" or "Why did this actor never get a message?"

This is where the Spider pattern comes in."

The pattern is described Discovering Message Flows in Actor System with the Spider Pattern.

7.10 Scheduling Periodic Messages

This pattern describes how to schedule periodic messages to yourself in two different ways.

The first way is to set up periodic message scheduling in the constructor of the actor, and cancel that scheduled sending in `postStop` or else we might have multiple registered message sends to the same actor.

Note: With this approach the scheduled periodic message send will be restarted with the actor on restarts. This also means that the time period that elapses between two tick messages during a restart may drift off based on when you restart the scheduled message sends relative to the time that the last message was sent, and how long the initial delay is. Worst case scenario is `interval` plus `initialDelay`.

```
class ScheduleInConstructor extends Actor {
  import context.dispatcher
  val tick =
    context.system.scheduler.schedule(500 millis, 1000 millis, self, "tick")

  override def postStop() = tick.cancel()

  def receive = {
    case "tick" =>
      // do something useful here
  }
}
```

The second variant sets up an initial one shot message send in the `preStart` method of the actor, and the then the actor when it receives this message sets up a new one shot message send. You also have to override `postRestart` so we don't call `preStart` and schedule the initial message send again.

Note: With this approach we won't fill up the mailbox with tick messages if the actor is under pressure, but only schedule a new tick message when we have seen the previous one.

```
class ScheduleInReceive extends Actor {
  import context._

  override def preStart() =
    system.scheduler.scheduleOnce(500 millis, self, "tick")

  // override postRestart so we don't call preStart and schedule a new message
  override def postRestart(reason: Throwable) = {}

  def receive = {
    case "tick" =>
      // send another periodic tick after the specified delay
```

```
      system.scheduler.scheduleOnce(1000 millis, self, "tick")
      // do something useful here
  }
}
```

7.11 Template Pattern

Contributed by: N. N.

This is an especially nice pattern, since it does even come with some empty example code:

```
class ScalaTemplate {
  println("Hello, Template!")
  // uninteresting stuff ...
}
```

Note: Spread the word: this is the easiest way to get famous!

Please keep this pattern at the end of this file.

CHAPTER

EIGHT

EXPERIMENTAL MODULES

The following modules of Akka are marked as experimental, which means that they are in early access mode, which also means that they are not covered by commercial support. The purpose of releasing them early, as experimental, is to make them easily available and improve based on feedback, or even discover that the module wasn't useful.

An experimental module doesn't have to obey the rule of staying binary compatible between micro releases. Breaking API changes may be introduced in minor releases without notice as we refine and simplify based on your feedback. An experimental module may be dropped in minor releases without prior deprecation.

8.1 Persistence

Akka persistence enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster. The key concept behind Akka persistence is that only changes to an actor's internal state are persisted but never its current state directly (except for optional snapshots). These changes are only ever appended to storage, nothing is ever mutated, which allows for very high transaction rates and efficient replication. Stateful actors are recovered by replaying stored changes to these actors from which they can rebuild internal state. This can be either the full history of changes or starting from a snapshot which can dramatically reduce recovery times. Akka persistence also provides point-to-point communication channels with at-least-once message delivery semantics.

> **Warning:** This module is marked as **"experimental"** as of its introduction in Akka 2.3.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum the binary compatibility guarantee for maintenance releases does not apply to the contents of the `akka.persistence` package.

Akka persistence is inspired by and the official replacement of the eventsourced library. It follows the same concepts and architecture of eventsourced but significantly differs on API and implementation level. See also *Migration Guide Eventsourced to Akka Persistence 2.3.x*

8.1.1 Dependencies

Akka persistence is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-persistence-experimental" % "2.3.2"
```

8.1.2 Architecture

- *Processor*: A processor is a persistent, stateful actor. Messages sent to a processor are written to a journal before its `receive` method is called. When a processor is started or restarted, journaled messages are replayed to that processor, so that it can recover internal state from these messages.

- *View*: A view is a persistent, stateful actor that receives journaled messages that have been written by another processor. A view itself does not journal new messages, instead, it updates internal state only from a processor's replicated message stream.
- *Channel*: Channels are used by processors and views to communicate with other actors. They prevent that replayed messages are redundantly delivered to these actors and provide at-least-once message delivery semantics, also in case of sender and receiver JVM crashes.
- *Journal*: A journal stores the sequence of messages sent to a processor. An application can control which messages are journaled and which are received by the processor without being journaled. The storage backend of a journal is pluggable. The default journal storage plugin writes to the local filesystem, replicated journals are available as Community plugins.
- *Snapshot store*: A snapshot store persists snapshots of a processor's or a view's internal state. Snapshots are used for optimizing recovery times. The storage backend of a snapshot store is pluggable. The default snapshot storage plugin writes to the local filesystem.
- *Event sourcing*. Based on the building blocks described above, Akka persistence provides abstractions for the development of event sourced applications (see section *Event sourcing*)

8.1.3 Processors

A processor can be implemented by extending the `Processor` trait and implementing the `receive` method.

```
import akka.persistence.{ Persistent, PersistenceFailure, Processor }

class MyProcessor extends Processor {
  def receive = {
    case Persistent(payload, sequenceNr) =>
    // message successfully written to journal
    case PersistenceFailure(payload, sequenceNr, cause) =>
    // message failed to be written to journal
    case other =>
    // message not written to journal
  }
}
```

Processors only write messages of type `Persistent` to the journal, others are received without being persisted. When a processor's `receive` method is called with a `Persistent` message it can safely assume that this message has been successfully written to the journal. If a journal fails to write a `Persistent` message then the processor is stopped, by default. If a processor should continue running on persistence failures it must handle `PersistenceFailure` messages. In this case, a processor may want to inform the sender about the failure, so that the sender can re-send the message, if needed.

A `Processor` itself is an `Actor` and can therefore be instantiated with `actorOf`.

```
import akka.actor.Props

val processor = actorOf(Props[MyProcessor], name = "myProcessor")

processor ! Persistent("foo") // will be journaled
processor ! "bar" // will not be journaled
```

Recovery

By default, a processor is automatically recovered on start and on restart by replaying journaled messages. New messages sent to a processor during recovery do not interfere with replayed messages. New messages will only be received by a processor after recovery completes.

Recovery customization

Automated recovery on start can be disabled by overriding `preStart` with an empty implementation.

```
override def preStart() = ()
```

In this case, a processor must be recovered explicitly by sending it a `Recover()` message.

```
processor ! Recover()
```

If not overridden, `preStart` sends a `Recover()` message to `self`. Applications may also override `preStart` to define further `Recover()` parameters such as an upper sequence number bound, for example.

```
override def preStart() {
  self ! Recover(toSequenceNr = 457L)
}
```

Upper sequence number bounds can be used to recover a processor to past state instead of current state. Automated recovery on restart can be disabled by overriding `preRestart` with an empty implementation.

```
override def preRestart(reason: Throwable, message: Option[Any]) = ()
```

Recovery status

A processor can query its own recovery status via the methods

```
def recoveryRunning: Boolean
def recoveryFinished: Boolean
```

Sometimes there is a need for performing additional initialization when the recovery has completed, before processing any other message sent to the processor. The processor can send itself a message from `preStart`. It will be stashed and received after recovery. The mailbox may contain other messages that are queued in front of that message and therefore you need to stash until you receive that message.

```
override def preStart(): Unit = {
  super.preStart()
  self ! "FIRST"
}

def receive = initializing.orElse(active)

def initializing: Receive = {
  case "FIRST" =>
    recoveryCompleted()
    context.become(active)
    unstashAll()
  case other if recoveryFinished =>
    stash()
}

def recoveryCompleted(): Unit = {
  // perform init after recovery, before any other messages
  // ...
}

def active: Receive = {
  case Persistent(msg, _) => //...
}
```

Failure handling

A persistent message that caused an exception will be received again by a processor after restart. To prevent a replay of that message during recovery it can be deleted.

```
override def preRestart(reason: Throwable, message: Option[Any]) {
  message match {
    case Some(p: Persistent) => deleteMessage(p.sequenceNr)
    case _                   =>
  }
  super.preRestart(reason, message)
}
```

Message deletion

A processor can delete a single message by calling the `deleteMessage` method with the sequence number of that message as argument. An optional `permanent` parameter specifies whether the message shall be permanently deleted from the journal or only marked as deleted. In both cases, the message won't be replayed. Later extensions to Akka persistence will allow to replay messages that have been marked as deleted which can be useful for debugging purposes, for example. To delete all messages (journaled by a single processor) up to a specified sequence number, processors should call the `deleteMessages` method.

Identifiers

A processor must have an identifier that doesn't change across different actor incarnations. It defaults to the `String` representation of processor's path without the address part and can be obtained via the `processorId` method.

```
def processorId: String
```

Applications can customize a processor's id by specifying an actor name during processor creation as shown in section *Processors*. This changes that processor's name in its actor hierarchy and hence influences only part of the processor id. To fully customize a processor's id, the `processorId` method must be overridden.

```
override def processorId = "my-stable-processor-id"
```

Overriding `processorId` is the recommended way to generate stable identifiers.

8.1.4 Views

Views can be implemented by extending the `View` trait and implementing the `receive` and the `processorId` methods.

```
class MyView extends View {
  def processorId: String = "some-processor-id"

  def receive: Actor.Receive = {
    case Persistent(payload, sequenceNr) => // ...
  }
}
```

The `processorId` identifies the processor from which the view receives journaled messages. It is not necessary the referenced processor is actually running. Views read messages from a processor's journal directly. When a processor is started later and begins to write new messages, the corresponding view is updated automatically, by default.

Updates

The default update interval of all views of an actor system is configurable:

```
akka.persistence.view.auto-update-interval = 5s
```

`View` implementation classes may also override the `autoUpdateInterval` method to return a custom update interval for a specific view class or view instance. Applications may also trigger additional updates at any time by sending a view an `Update` message.

```
val view = system.actorOf(Props[MyView])
view ! Update(await = true)
```

If the `await` parameter is set to `true`, messages that follow the `Update` request are processed when the incremental message replay, triggered by that update request, completed. If set to `false` (default), messages following the update request may interleave with the replayed message stream. Automated updates always run with `await = false`.

Automated updates of all views of an actor system can be turned off by configuration:

```
akka.persistence.view.auto-update = off
```

Implementation classes may override the configured default value by overriding the `autoUpdate` method. To limit the number of replayed messages per update request, applications can configure a custom `akka.persistence.view.auto-update-replay-max` value or override the `autoUpdateReplayMax` method. The number of replayed messages for manual updates can be limited with the `replayMax` parameter of the `Update` message.

Recovery

Initial recovery of views works in the very same way as for *Processors* (i.e. by sending a `Recover` message to self). The maximum number of replayed messages during initial recovery is determined by `autoUpdateReplayMax`. Further possibilities to customize initial recovery are explained in section *Processors*.

Identifiers

A view must have an identifier that doesn't change across different actor incarnations. It defaults to the `String` representation of the actor path without the address part and can be obtained via the `viewId` method.

Applications can customize a view's id by specifying an actor name during view creation. This changes that view's name in its actor hierarchy and hence influences only part of the view id. To fully customize a view's id, the `viewId` method must be overridden. Overriding `viewId` is the recommended way to generate stable identifiers.

The `viewId` must differ from the referenced `processorId`, unless *Snapshots* of a view and its processor shall be shared (which is what applications usually do not want).

8.1.5 Channels

Channels are special actors that are used by processors or views to communicate with other actors (channel destinations). The following discusses channels in context of processors but this is also applicable to views.

Channels prevent redundant delivery of replayed messages to destinations during processor recovery. A replayed message is retained by a channel if its delivery has been confirmed by a destination.

```
import akka.actor.{ Actor, Props }
import akka.persistence.{ Channel, Deliver, Persistent, Processor }

class MyProcessor extends Processor {
```

```
  val destination = context.actorOf(Props[MyDestination])
  val channel = context.actorOf(Channel.props(), name = "myChannel")

  def receive = {
    case p @ Persistent(payload, _) =>
      channel ! Deliver(p.withPayload(s"processed ${payload}"), destination.path)
  }
}

class MyDestination extends Actor {
  def receive = {
    case p @ ConfirmablePersistent(payload, sequenceNr, redeliveries) =>
      // ...
      p.confirm()
  }
}
```

A channel is ready to use once it has been created, no recovery or further activation is needed. A `Deliver` request instructs a channel to send a `Persistent` message to a destination. A destination is provided as `ActorPath` and messages are sent by the channel via that path's `ActorSelection`. Sender references are preserved by a channel, therefore, a destination can reply to the sender of a `Deliver` request.

Note: Sending via a channel has at-least-once delivery semantics—by virtue of either the sending actor or the channel being persistent—which means that the semantics do not match those of a normal `ActorRef` send operation:

- it is not at-most-once delivery
- message order for the same sender–receiver pair is not retained due to possible resends
- after a crash and restart of the destination messages are still delivered—to the new actor incarnation

These semantics match precisely what an `ActorPath` represents (see *Actor Lifecycle*), therefore you need to supply a path and not a reference when constructing `Deliver` messages.

If a processor wants to reply to a `Persistent` message sender it should use the `sender` path as channel destination.

```
channel ! Deliver(p.withPayload(s"processed ${payload}"), sender.path)
```

Persistent messages delivered by a channel are of type `ConfirmablePersistent`. `ConfirmablePersistent` extends `Persistent` by adding the methods `confirm` and `redeliveries` (see also *Message re-delivery*). A channel destination confirms the delivery of a `ConfirmablePersistent` message by calling `confirm()` on that message. This asynchronously writes a confirmation entry to the journal. Replayed messages internally contain confirmation entries which allows a channel to decide if it should retain these messages or not.

A `Processor` can also be used as channel destination i.e. it can persist `ConfirmablePersistent` messages too.

Message re-delivery

Channels re-deliver messages to destinations if they do not confirm delivery within a configurable timeout. This timeout can be specified as `redeliverInterval` when creating a channel, optionally together with the maximum number of re-deliveries a channel should attempt for each unconfirmed message. The number of re-delivery attempts can be obtained via the `redeliveries` method on `ConfirmablePersistent` or by pattern matching.

```
context.actorOf(Channel.props(
  ChannelSettings(redeliverInterval = 30 seconds, redeliverMax = 15)),
  name = "myChannel")
```

A channel keeps messages in memory until their successful delivery has been confirmed or the maximum number of re-deliveries is reached. To be notified about messages that have reached the maximum number of re-deliveries, applications can register a listener at channel creation.

```
class MyListener extends Actor {
  def receive = {
    case RedeliverFailure(messages) => // ...
  }
}

val myListener = context.actorOf(Props[MyListener])
val myChannel = context.actorOf(Channel.props(
  ChannelSettings(redeliverFailureListener = Some(myListener))))
```

A listener receives `RedeliverFailure` notifications containing all messages that could not be delivered. On receiving a `RedeliverFailure` message, an application may decide to restart the sending processor to enforce a re-send of these messages to the channel or confirm these messages to prevent further re-sends. The sending processor can also be restarted any time later to re-send unconfirmed messages.

This combination of

- message persistence by sending processors
- message replays by sending processors
- message re-deliveries by channels and
- application-level confirmations (acknowledgements) by destinations

enables channels to provide at-least-once message delivery semantics. Possible duplicates can be detected by destinations by tracking message sequence numbers. Message sequence numbers are generated per sending processor. Depending on how a processor routes outbound messages to destinations, they may either see a contiguous message sequence or a sequence with gaps.

> **Warning:** If a processor emits more than one outbound message per inbound `Persistent` message it **must** use a separate channel for each outbound message to ensure that confirmations are uniquely identifiable, otherwise, at-least-once message delivery semantics do not apply. This rule has been introduced to avoid writing additional outbound message identifiers to the journal which would decrease the overall throughput. It is furthermore recommended to collapse multiple outbound messages to the same destination into a single outbound message, otherwise, if sent via multiple channels, their ordering is not defined.

If an application wants to have more control how sequence numbers are assigned to messages it should use an application-specific sequence number generator and include the generated sequence numbers into the `payload` of `Persistent` messages.

Persistent channels

Channels created with `Channel.props` do not persist messages. These channels are usually used in combination with a sending processor that takes care of persistence, hence, channel-specific persistence is not necessary in this case. They are referred to as transient channels in the following.

Persistent channels are like transient channels but additionally persist messages before delivering them. Messages that have been persisted by a persistent channel are deleted when destinations confirm their delivery. A persistent channel can be created with `PersistentChannel.props` and configured with a `PersistentChannelSettings` object.

```
val channel = context.actorOf(PersistentChannel.props(
  PersistentChannelSettings(redeliverInterval = 30 seconds, redeliverMax = 15)),
  name = "myPersistentChannel")

channel ! Deliver(Persistent("example"), destination.path)
```

A persistent channel is useful for delivery of messages to slow destinations or destinations that are unavailable for a long time. It can constrain the number of pending confirmations based on the `pendingConfirmationsMax` and `pendingConfirmationsMin` parameters of `PersistentChannelSettings`.

```
PersistentChannelSettings(
  pendingConfirmationsMax = 10000,
  pendingConfirmationsMin = 2000)
```

It suspends delivery when the number of pending confirmations reaches `pendingConfirmationsMax` and resumes delivery again when this number falls below `pendingConfirmationsMin`. This prevents both, flooding destinations with more messages than they can process and unlimited memory consumption by the channel. A persistent channel continues to persist new messages even when message delivery is temporarily suspended.

Standalone usage

Applications may also use channels standalone. Transient channels can be used standalone if re-delivery attempts to destinations are required but message loss in case of a sender JVM crash is not an issue. If message loss in case of a sender JVM crash is an issue, persistent channels should be used. In this case, applications may want to receive replies from the channel whether messages have been successfully persisted or not. This can be enabled by creating the channel with the `replyPersistent` configuration parameter set to `true`:

```
PersistentChannelSettings(replyPersistent = true)
```

With this setting, either the successfully persisted message is replied to the sender or a `PersistenceFailure` message. In case the latter case, the sender should re-send the message.

Identifiers

In the same way as *Processors* and *Views*, channels also have an identifier that defaults to a channel's path. A channel identifier can therefore be customized by using a custom actor name at channel creation. This changes that channel's name in its actor hierarchy and hence influences only part of the channel identifier. To fully customize a channel identifier, it should be provided as argument `Channel.props(String)` or `PersistentChannel.props(String)` (recommended to generate stable identifiers).

```
context.actorOf(Channel.props("my-stable-channel-id"))
```

8.1.6 Persistent messages

Payload

The payload of a `Persistent` message can be obtained via its

```
def payload: Any
```

method or by pattern matching

```
case Persistent(payload, _) =>
```

Inside processors, new persistent messages are derived from the current persistent message before sending them via a channel, either by calling `p.withPayload(...)` or `Persistent(...)` where the latter uses the implicit `currentPersistentMessage` made available by `Processor`.

```
implicit def currentPersistentMessage: Option[Persistent]
```

This is necessary for delivery confirmations to work properly. Both ways are equivalent but we recommend using `p.withPayload(...)` for clarity.

Sequence number

The sequence number of a `Persistent` message can be obtained via its

```
def sequenceNr: Long
```

method or by pattern matching

```
case Persistent(_, sequenceNr) =>
```

Persistent messages are assigned sequence numbers on a per-processor basis (or per channel basis if used standalone). A sequence starts at `1L` and doesn't contain gaps unless a processor deletes messages.

8.1.7 Snapshots

Snapshots can dramatically reduce recovery times of processors and views. The following discusses snapshots in context of processors but this is also applicable to views.

Processors can save snapshots of internal state by calling the `saveSnapshot` method. If saving of a snapshot succeeds, the processor receives a `SaveSnapshotSuccess` message, otherwise a `SaveSnapshotFailure` message

```
class MyProcessor extends Processor {
  var state: Any = _

  def receive = {
    case "snap"                                => saveSnapshot(state)
    case SaveSnapshotSuccess(metadata)         => // ...
    case SaveSnapshotFailure(metadata, reason) => // ...
  }
}
```

where `metadata` is of type `SnapshotMetadata`:

```
case class SnapshotMetadata(processorId: String, sequenceNr: Long, timestamp: Long = 0L)
```

During recovery, the processor is offered a previously saved snapshot via a `SnapshotOffer` message from which it can initialize internal state.

```
class MyProcessor extends Processor {
  var state: Any = _

  def receive = {
    case SnapshotOffer(metadata, offeredSnapshot) => state = offeredSnapshot
    case Persistent(payload, sequenceNr)          => // ...
  }
}
```

The replayed messages that follow the `SnapshotOffer` message, if any, are younger than the offered snapshot. They finally recover the processor to its current (i.e. latest) state.

In general, a processor is only offered a snapshot if that processor has previously saved one or more snapshots and at least one of these snapshots matches the `SnapshotSelectionCriteria` that can be specified for recovery.

```
processor ! Recover(fromSnapshot = SnapshotSelectionCriteria(
  maxSequenceNr = 457L,
  maxTimestamp = System.currentTimeMillis))
```

If not specified, they default to `SnapshotSelectionCriteria.Latest` which selects the latest (= youngest) snapshot. To disable snapshot-based recovery, applications should use `SnapshotSelectionCriteria.None`. A recovery where no saved snapshot matches the specified `SnapshotSelectionCriteria` will replay all journaled messages.

Snapshot deletion

A processor can delete individual snapshots by calling the `deleteSnapshot` method with the sequence number and the timestamp of a snapshot as argument. To bulk-delete snapshots matching `SnapshotSelectionCriteria`, processors should use the `deleteSnapshots` method.

8.1.8 Event sourcing

In all the examples so far, messages that change a processor's state have been sent as `Persistent` messages by an application, so that they can be replayed during recovery. From this point of view, the journal acts as a write-ahead-log for whatever `Persistent` messages a processor receives. This is also known as *command sourcing*. Commands, however, may fail and some applications cannot tolerate command failures during recovery.

For these applications Event Sourcing is a better choice. Applied to Akka persistence, the basic idea behind event sourcing is quite simple. A processor receives a (non-persistent) command which is first validated if it can be applied to the current state. Here, validation can mean anything, from simple inspection of a command message's fields up to a conversation with several external services, for example. If validation succeeds, events are generated from the command, representing the effect of the command. These events are then persisted and, after successful persistence, used to change a processor's state. When the processor needs to be recovered, only the persisted events are replayed of which we know that they can be successfully applied. In other words, events cannot fail when being replayed to a processor, in contrast to commands. Eventsourced processors may of course also process commands that do not change application state, such as query commands, for example.

Akka persistence supports event sourcing with the `EventsourcedProcessor` trait (which implements event sourcing as a pattern on top of command sourcing). A processor that extends this trait does not handle `Persistent` messages directly but uses the `persist` method to persist and handle events. The behavior of an `EventsourcedProcessor` is defined by implementing `receiveRecover` and `receiveCommand`. This is demonstrated in the following example.

```
import akka.actor._
import akka.persistence._

case class Cmd(data: String)
case class Evt(data: String)

case class ExampleState(events: List[String] = Nil) {
  def update(evt: Evt) = copy(evt.data :: events)
  def size = events.length
  override def toString: String = events.reverse.toString
}

class ExampleProcessor extends EventsourcedProcessor {
  var state = ExampleState()

  def updateState(event: Evt): Unit =
    state = state.update(event)

  def numEvents =
    state.size

  val receiveRecover: Receive = {
    case evt: Evt                                 => updateState(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  }

  val receiveCommand: Receive = {
    case Cmd(data) =>
      persist(Evt(s"${data}-${numEvents}"))(updateState)
      persist(Evt(s"${data}-${numEvents + 1}")) { event =>
        updateState(event)
        context.system.eventStream.publish(event)
```

```
      }
    case "snap"  => saveSnapshot(state)
    case "print" => println(state)
  }

}
```

The example defines two data types, `Cmd` and `Evt` to represent commands and events, respectively. The `state` of the `ExampleProcessor` is a list of persisted event data contained in `ExampleState`.

The processor's `receiveRecover` method defines how `state` is updated during recovery by handling `Evt` and `SnapshotOffer` messages. The processor's `receiveCommand` method is a command handler. In this example, a command is handled by generating two events which are then persisted and handled. Events are persisted by calling `persist` with an event (or a sequence of events) as first argument and an event handler as second argument.

The `persist` method persists events asynchronously and the event handler is executed for successfully persisted events. Successfully persisted events are internally sent back to the processor as individual messages that trigger event handler executions. An event handler may close over processor state and mutate it. The sender of a persisted event is the sender of the corresponding command. This allows event handlers to reply to the sender of a command (not shown).

The main responsibility of an event handler is changing processor state using event data and notifying others about successful state changes by publishing events.

When persisting events with `persist` it is guaranteed that the processor will not receive further commands between the `persist` call and the execution(s) of the associated event handler. This also holds for multiple `persist` calls in context of a single command.

The easiest way to run this example yourself is to download Typesafe Activator and open the tutorial named Akka Persistence Samples with Scala. It contains instructions on how to run the `EventsourcedExample`.

Note: It's also possible to switch between different command handlers during normal processing and recovery with `context.become()` and `context.unbecome()`. To get the actor into the same state after recovery you need to take special care to perform the same state transitions with `become` and `unbecome` in the `receiveRecover` method as you would have done in the command handler.

Reliable event delivery

Sending events from an event handler to another actor has at-most-once delivery semantics. For at-least-once delivery, *Channels* must be used. In this case, also replayed events (received by `receiveRecover`) must be sent to a channel, as shown in the following example:

```
class MyEventsourcedProcessor(destination: ActorRef) extends EventsourcedProcessor {
  val channel = context.actorOf(Channel.props("channel"))

  def handleEvent(event: String) = {
    // update state
    // ...
    // reliably deliver events
    channel ! Deliver(Persistent(event), destination.path)
  }

  def receiveRecover: Receive = {
    case event: String => handleEvent(event)
  }

  def receiveCommand: Receive = {
    case "cmd" => {
      // ...
      persist("evt")(handleEvent)
```

```
    }
  }
}
```

In larger integration scenarios, channel destinations may be actors that submit received events to an external message broker, for example. After having successfully submitted an event, they should call `confirm()` on the received `ConfirmablePersistent` message.

8.1.9 Batch writes

To optimize throughput, a `Processor` internally batches received `Persistent` messages under high load before writing them to the journal (as a single batch). The batch size dynamically grows from 1 under low and moderate loads to a configurable maximum size (default is `200`) under high load.

```
akka.persistence.journal.max-message-batch-size = 200
```

A new batch write is triggered by a processor as soon as a batch reaches the maximum size or if the journal completed writing the previous batch. Batch writes are never timer-based which keeps latencies at a minimum.

Applications that want to have more explicit control over batch writes and batch sizes can send processors `PersistentBatch` messages.

```
class MyProcessor extends Processor {
  def receive = {
    case Persistent("a", _) => // ...
    case Persistent("b", _) => // ...
  }
}

val system = ActorSystem("example")
val processor = system.actorOf(Props[MyProcessor])

processor ! PersistentBatch(List(Persistent("a"), Persistent("b")))
```

`Persistent` messages contained in a `PersistentBatch` are always written atomically, even if the batch size is greater than `max-message-batch-size`. Also, a `PersistentBatch` is written isolated from other batches. `Persistent` messages contained in a `PersistentBatch` are received individually by a processor.

`PersistentBatch` messages, for example, are used internally by an `EventsourcedProcessor` to ensure atomic writes of events. All events that are persisted in context of a single command are written as a single batch to the journal (even if `persist` is called multiple times per command). The recovery of an `EventsourcedProcessor` will therefore never be done partially (with only a subset of events persisted by a single command).

Confirmation and deletion operations performed by *Channels* are also batched. The maximum confirmation and deletion batch sizes are configurable with `akka.persistence.journal.max-confirmation-batch-size` and `akka.persistence.journal.max-deletion-batch-size`, respectively.

8.1.10 Storage plugins

Storage backends for journals and snapshot stores are pluggable in Akka persistence. The default journal plugin writes messages to LevelDB (see *Local LevelDB journal*). The default snapshot store plugin writes snapshots as individual files to the local filesystem (see *Local snapshot store*). Applications can provide their own plugins by implementing a plugin API and activate them by configuration. Plugin development requires the following imports:

```
import scala.concurrent.Future
import scala.collection.immutable.Seq
import akka.persistence._
```

```
import akka.persistence.journal._
import akka.persistence.snapshot._
```

Journal plugin API

A journal plugin either extends `SyncWriteJournal` or `AsyncWriteJournal`. `SyncWriteJournal` is an actor that should be extended when the storage backend API only supports synchronous, blocking writes. In this case, the methods to be implemented are:

```
/**
 * Plugin API: synchronously writes a batch of persistent messages to the journal.
 * The batch write must be atomic i.e. either all persistent messages in the batch
 * are written or none.
 */
def writeMessages(messages: immutable.Seq[PersistentRepr]): Unit

/**
 * Plugin API: synchronously writes a batch of delivery confirmations to the journal.
 */
def writeConfirmations(confirmations: immutable.Seq[PersistentConfirmation]): Unit

/**
 * Plugin API: synchronously deletes messages identified by `messageIds` from the
 * journal. If `permanent` is set to `false`, the persistent messages are marked as
 * deleted, otherwise they are permanently deleted.
 */
def deleteMessages(messageIds: immutable.Seq[PersistentId], permanent: Boolean): Unit

/**
 * Plugin API: synchronously deletes all persistent messages up to `toSequenceNr`
 * (inclusive). If `permanent` is set to `false`, the persistent messages are marked
 * as deleted, otherwise they are permanently deleted.
 */
def deleteMessagesTo(processorId: String, toSequenceNr: Long, permanent: Boolean): Unit
```

`AsyncWriteJournal` is an actor that should be extended if the storage backend API supports asynchronous, non-blocking writes. In this case, the methods to be implemented are:

```
/**
 * Plugin API: asynchronously writes a batch of persistent messages to the journal.
 * The batch write must be atomic i.e. either all persistent messages in the batch
 * are written or none.
 */
def asyncWriteMessages(messages: immutable.Seq[PersistentRepr]): Future[Unit]

/**
 * Plugin API: asynchronously writes a batch of delivery confirmations to the journal.
 */
def asyncWriteConfirmations(confirmations: immutable.Seq[PersistentConfirmation]): Future[Unit]

/**
 * Plugin API: asynchronously deletes messages identified by `messageIds` from the
 * journal. If `permanent` is set to `false`, the persistent messages are marked as
 * deleted, otherwise they are permanently deleted.
 */
def asyncDeleteMessages(messageIds: immutable.Seq[PersistentId], permanent: Boolean): Future[Unit

/**
 * Plugin API: asynchronously deletes all persistent messages up to `toSequenceNr`
 * (inclusive). If `permanent` is set to `false`, the persistent messages are marked
 * as deleted, otherwise they are permanently deleted.
```

```
 */
def asyncDeleteMessagesTo(processorId: String, toSequenceNr: Long, permanent: Boolean): Future[Un
```

Message replays and sequence number recovery are always asynchronous, therefore, any journal plugin must implement:

```
/**
 * Plugin API: asynchronously replays persistent messages. Implementations replay
 * a message by calling `replayCallback`. The returned future must be completed
 * when all messages (matching the sequence number bounds) have been replayed.
 * The future must be completed with a failure if any of the persistent messages
 * could not be replayed.
 *
 * The `replayCallback` must also be called with messages that have been marked
 * as deleted. In this case a replayed message's `deleted` method must return
 * `true`.
 *
 * The channel ids of delivery confirmations that are available for a replayed
 * message must be contained in that message's `confirms` sequence.
 *
 * @param processorId processor id.
 * @param fromSequenceNr sequence number where replay should start (inclusive).
 * @param toSequenceNr sequence number where replay should end (inclusive).
 * @param max maximum number of messages to be replayed.
 * @param replayCallback called to replay a single message. Can be called from any
 *                       thread.
 *
 * @see [[AsyncWriteJournal]]
 * @see [[SyncWriteJournal]]
 */
def asyncReplayMessages(processorId: String, fromSequenceNr: Long, toSequenceNr: Long, max: Long)

/**
 * Plugin API: asynchronously reads the highest stored sequence number for the
 * given `processorId`.
 *
 * @param processorId processor id.
 * @param fromSequenceNr hint where to start searching for the highest sequence
 *                       number.
 */
def asyncReadHighestSequenceNr(processorId: String, fromSequenceNr: Long): Future[Long]
```

A journal plugin can be activated with the following minimal configuration:

```
# Path to the journal plugin to be used
akka.persistence.journal.plugin = "my-journal"

# My custom journal plugin
my-journal {
  # Class name of the plugin.
  class = "docs.persistence.MyJournal"
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.actor.default-dispatcher"
}
```

The specified plugin `class` must have a no-arg constructor. The `plugin-dispatcher` is the dispatcher used for the plugin actor. If not specified, it defaults to `akka.persistence.dispatchers.default-plugin-dispatcher` for `SyncWriteJournal` plugins and `akka.actor.default-dispatcher` for `AsyncWriteJournal` plugins.

Snapshot store plugin API

A snapshot store plugin must extend the `SnapshotStore` actor and implement the following methods:

```
/**
 * Plugin API: asynchronously loads a snapshot.
 *
 * @param processorId processor id.
 * @param criteria selection criteria for loading.
 */
def loadAsync(processorId: String, criteria: SnapshotSelectionCriteria): Future[Option[SelectedSna

/**
 * Plugin API: asynchronously saves a snapshot.
 *
 * @param metadata snapshot metadata.
 * @param snapshot snapshot.
 */
def saveAsync(metadata: SnapshotMetadata, snapshot: Any): Future[Unit]

/**
 * Plugin API: called after successful saving of a snapshot.
 *
 * @param metadata snapshot metadata.
 */
def saved(metadata: SnapshotMetadata)

/**
 * Plugin API: deletes the snapshot identified by `metadata`.
 *
 * @param metadata snapshot metadata.
 */

def delete(metadata: SnapshotMetadata)

/**
 * Plugin API: deletes all snapshots matching `criteria`.
 *
 * @param processorId processor id.
 * @param criteria selection criteria for deleting.
 */
def delete(processorId: String, criteria: SnapshotSelectionCriteria)
```

A snapshot store plugin can be activated with the following minimal configuration:

```
# Path to the snapshot store plugin to be used
akka.persistence.snapshot-store.plugin = "my-snapshot-store"

# My custom snapshot store plugin
my-snapshot-store {
  # Class name of the plugin.
  class = "docs.persistence.MySnapshotStore"
  # Dispatcher for the plugin actor.
  plugin-dispatcher = "akka.persistence.dispatchers.default-plugin-dispatcher"
}
```

The specified plugin `class` must have a no-arg constructor. The `plugin-dispatcher` is the dispatcher used for the plugin actor. If not specified, it defaults to `akka.persistence.dispatchers.default-plugin-dispatcher`.

8.1.11 Pre-packaged plugins

Local LevelDB journal

The default journal plugin is `akka.persistence.journal.leveldb` which writes messages to a local LevelDB instance. The default location of the LevelDB files is a directory named `journal` in the current working directory. This location can be changed by configuration where the specified path can be relative or absolute:

```
akka.persistence.journal.leveldb.dir = "target/journal"
```

With this plugin, each actor system runs its own private LevelDB instance.

Shared LevelDB journal

A LevelDB instance can also be shared by multiple actor systems (on the same or on different nodes). This, for example, allows processors to failover to a backup node and continue using the shared journal instance from the backup node.

> **Warning:** A shared LevelDB instance is a single point of failure and should therefore only be used for testing purposes. Highly-available, replicated journal are available as Community plugins.

A shared LevelDB instance is started by instantiating the `SharedLeveldbStore` actor.

```
import akka.persistence.journal.leveldb.SharedLeveldbStore

val store = system.actorOf(Props[SharedLeveldbStore], "store")
```

By default, the shared instance writes journaled messages to a local directory named `journal` in the current working directory. The storage location can be changed by configuration:

```
akka.persistence.journal.leveldb-shared.store.dir = "target/shared"
```

Actor systems that use a shared LevelDB store must activate the `akka.persistence.journal.leveldb-shared` plugin.

```
akka.persistence.journal.plugin = "akka.persistence.journal.leveldb-shared"
```

This plugin must be initialized by injecting the (remote) `SharedLeveldbStore` actor reference. Injection is done by calling the `SharedLeveldbJournal.setStore` method with the actor reference as argument.

```
trait SharedStoreUsage extends Actor {
  override def preStart(): Unit = {
    context.actorSelection("akka.tcp://example@127.0.0.1:2552/user/store") ! Identify(1)
  }

  def receive = {
    case ActorIdentity(1, Some(store)) =>
      SharedLeveldbJournal.setStore(store, context.system)
  }
}
```

Internal journal commands (sent by processors) are buffered until injection completes. Injection is idempotent i.e. only the first injection is used.

Local snapshot store

The default snapshot store plugin is `akka.persistence.snapshot-store.local`. It writes snapshot files to the local filesystem. The default storage location is a directory named `snapshots` in the current working directory. This can be changed by configuration where the specified path can be relative or absolute:

```
akka.persistence.snapshot-store.local.dir = "target/snapshots"
```

8.1.12 Custom serialization

Serialization of snapshots and payloads of `Persistent` messages is configurable with Akka's *Serialization* infrastructure. For example, if an application wants to serialize

- payloads of type `MyPayload` with a custom `MyPayloadSerializer` and
- snapshots of type `MySnapshot` with a custom `MySnapshotSerializer`

it must add

```
akka.actor {
  serializers {
    my-payload = "docs.persistence.MyPayloadSerializer"
    my-snapshot = "docs.persistence.MySnapshotSerializer"
  }
  serialization-bindings {
    "docs.persistence.MyPayload" = my-payload
    "docs.persistence.MySnapshot" = my-snapshot
  }
}
```

to the application configuration. If not specified, a default serializer is used.

8.1.13 Testing

When running tests with LevelDB default settings in `sbt`, make sure to set `fork := true` in your sbt project otherwise, you'll see an `UnsatisfiedLinkError`. Alternatively, you can switch to a LevelDB Java port by setting

```
akka.persistence.journal.leveldb.native = off
```

or

```
akka.persistence.journal.leveldb-shared.store.native = off
```

in your Akka configuration. The LevelDB Java port is for testing purposes only.

8.1.14 Miscellaneous

State machines

State machines can be persisted by mixing in the `FSM` trait into processors.

```
import akka.actor.FSM
import akka.persistence.{ Processor, Persistent }

class PersistentDoor extends Processor with FSM[String, Int] {
  startWith("closed", 0)

  when("closed") {
    case Event(Persistent("open", _), counter) =>
      goto("open") using (counter + 1) replying (counter)
  }

  when("open") {
    case Event(Persistent("close", _), counter) =>
      goto("closed") using (counter + 1) replying (counter)
```

```
  }
}
```

8.1.15 Configuration

There are several configuration properties for the persistence module, please refer to the *reference configuration*.

8.2 Multi Node Testing

Note: This module is *experimental*. This document describes how to use the features implemented so far. More features are coming in Akka Coltrane. Track progress of the Coltrane milestone in Assembla.

8.2.1 Multi Node Testing Concepts

When we talk about multi node testing in Akka we mean the process of running coordinated tests on multiple actor systems in different JVMs. The multi node testing kit consist of three main parts.

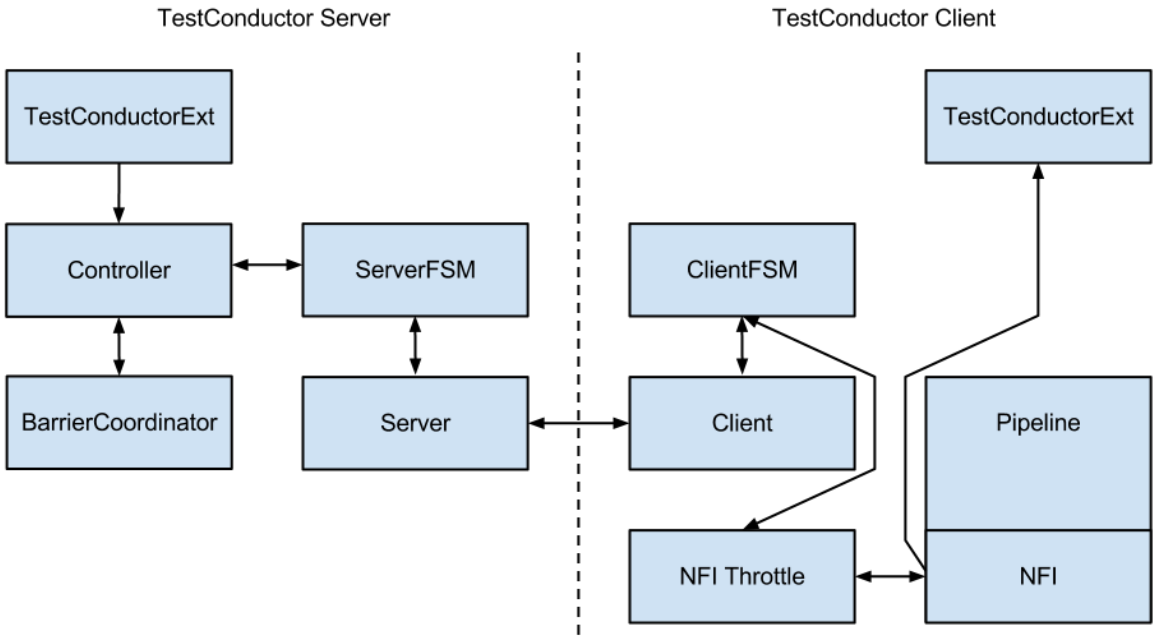
- The Test Conductor. that coordinates and controls the nodes under test.
- The Multi Node Spec. that is a convenience wrapper for starting the `TestConductor` and letting all nodes connect to it.
- The SbtMultiJvm Plugin. that starts tests in multiple JVMs possibly on multiple machines.

8.2.2 The Test Conductor

The basis for the multi node testing is the `TestConductor`. It is an Akka Extension that plugs in to the network stack and it is used to coordinate the nodes participating in the test and provides several features including:

- Node Address Lookup: Finding out the full path to another test node (No need to share configuration between test nodes)
- Node Barrier Coordination: Waiting for other nodes at named barriers.
- Network Failure Injection: Throttling traffic, dropping packets, unplugging and plugging nodes back in.

This is a schematic overview of the test conductor.



The test conductor server is responsible for coordinating barriers and sending commands to the test conductor clients that act upon them, e.g. throttling network traffic to/from another client. More information on the possible operations is availible in the `akka.remote.testconductor.Conductor` API documentation.

8.2.3 The Multi Node Spec

The Multi Node Spec consists of two parts. The `MultiNodeConfig` that is responsible for common configuration and enumerating and naming the nodes under test. The `MultiNodeSpec` that contains a number of convenience functions for making the test nodes interact with each other. More information on the possible operations is available in the `akka.remote.testkit.MultiNodeSpec` API documentation.

The setup of the `MultiNodeSpec` is configured through java system properties that you set on all JVMs that's going to run a node under test. These can easily be set on the JVM command line with `-Dproperty=value`.

These are the available properties:

- `multinode.max-nodes`

  The maximum number of nodes that a test can have.

- `multinode.host`

  The host name or IP for this node. Must be resolvable using InetAddress.getByName.

- `multinode.port`

  The port number for this node. Defaults to 0 which will use a random port.

- `multinode.server-host`

  The host name or IP for the server node. Must be resolvable using InetAddress.getByName.

- `multinode.server-port`

  The port number for the server node. Defaults to 4711.

- `multinode.index`

  The index of this node in the sequence of roles defined for the test. The index 0 is special and that machine will be the server. All failure injection and throttling must be done from this node.

8.2.4 The SbtMultiJvm Plugin

The *SbtMultiJvm Plugin* has been updated to be able to run multi node tests, by automatically generating the relevant `multinode.*` properties. This means that you can easily run multi node tests on a single machine without any special configuration by just running them as normal multi-jvm tests. These tests can then be run distributed over multiple machines without any changes simply by using the multi-node additions to the plugin.

Multi Node Specific Additions

The plugin also has a number of new `multi-node-*` sbt tasks and settings to support running tests on multiple machines. The necessary test classes and dependencies are packaged for distribution to other machines with SbtAssembly into a jar file with a name on the format `<projectName>_<scalaVersion>-<projectVersion>-multi-jvm-assembly.jar`

Note: To be able to distribute and kick off the tests on multiple machines, it is assumed that both host and target systems are POSIX like systems with `ssh` and `rsync` available.

These are the available sbt multi-node settings:

- `multiNodeHosts`

  A sequence of hosts to use for running the test, on the form `user@host:java` where host is the only required part. Will override settings from file.

- `multiNodeHostsFileName`

  A file to use for reading in the hosts to use for running the test. One per line on the same format as above. Defaults to `multi-node-test.hosts` in the base project directory.

- `multiNodeTargetDirName`

  A name for the directory on the target machine, where to copy the jar file. Defaults to `multi-node-test` in the base directory of the ssh user used to rsync the jar file.

- `multiNodeJavaName`

  The name of the default Java executable on the target machines. Defaults to `java`.

Here are some examples of how you define hosts:

- `localhost`

  The current user on localhost using the default java.

- `user1@host1`

  User `user1` on host `host1` with the default java.

- `user2@host2:/usr/lib/jvm/java-7-openjdk-amd64/bin/java`

  User `user2` on host `host2` using java 7.

- `host3:/usr/lib/jvm/java-6-openjdk-amd64/bin/java`

  The current user on host `host3` using java 6.

Running the Multi Node Tests

To run all the multi node test in multi-node mode (i.e. distributing the jar files and kicking off the tests remotely) from inside sbt, use the `multi-node-test` task:

```
multi-node-test
```

To run all of them in multi-jvm mode (i.e. all JVMs on the local machine) do:

```
multi-jvm:test
```

To run individual tests use the `multi-node-test-only` task:

```
multi-node-test-only your.MultiNodeTest
```

To run individual tests in the multi-jvm mode do:

```
multi-jvm:test-only your.MultiNodeTest
```

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.

8.2.5 Preparing Your Project for Multi Node Testing

The multi node testing kit is a separate jar file. Make sure that you have the following dependency in your project:

```
"com.typesafe.akka" %% "akka-multi-node-testkit" % "2.3.2"
```

If you are using the latest nightly build you should pick a timestamped Akka version from http://repo.typesafe.com/typesafe/snapshots/com/typesafe/akka/akka-multi-node-testkit_2.10/. We recommend against using `SNAPSHOT` in order to obtain stable builds.

8.2.6 A Multi Node Testing Example

First we need some scaffolding to hook up the `MultiNodeSpec` with your favorite test framework. Lets define a trait `STMultiNodeSpec` that uses ScalaTest to start and stop `MultiNodeSpec`.

```
package sample.multinode

import org.scalatest.{ BeforeAndAfterAll, WordSpecLike }
import org.scalatest.Matchers
import akka.remote.testkit.MultiNodeSpecCallbacks

/**
 * Hooks up MultiNodeSpec with ScalaTest
 */
trait STMultiNodeSpec extends MultiNodeSpecCallbacks
  with WordSpecLike with Matchers with BeforeAndAfterAll {

  override def beforeAll() = multiNodeSpecBeforeAll()

  override def afterAll() = multiNodeSpecAfterAll()
}
```

Then we need to define a configuration. Lets use two nodes `"node1` and `"node2"` and call it `MultiNodeSampleConfig`.

```
package sample.multinode
import akka.remote.testkit.MultiNodeConfig

object MultiNodeSampleConfig extends MultiNodeConfig {
  val node1 = role("node1")
  val node2 = role("node2")
}
```

And then finally to the node test code. That starts the two nodes, and demonstrates a barrier, and a remote actor message send/receive.

```
package sample.multinode
import akka.remote.testkit.MultiNodeSpec
```

```
import akka.testkit.ImplicitSender
import akka.actor.{ Props, Actor }

class MultiNodeSampleSpecMultiJvmNode1 extends MultiNodeSample
class MultiNodeSampleSpecMultiJvmNode2 extends MultiNodeSample

object MultiNodeSample {
  class Ponger extends Actor {
    def receive = {
      case "ping" => sender() ! "pong"
    }
  }
}

class MultiNodeSample extends MultiNodeSpec(MultiNodeSampleConfig)
  with STMultiNodeSpec with ImplicitSender {

  import MultiNodeSampleConfig._
  import MultiNodeSample._

  def initialParticipants = roles.size

  "A MultiNodeSample" must {

    "wait for all nodes to enter a barrier" in {
      enterBarrier("startup")
    }

    "send to and receive from a remote node" in {
      runOn(node1) {
        enterBarrier("deployed")
        val ponger = system.actorSelection(node(node2) / "user" / "ponger")
        ponger ! "ping"
        expectMsg("pong")
      }

      runOn(node2) {
        system.actorOf(Props[Ponger], "ponger")
        enterBarrier("deployed")
      }

      enterBarrier("finished")
    }
  }
}
```

The easiest way to run this example yourself is to download Typesafe Activator and open the tutorial named Akka Multi-Node Testing Sample with Scala.

8.2.7 Things to Keep in Mind

There are a couple of things to keep in mind when writing multi node tests or else your tests might behave in surprising ways.

- Don't issue a shutdown of the first node. The first node is the controller and if it shuts down your test will break.
- To be able to use `blackhole`, `passThrough`, and `throttle` you must activate the failure injector and throttler transport adapters by specifying `testTransport(on = true)` in your MultiNodeConfig.
- Throttling, shutdown and other failure injections can only be done from the first node, which again is the controller.

- Don't ask for the address of a node using `node(address)` after the node has been shut down. Grab the address before shutting down the node.
- Don't use MultiNodeSpec methods like address lookup, barrier entry et.c. from other threads than the main test thread. This also means that you shouldn't use them from inside an actor, a future, or a scheduled task.

8.2.8 Configuration

There are several configuration properties for the Multi-Node Testing module, please refer to the *reference configuration*.

8.3 Actors (Java with Lambda Support)

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.

Warning: The Java with lambda support part of Akka is marked as **"experimental"** as of its introduction in Akka 2.3.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum, but the binary compatibility guarantee for maintenance releases does not apply to the `akka.actor.AbstractActor`, related classes and the `akka.japi.pf` package.

8.3.1 Creating Actors

Note: 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 *Actor References, Paths and Addresses*.

Defining an Actor class

Actor classes are implemented by extending the `AbstractActor` class and setting the "initial behavior" in the constructor by calling the `receive` method in the `AbstractActor`.

The argument to the `receive` method is a `PartialFunction<Object,BoxedUnit>` that defines which messages your Actor can handle, along with the implementation of how the messages should be processed.

Don't let the type signature scare you. To allow you to easily build up a partial function there is a builder named `ReceiveBuilder` that you can use.

Here is an example:

```
import akka.actor.AbstractActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.pf.ReceiveBuilder;

public class MyActor extends AbstractActor {
  private final LoggingAdapter log = Logging.getLogger(context().system(), this);

  public MyActor() {
```

```
    receive(ReceiveBuilder.
      match(String.class, s -> {
        log.info("Received String message: {}", s);
      }).
      matchAny(o -> log.info("received unknown message")).build()
    );
  }
}
```

Please note that the Akka Actor `receive` message loop is exhaustive, which is different compared to Erlang and the late 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`.

Note further that the return type of the behavior defined above is `Unit`; if the actor shall reply to the received message then this must be done explicitly as explained below.

The argument to the `receive` method is a partial function object, which is stored within the actor as its "initial behavior", see Become/Unbecome for further information on changing the behavior of an actor after its construction.

Props

`Props` is a configuration class to specify options for the creation of actors, think of it as an immutable and thus freely shareable recipe for creating an actor including associated deployment information (e.g. which dispatcher to use, see more below). Here are some examples of how to create a `Props` instance.

```
import akka.actor.Props;
```

```
Props props1 = Props.create(MyActor.class);
Props props2 = Props.create(ActorWithArgs.class,
  () -> new ActorWithArgs("arg")); // careful, see below
Props props3 = Props.create(ActorWithArgs.class, "arg");
```

The second variant shows how to pass constructor arguments to the `Actor` being created, but it should only be used outside of actors as explained below.

The last line shows a possibility to pass constructor arguments regardless of the context it is being used in. The presence of a matching constructor is verified during construction of the `Props` object, resulting in an `IllegalArgumentException` if no or multiple matching constructors are found.

Dangerous Variants

```
// NOT RECOMMENDED within another actor:
// encourages to close over enclosing class
Props props7 = Props.create(ActorWithArgs.class,
  () -> new ActorWithArgs("arg"));
```

This method is not recommended to be used within another actor because it encourages to close over the enclosing scope, resulting in non-serializable `Props` and possibly race conditions (breaking the actor encapsulation). On the other hand using this variant in a `Props` factory in the actor's companion object as documented under "Recommended Practices" below is completely fine.

There were two use-cases for these methods: passing constructor arguments to the actor—which is solved by the newly introduced `Props.create(clazz, args)` method above or the recommended practice below—and creating actors "on the spot" as anonymous classes. The latter should be solved by making these actors named classes instead (if they are not declared within a top-level `object` then the enclosing instance's `this` reference needs to be passed as the first argument).

Warning: Declaring one actor within another is very dangerous and breaks actor encapsulation. Never pass an actor's `this` reference into `Props`!

Recommended Practices

It is a good idea to provide factory methods on the companion object of each `Actor` which help keeping the creation of suitable `Props` as close to the actor definition as possible. This also avoids the pitfalls associated with using the `Props.create(...)` method which takes a by-name argument, since within a companion object the given code block will not retain a reference to its enclosing scope:

```
public class DemoActor extends AbstractActor {
  /**
   * Create Props for an actor of this type.
   * @param magicNumber The magic number to be passed to this actor's constructor.
   * @return a Props for creating this actor, which can then be further configured
   *         (e.g. calling `.withDispatcher()` on it)
   */
  static Props props(Integer magicNumber) {
    // You need to specify the actual type of the returned actor
    // since Java 8 lambdas have some runtime type information erased
    return Props.create(DemoActor.class, () -> new DemoActor(magicNumber));
  }

  private final Integer magicNumber;

  DemoActor(Integer magicNumber) {
    this.magicNumber = magicNumber;
    receive(ReceiveBuilder.
      match(Integer.class, i -> {
        sender().tell(i + magicNumber, self());
      }).build()
    );
  }
}

public class SomeOtherActor extends AbstractActor {
  // Props(new DemoActor(42)) would not be safe
  ActorRef demoActor = context().actorOf(DemoActor.props(42), "demo");
  // ...
}
```

Creating Actors with Props

Actors are created by passing a `Props` instance into the `actorOf` factory method which is available on `ActorSystem` and `ActorContext`.

```
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
```

```
// ActorSystem is a heavy object: create only one per application
final ActorSystem system = ActorSystem.create("MySystem", config);
final ActorRef myActor = system.actorOf(Props.create(MyActor.class), "myactor");
```

Using the `ActorSystem` will create top-level actors, supervised by the actor system's provided guardian actor, while using an actor's context will create a child actor.

```
public class FirstActor extends AbstractActor {
  final ActorRef child = context().actorOf(Props.create(MyActor.class), "myChild");
  // plus some behavior ...
}
```

It is recommended to create a hierarchy of children, grand-children and so on such that it fits the logical failure-handling structure of the application, see *Actor Systems*.

The call to `actorOf` returns an instance of `ActorRef`. This is a handle to the actor instance and the only way to interact with it. 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.

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 `$`, but it may contain URL encoded characters (eg. `%20` for a blank space). If the given name is already in use by another child to the same parent an *InvalidActorNameException* is thrown.

Actors are automatically started asynchronously when created.

Dependency Injection

If your UntypedActor has a constructor that takes parameters then those need to be part of the `Props` as well, as described above. But there are cases when a factory method must be used, for example when the actual constructor arguments are determined by a dependency injection framework.

```
import akka.actor.Actor;
import akka.actor.IndirectActorProducer;
```

```
class DependencyInjector implements IndirectActorProducer {
  final Object applicationContext;
  final String beanName;

  public DependencyInjector(Object applicationContext, String beanName) {
    this.applicationContext = applicationContext;
    this.beanName = beanName;
  }

  @Override
  public Class<? extends Actor> actorClass() {
    return MyActor.class;
  }

  @Override
  public MyActor produce() {
    MyActor result;
    // obtain fresh Actor instance from DI framework ...
    return result;
  }
}

  final ActorRef myActor = getContext().actorOf(
    Props.create(DependencyInjector.class, applicationContext, "MyActor"),
      "myactor3");
```

Warning: You might be tempted at times to offer an `IndirectActorProducer` 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*.
When using a dependency injection framework, actor beans *MUST NOT* have singleton scope.

Techniques for dependency injection and integration with dependency injection frameworks are described in more depth in the Using Akka with Dependency Injection guideline and the Akka Java Spring tutorial in Typesafe Activator.

The Inbox

When writing code outside of actors which shall communicate with actors, the `ask` pattern can be a solution (see below), but there are two thing it cannot do: receiving multiple replies (e.g. by subscribing an `ActorRef` to a notification service) and watching other actors' lifecycle. For these purposes there is the `Inbox` class:

```
final Inbox inbox = Inbox.create(system);
inbox.send(target, "hello");
assert inbox.receive(Duration.create(1, TimeUnit.SECONDS)).equals("world");
```

The `send` method wraps a normal `tell` and supplies the internal actor's reference as the sender. This allows the reply to be received on the last line. Watching an actor is quite simple as well:

```
final Inbox inbox = Inbox.create(system);
inbox.watch(target);
target.tell(PoisonPill.getInstance(), ActorRef.noSender());
assert inbox.receive(Duration.create(1, TimeUnit.SECONDS)) instanceof Terminated;
```

8.3.2 Actor API

The `AbstractActor` class defines a method called `receive`, that is used to set the "initial 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 (set configuration item `akka.actor.debug.unhandled` to `on` to have them converted into actual Debug messages).

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

  This strategy is typically declared inside the actor in order to have access to the actor's internal state within the decider function: since failure is communicated as a message sent to the supervisor and processed like other messages (albeit outside of the normal behavior), all values and variables within the actor are available, as is the `sender` reference (which will be the immediate child reporting the failure; if the original failure occurred within a distant descendant it is still reported one level up at a time).
- `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*

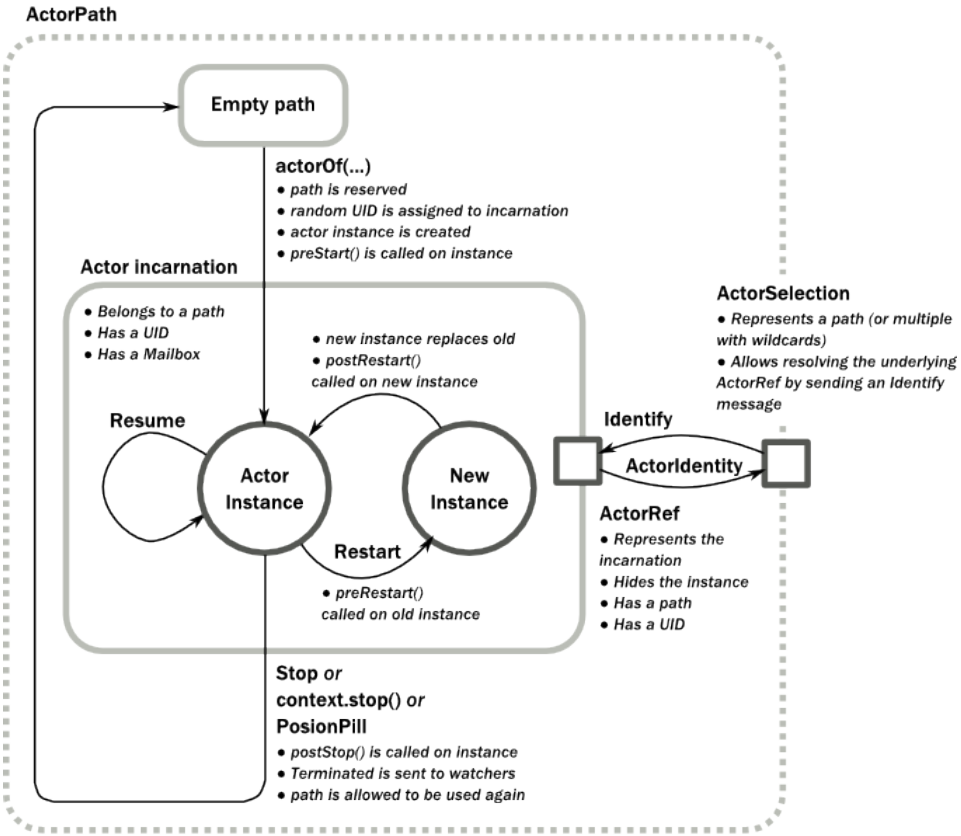
The remaining visible methods are user-overridable life-cycle hooks which are described in the following:

```
public void preStart() {
}

public void preRestart(Throwable reason, scala.Option<Object> message) {
  for (ActorRef each : getContext().getChildren()) {
    getContext().unwatch(each);
    getContext().stop(each);
```

```
  }
  postStop();
}

public void postRestart(Throwable reason) {
  preStart();
}

public void postStop() {
}
```

The implementations shown above are the defaults provided by the `AbstractActor` class.

Actor Lifecycle



A path in an actor system represents a "place" which might be occupied by a living actor. Initially (apart from system initialized actors) a path is empty. When `actorOf()` is called it assigns an *incarnation* of the actor described by the passed `Props` to the given path. An actor incarnation is identified by the path *and a UID*. A restart only swaps the `Actor` instance defined by the `Props` but the incarnation and hence the UID remains the same.

The lifecycle of an incarnation ends when the actor is stopped. At that point the appropriate lifecycle events are called and watching actors are notified of the termination. After the incarnation is stopped, the path can be reused again by creating an actor with `actorOf()`. In this case the name of the new incarnation will be the same as the previous one but the UIDs will differ.

An `ActorRef` always represents an incarnation (path and UID) not just a given path. Therefore if an actor is stopped and a new one with the same name is created an `ActorRef` of the old incarnation will not point to the new one.

`ActorSelection` on the other hand points to the path (or multiple paths if wildcards are used) and is completely oblivious to which incarnation is currently occupying it. `ActorSelection` cannot be watched for this reason. It is possible to resolve the current incarnation's `ActorRef` living under the path by sending an `Identify` message to the `ActorSelection` which will be replied to with an `ActorIdentity` containing the correct reference (see *Identifying Actors via Actor Selection*). This can also be done with the `resolveOne` method of the `ActorSelection`, which returns a `Future` of the matching `ActorRef`.

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:

```
public class WatchActor extends AbstractActor {
  private final ActorRef child = context().actorOf(Props.empty(), "target");
  private ActorRef lastSender = system.deadLetters();

  public WatchActor() {
    context().watch(child); // <-- this is the only call needed for registration

    receive(ReceiveBuilder.
      matchEquals("kill", s -> {
        context().stop(child);
        lastSender = sender();
      }).
      match(Terminated.class, t -> t.actor().equals(child), t -> {
        lastSender.tell("finished", self());
      }).build()
    );
  }
}
```

It should be noted that the `Terminated` message is generated independent of the order in which registration and termination occur. In particular, the watching actor will receive a `Terminated` message even if the watched actor has already been terminated at the time of registration.

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)`. This works even if the `Terminated` message has already been enqueued in the mailbox; after calling `unwatch` no `Terminated` message for that actor will be processed anymore.

Start Hook

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

```
@Override
public void preStart() {
  target = context().actorOf(Props.create(MyActor.class, "target"));
}
```

This method is called when the actor is first created. During restarts it is called by the default implementation of `postRestart`, which means that by overriding that method you can choose whether the initialization code in this method is called only exactly once for this actor or for every restart. Initialization code which is part of the

actor's constructor will always be called when an instance of the actor class is created, which happens at every restart.

Restart Hooks

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors may be restarted in case an exception is thrown while processing a message (see *Supervision and Monitoring*). 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, or if an actor is restarted due to a sibling's failure. If the message is available, then that message's sender is also accessible in the usual way (i.e. by calling `sender`).

   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.

> **Warning:** Be aware that the ordering of failure notifications relative to user messages is not deterministic. In particular, a parent might restart its child before it has processed the last messages sent by the child before the failure. See *Discussion: Message Ordering* for details.

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`.

8.3.3 Identifying Actors via Actor Selection

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 `ActorSelection` with the result:

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

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.

The path elements of an actor selection may contain wildcard patterns allowing for broadcasting of messages to that section:

```
// will look all children to serviceB with names starting with worker
context().actorSelection("/user/serviceB/worker*");
// will look up all siblings beneath same supervisor
context().actorSelection("../*");
```

Messages can be sent via the `ActorSelection` and the path of the `ActorSelection` is looked up when delivering each message. If the selection does not match any actors the message will be dropped.

To acquire an `ActorRef` for an `ActorSelection` you need to send a message to the selection and use the `sender()` reference of the reply from the actor. There is a built-in `Identify` message that all Actors will understand and automatically reply to with a `ActorIdentity` message containing the `ActorRef`. This message is handled specially by the actors which are traversed in the sense that if a concrete name lookup fails (i.e. a non-wildcard path element does not correspond to a live actor) then a negative result is generated. Please note that this does not mean that delivery of that reply is guaranteed, it still is a normal message.

```
import akka.actor.ActorIdentity;
import akka.actor.ActorSelection;
import akka.actor.Identify;
```

```
public class Follower extends AbstractActor {
  final Integer identifyId = 1;

  public Follower(){
    ActorSelection selection = context().actorSelection("/user/another");
    selection.tell(new Identify(identifyId), self());

    receive(ReceiveBuilder.
      match(ActorIdentity.class, id -> id.getRef() != null, id -> {
        ActorRef ref = id.getRef();
        context().watch(ref);
        context().become(active(ref));
      }).
      match(ActorIdentity.class, id -> id.getRef() == null, id -> {
        context().stop(self());
      }).build()
    );
  }

  final PartialFunction<Object, BoxedUnit> active(final ActorRef another) {
    return ReceiveBuilder.
      match(Terminated.class, t -> t.actor().equals(another), t -> {
        context().stop(self());
      }).build();
  }
}
```

You can also acquire an `ActorRef` for an `ActorSelection` with the `resolveOne` method of the `ActorSelection`. It returns a `Future` of the matching `ActorRef` if such an actor exists. It is completed with failure [[akka.actor.ActorNotFound]] if no such actor exists or the identification didn't complete within the supplied *timeout*.

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

```
context().actorSelection("akka.tcp://app@otherhost:1234/user/serviceB");
```

An example demonstrating actor look-up is given in *remote-sample-java*.

Note: `actorFor` is deprecated in favor of `actorSelection` because actor references acquired with `actorFor` behaves different for local and remote actors. In the case of a local actor reference, the named actor needs to exist before the lookup, or else the acquired reference will be an `EmptyLocalActorRef`. This

will be true even if an actor with that exact path is created after acquiring the actor reference. For remote actor references acquired with *actorFor* the behaviour is different and sending messages to such a reference will under the hood look up the actor by path on the remote system for every message send.

8.3.4 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;
  }
}
```

8.3.5 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.

```
// don't forget to think about who is the sender (2nd argument)
target.tell(message, self());
```

The sender reference is passed along with the message and available within the receiving actor via its `sender` method while processing this message. Inside of an actor it is usually `self` who shall be the sender, but there can be cases where replies shall be routed to some other actor—e.g. the parent—in which the second argument to

`tell` would be a different one. Outside of an actor and if no reply is needed the second argument can be `null`; if a reply is needed outside of an actor you can use the ask-pattern described next..

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 scala.concurrent.Future;
import scala.concurrent.duration.Duration;
import akka.dispatch.Futures;
import akka.dispatch.Mapper;
import akka.util.Timeout;
```

```
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, "another 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 x = (String) it.next();
        final String s = (String) it.next();
        return new Result(x, s);
      }
    }, system.dispatcher());

pipe(transformed, system.dispatcher()).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 `sender().tell(reply, self())` 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.

> **Warning:** 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();
  sender().tell(result, self());
} catch (Exception e) {
  sender().tell(new akka.actor.Status.Failure(e), self());
  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`, `onSuccess`, or `onFailure` 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.

```
target.forward(result, context());
```

8.3.6 Receive messages

An Actor either has to set its initial receive behavior in the constructor by calling the `receive` method in the `AbstractActor`:

```
public SomeActor() {
  receive(ReceiveBuilder.
    // and some behavior ...
  build());
}
```

or by implementing the `receive` method in the `Actor` interface:

```
public abstract PartialFunction<Object, BoxedUnit> receive();
```

Both the argument to the `AbstractActor` `receive` method and the return type of the `Actor` `receive` method is a `PartialFunction<Object, BoxedUnit>` that defines which messages your Actor can handle, along with the implementation of how the messages should be processed.

Don't let the type signature scare you. To allow you to easily build up a partial function there is a builder named `ReceiveBuilder` that you can use.

Here is an example:

```
import akka.actor.AbstractActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.pf.ReceiveBuilder;

public class MyActor extends AbstractActor {
  private final LoggingAdapter log = Logging.getLogger(context().system(), this);

  public MyActor() {
    receive(ReceiveBuilder.
      match(String.class, s -> {
        log.info("Received String message: {}", s);
      }).
      matchAny(o -> log.info("received unknown message")).build()
    );
```

```
  }
}
```

8.3.7 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().tell(replyMsg, self())`. 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.

```
sender().tell(s, self());
```

8.3.8 Receive timeout

The *ActorContext* `setReceiveTimeout` defines the inactivity timeout after which the sending of a *ReceiveTimeout* message is triggered. When specified, the receive function should be able to handle an *akka.actor.ReceiveTimeout* message. 1 millisecond is the minimum supported timeout.

Please note that the receive timeout might fire and enqueue the *ReceiveTimeout* message right after another message was enqueued; hence it is **not guaranteed** that upon reception of the receive timeout there must have been an idle period beforehand as configured via this method.

Once set, the receive timeout stays in effect (i.e. continues firing repeatedly after inactivity periods). Pass in *Duration.Undefined* to switch off this feature.

```
public class ReceiveTimeoutActor extends AbstractActor {
  public ReceiveTimeoutActor() {
    // To set an initial delay
    context().setReceiveTimeout(Duration.create("10 seconds"));

    receive(ReceiveBuilder.
      matchEquals("Hello", s -> {
        // To set in a response to a message
        context().setReceiveTimeout(Duration.create("1 second"));
      }).
      match(ReceiveTimeout.class, r -> {
        // To turn it off
        context().setReceiveTimeout(Duration.Undefined());
      }).build()
    );
  }
}
```

8.3.9 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 internal termination notifications 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() {
  // clean up some resources ...
}
```

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 static akka.pattern.Patterns.gracefulStop;
import scala.concurrent.Await;
import scala.concurrent.Future;
import scala.concurrent.duration.Duration;
import akka.pattern.AskTimeoutException;
```

```
try {
  Future<Boolean> stopped =
    gracefulStop(actorRef, Duration.create(5, TimeUnit.SECONDS), Manager.SHUTDOWN);
  Await.result(stopped, Duration.create(6, TimeUnit.SECONDS));
  // the actor has been stopped
} catch (AskTimeoutException e) {
  // the actor wasn't stopped within 5 seconds
}
```

```
public class Manager extends AbstractActor {
  private static enum Shutdown {
    Shutdown
  }
  public static final Shutdown SHUTDOWN = Shutdown.Shutdown;

  private ActorRef worker =
  context().watch(context().actorOf(Props.create(Cruncher.class), "worker"));

  public Manager() {
    receive(ReceiveBuilder.
      matchEquals("job", s -> {
        worker.tell("crunch", self());
      }).
      matchEquals(SHUTDOWN, x -> {
        worker.tell(PoisonPill.getInstance(), self());
        context().become(shuttingDown);
      }).build()
```

```
    );
  }

  public PartialFunction<Object, BoxedUnit> shuttingDown =
    ReceiveBuilder.
      matchEquals("job", s -> {
        sender().tell("service unavailable, shutting down", self());
      }).
      match(Terminated.class, t -> t.actor().equals(worker), t -> {
        context().stop(self());
      }).build();
}
```

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()`.

In the above example a custom `Manager.Shutdown` message is sent to the target actor to initiate the process of stopping the actor. You can use `PoisonPill` for this, but then you have limited possibilities to perform interactions with other actors before stopping the target actor. Simple cleanup tasks can be handled in `postStop`.

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.

8.3.10 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<Object, BoxedUnit>` 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`:

```
public class HotSwapActor extends AbstractActor {
  private PartialFunction<Object, BoxedUnit> angry;
  private PartialFunction<Object, BoxedUnit> happy;

  public HotSwapActor() {
    angry =
      ReceiveBuilder.
        matchEquals("foo", s -> {
          sender().tell("I am already angry?", self());
        }).
        matchEquals("bar", s -> {
          context().become(happy);
        }).build();

    happy = ReceiveBuilder.
      matchEquals("bar", s -> {
        sender().tell("I am already happy :-)", self());
      }).
      matchEquals("foo", s -> {
        context().become(angry);
      }).build();
```

```
    receive(ReceiveBuilder.
      matchEquals("foo", s -> {
        context().become(angry);
      }).
      matchEquals("bar", s -> {
        context().become(happy);
      }).build()
    );
  }
}
```

This variant of the `become` method is useful for many different things, such as to implement a Finite State Machine (FSM, for an example see Dining Hakkers). It will replace the current behavior (i.e. the top of the behavior stack), which means that you do not use `unbecome`, instead always the next behavior is explicitly installed.

The other way of using `become` does not replace but add to the top of the behavior stack. In this case care must be taken to ensure that the number of "pop" operations (i.e. `unbecome`) matches the number of "push" ones in the long run, otherwise this amounts to a memory leak (which is why this behavior is not the default).

```
public class Swapper extends AbstractLoggingActor {
  public Swapper() {
    receive(ReceiveBuilder.
      matchEquals(Swap, s -> {
          log().info("Hi");
          context().become(ReceiveBuilder.
                  matchEquals(Swap, x -> {
                    log().info("Ho");
                    context().unbecome(); // resets the latest 'become' (just for fun)
                  }).build(), false); // push on top instead of replace
      }).build()
    );
  }
}

public class SwapperApp {
  public static void main(String[] args) {
    ActorSystem system = ActorSystem.create("SwapperSystem");
    ActorRef swapper = system.actorOf(Props.create(Swapper.class), "swapper");
    swapper.tell(Swap, ActorRef.noSender()); // logs Hi
    swapper.tell(Swap, ActorRef.noSender()); // logs Ho
    swapper.tell(Swap, ActorRef.noSender()); // logs Hi
    swapper.tell(Swap, ActorRef.noSender()); // logs Ho
    swapper.tell(Swap, ActorRef.noSender()); // logs Hi
    swapper.tell(Swap, ActorRef.noSender()); // logs Ho
    system.shutdown();
  }
}
```

8.3.11 Stash

The `AbstractActorWithStash` class enables an actor to temporarily stash away messages that can not or should not be handled using the actor's current behavior. Upon changing the actor's message handler, i.e., right before invoking `context().become()` or `context().unbecome()`, all stashed messages can be "unstashed", thereby prepending them to the actor's mailbox. This way, the stashed messages can be processed in the same order as they have been received originally. An actor that extends `AbstractActorWithStash` will automatically get a deque-based mailbox.

Note: The abstract class `AbstractActorWithStash` implements the marker interface `RequiresMessageQueue<DequeBasedMessageQueueSemantics>` which requests the system

to automatically choose a deque based mailbox implementation for the actor. If you want more control over the mailbox, see the documentation on mailboxes: *mailboxes-java*.

Here is an example of the `AbstractActorWithStash` class in action:

```
public class ActorWithProtocol extends AbstractActorWithStash {
  public ActorWithProtocol() {
    receive(ReceiveBuilder.
      matchEquals("open", s -> {
        context().become(ReceiveBuilder.
          matchEquals("write", ws -> { /* do writing */ }).
          matchEquals("close", cs -> {
            unstashAll();
            context().unbecome();
          }).
          matchAny(msg -> stash()).build(), false);
      }).
      matchAny(msg -> stash()).build()
    );
  }
}
```

Invoking `stash()` adds the current message (the message that the actor received last) to the actor's stash. It is typically invoked when handling the default case in the actor's message handler to stash messages that aren't handled by the other cases. It is illegal to stash the same message twice; to do so results in an `IllegalStateException` being thrown. The stash may also be bounded in which case invoking `stash()` may lead to a capacity violation, which results in a `StashOverflowException`. The capacity of the stash can be configured using the `stash-capacity` setting (an `Int`) of the mailbox's configuration.

Invoking `unstashAll()` enqueues messages from the stash to the actor's mailbox until the capacity of the mailbox (if any) has been reached (note that messages from the stash are prepended to the mailbox). In case a bounded mailbox overflows, a `MessageQueueAppendFailedException` is thrown. The stash is guaranteed to be empty after calling `unstashAll()`.

The stash is backed by a `scala.collection.immutable.Vector`. As a result, even a very large number of messages may be stashed without a major impact on performance.

Note that the stash is part of the ephemeral actor state, unlike the mailbox. Therefore, it should be managed like other parts of the actor's state which have the same property. The `AbstractActorWithStash` implementation of `preRestart` will call `unstashAll()`, which is usually the desired behavior.

Note: If you want to enforce that your actor can only work with an unbounded stash, then you should use the `AbstractActorWithUnboundedStash` class instead.

8.3.12 Killing an Actor

You can kill an actor by sending a `Kill` message. This will cause the actor to throw a `ActorKilledException`, triggering a failure. The actor will suspend operation and its supervisor will be asked how to handle the failure, which may mean resuming the actor, restarting it or terminating it completely. See *What Supervision Means* for more information.

Use `Kill` like this:

```
victim.tell(akka.actor.Kill.getInstance(), ActorRef.noSender());
```

8.3.13 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 (i.e. taken out of its mailbox and handed over to the current behavior), 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). Another possibility would be to have a look at the *PeekMailbox pattern*.

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 code within an actor throws an exception, that actor is suspended and the supervision process is started (see *Supervision and Monitoring*). Depending on the supervisor's decision the actor is resumed (as if nothing happened), restarted (wiping out its internal state and starting from scratch) or terminated.

8.3.14 Initialization patterns

The rich lifecycle hooks of Actors provide a useful toolkit to implement various initialization patterns. During the lifetime of an `ActorRef`, an actor can potentially go through several restarts, where the old instance is replaced by a fresh one, invisibly to the outside observer who only sees the `ActorRef`.

One may think about the new instances as "incarnations". Initialization might be necessary for every incarnation of an actor, but sometimes one needs initialization to happen only at the birth of the first instance when the `ActorRef` is created. The following sections provide patterns for different initialization needs.

Initialization via constructor

Using the constructor for initialization has various benefits. First of all, it makes it possible to use `val` fields to store any state that does not change during the life of the actor instance, making the implementation of the actor more robust. The constructor is invoked for every incarnation of the actor, therefore the internals of the actor can always assume that proper initialization happened. This is also the drawback of this approach, as there are cases when one would like to avoid reinitializing internals on restart. For example, it is often useful to preserve child actors across restarts. The following section provides a pattern for this case.

Initialization via preStart

The method `preStart()` of an actor is only called once directly during the initialization of the first instance, that is, at creation of its `ActorRef`. In the case of restarts, `preStart()` is called from `postRestart()`, therefore if not overridden, `preStart()` is called on every incarnation. However, overriding `postRestart()` one can disable this behavior, and ensure that there is only one call to `preStart()`.

One useful usage of this pattern is to disable creation of new `ActorRefs` for children during restarts. This can be achieved by overriding `preRestart()`:

```
@Override
public void preStart() {
  // Initialize children here
}

// Overriding postRestart to disable the call to preStart()
// after restarts
```

```
@Override
public void postRestart(Throwable reason) {
}

// The default implementation of preRestart() stops all the children
// of the actor. To opt-out from stopping the children, we
// have to override preRestart()
@Override
public void preRestart(Throwable reason, Option<Object> message)
  throws Exception {
  // Keep the call to postStop(), but no stopping of children
  postStop();
}
```

Please note, that the child actors are *still restarted*, but no new `ActorRef` is created. One can recursively apply the same principles for the children, ensuring that their `preStart()` method is called only at the creation of their refs.

For more information see *What Restarting Means*.

Initialization via message passing

There are cases when it is impossible to pass all the information needed for actor initialization in the constructor, for example in the presence of circular dependencies. In this case the actor should listen for an initialization message, and use `become()` or a finite state-machine state transition to encode the initialized and uninitialized states of the actor.

```
receive(ReceiveBuilder.
  matchEquals("init", m1 -> {
    initializeMe = "Up and running";
    context().become(ReceiveBuilder.
      matchEquals("U OK?", m2 -> {
        sender().tell(initializeMe, self());
      }).build());
  }).build()
```

If the actor may receive messages before it has been initialized, a useful tool can be the `Stash` to save messages until the initialization finishes, and replaying them after the actor became initialized.

Warning: This pattern should be used with care, and applied only when none of the patterns above are applicable. One of the potential issues is that messages might be lost when sent to remote actors. Also, publishing an `ActorRef` in an uninitialized state might lead to the condition that it receives a user message before the initialization has been done.

8.3.15 Lambdas and Performance

There is one big difference between the optimized partial functions created by the Scala compiler and the ones created by the `ReceiveBuilder`. The partial functions created by the `ReceiveBuilder` consist of multiple lambda expressions for every match statement, where each lambda is an object referencing the code to be run. This is something that the JVM can have problems optimizing and the resulting code might not be as performant as the Scala equivalent or the corresponding *untyped actor* version.

8.4 FSM (Java with Lambda Support)

8.4.1 Overview

The FSM (Finite State Machine) is available as an abstract base class that implements an 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'.*

Warning: The Java with lambda support part of Akka is marked as **"experimental"** as of its introduction in Akka 2.3.0. We will continue to improve this API based on our users' feedback, which implies that while we try to keep incompatible changes to a minimum, but the binary compatibility guarantee for maintenance releases does not apply to the `akka.actor.AbstractFSM`, related classes and the `akka.japi.pf` package.

8.4.2 A Simple Example

To demonstrate most of the features of the `AbstractFSM` class, 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.AbstractFSM;
import akka.actor.ActorRef;
import akka.japi.pf.UnitMatch;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;
import scala.concurrent.duration.Duration;
```

The contract of our "Buncher" actor is that it accepts or produces the following messages:

```
public final class SetTarget {
  private final ActorRef ref;

  public SetTarget(ActorRef ref) {
    this.ref = ref;
  }

  public ActorRef getRef() {
    return ref;
  }
  // boilerplate ...
}

public final class Queue {
  private final Object obj;

  public Queue(Object obj) {
    this.obj = obj;
  }

  public Object getObj() {
    return obj;
  }
```

```
  // boilerplate ...
}

public final class Batch {
  private final List<Object> list;

  public Batch(List<Object> list) {
    this.list = list;
  }

  public List<Object> getList() {
    return list;
  }
  // boilerplate ...
}

public enum Flush {
  Flush
}
```

`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.

The actor can be in two states: no message queued (aka `Idle`) or some message queued (aka `Active`). The states and the state data is defined like this:

```
// states
enum State {
  Idle, Active
}

// state data
interface Data {
}

enum Uninitialized implements Data {
  Uninitialized
}

final class Todo implements Data {
  private final ActorRef target;
  private final List<Object> queue;

  public Todo(ActorRef target, List<Object> queue) {
    this.target = target;
    this.queue = queue;
  }

  public ActorRef getTarget() {
    return target;
  }

  public List<Object> getQueue() {
    return queue;
  }
  // boilerplate ...
}
```

The actor starts out in the idle state. Once a message arrives it will go to the active state and stay there 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:

```
public class Buncher extends AbstractFSM<State, Data> {
  {
    startWith(Idle, Uninitialized);

    when(Idle,
      matchEvent(SetTarget.class, Uninitialized.class,
        (setTarget, uninitialized) ->
          stay().using(new Todo(setTarget.getRef(), new LinkedList<>()))));

    // transition elided ...

    when(Active, Duration.create(1, "second"),
      matchEvent(Arrays.asList(Flush.class, StateTimeout()), Todo.class,
        (event, todo) -> goTo(Idle).using(todo.copy(new LinkedList<>()))));

    // unhandled elided ...

    initialize();
  }
}
```

The basic strategy is to declare the actor, by inheriting the `AbstractFSM` class 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:

- `startWith` 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(
  matchEvent(Queue.class, Todo.class,
    (queue, todo) -> goTo(Active).using(todo.addElement(queue.getObj()))).
    anyEvent((event, state) -> {
      log().warning("received unhandled request {} in state {}/{}",
        event, stateName(), state);
      return 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(
  matchState(Active, Idle, () -> {
    // reuse this matcher
    final UnitMatch<Data> m = UnitMatch.create(
```

```
      matchData(Todo.class,
        todo -> todo.getTarget().tell(new Batch(todo.getQueue()), self())));
    m.match(stateData());
  }).
  state(Idle, Active, () -> {/* Do something here */}));
```

The transition callback is a partial function which takes as input a pair of states—the current and the next state. 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*, here using JUnit as an example:

```
public class BuncherTest {

  static ActorSystem system;

  @BeforeClass
  public static void setup() {
    system = ActorSystem.create("BuncherTest");
  }

  @AfterClass
  public static void tearDown() {
    JavaTestKit.shutdownActorSystem(system);
    system = null;
  }

  @Test
  public void testBuncherActorBatchesCorrectly() {
    new JavaTestKit(system) {{
      final ActorRef buncher =
        system.actorOf(Props.create(Buncher.class));
      final ActorRef probe = getRef();

      buncher.tell(new SetTarget(probe), probe);
      buncher.tell(new Queue(42), probe);
      buncher.tell(new Queue(43), probe);
      LinkedList<Object> list1 = new LinkedList<>();
      list1.add(42);
      list1.add(43);
      expectMsgEquals(new Batch(list1));
      buncher.tell(new Queue(44), probe);
      buncher.tell(Flush, probe);
      buncher.tell(new Queue(45), probe);
      LinkedList<Object> list2 = new LinkedList<>();
      list2.add(44);
      expectMsgEquals(new Batch(list2));
      LinkedList<Object> list3 = new LinkedList<>();
      list3.add(45);
      expectMsgEquals(new Batch(list3));
      system.stop(buncher);
    }};
  }

  @Test
  public void testBuncherActorDoesntBatchUninitialized() {
    new JavaTestKit(system) {{
      final ActorRef buncher =
        system.actorOf(Props.create(Buncher.class));
      final ActorRef probe = getRef();

      buncher.tell(new Queue(42), probe);
```

```
      expectNoMsg();
      system.stop(buncher);
    }};
  }
}
```

8.4.3 Reference

The AbstractFSM Class

The `AbstractFSM` abstract class is the base class used to implement an FSM. It implements Actor since an Actor is created to drive the FSM.

```
public class Buncher extends AbstractFSM<State, Data> {
  {
    // fsm body ...
  }
}
```

Note: The AbstractFSM class defines a `receive` method which handles internal messages and passes everything else through to the FSM logic (according to the current state). When overriding the `receive` method, keep in mind that e.g. state timeout handling depends on actually passing the messages through the FSM logic.

The `AbstractFSM` class takes two type parameters:

1. the supertype of all state names, usually an enum,
2. the type of the state data which are tracked by the `AbstractFSM` 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 `AbstractFSM` class. 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 state function builder syntax as demonstrated below:

```
when(Idle,
  matchEvent(SetTarget.class, Uninitialized.class,
    (setTarget, uninitialized) ->
      stay().using(new Todo(setTarget.getRef(), new LinkedList<>()))));
```

Warning: It is required that you define handlers for each of the possible FSM states, otherwise there will be failures when trying to switch to undeclared states.

It is recommended practice to declare the states as an enum and then verify that there is a `when` clause for each of the states. If you want to leave the handling of a state "unhandled" (more below), it still needs to be declared like this:

```
when(SomeState, AbstractFSM.NullFunction());
```

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(
  matchEvent(X.class, (x, data) -> {
    log().info("Received unhandled event: " + x);
    return stay();
  }).
  anyEvent((event, data) -> {
    log().warning("Received unknown event: " + event);
    return goTo(Error);
  }));
}
```

Within this handler the state of the FSM may be queried using the `stateName` method.

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 modifiers can be chained to achieve a nice and concise description:

```
when(SomeState, matchAnyEvent((msg, data) -> {
    return goTo(Processing).using(newData).
      forMax(Duration.create(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(
  matchState(Active, Idle, () -> setTimer("timeout",
    Tick, Duration.create(1, SECONDS), true)).
  state(Active, null, () -> cancelTimer("timeout")).
  state(null, Idle, (f, t) -> log().info("entering Idle from " + f)));
```

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:

```
public void handler(StateType from, StateType to) {
  // handle transition here
}

onTransition(this::handler);
```

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.

Stopping a listener without unregistering will not remove the listener from the subscription list; use `UnsubscribeTransitionCallback` before stopping the listener.

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. Any existing timer with the same name will automatically be canceled before adding the new timer.

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

```
isTimerActive(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(Error, matchEventEquals("stop", (event, data) -> {
  // do cleanup ...
  return 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(
  matchStop(Normal(),
    (state, data) -> {/* Do something here */}).
    stop(Shutdown(),
      (state, data) -> {/* Do something here */}).
```

```
    stop(Failure.class,
      (reason, state, data) -> {/* Do something here */}));
```

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 `AbstractFSM` class 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`.

8.4.4 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 *Configuration* enables logging of an event trace by `LoggingFSM` instances:

```
public class MyFSM extends AbstractLoggingFSM<StateType, Data> {
  // body elided ...
}
```

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 `AbstractLoggingFSM` class 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:

```
public class MyFSM extends AbstractLoggingFSM<StateType, Data> {
  @Override
  public int logDepth() { return 12; }
  {
    onTermination(
      matchStop(Failure.class, (reason, state, data) -> {
        String lastEvents = getLog().mkString("\n\t");
        log().warning("Failure in state " + state + " with data " + data + "\n" +
          "Events leading up to this point:\n\t" + lastEvents);
      })
    );
    //...
  }
}
```

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.

8.4.5 Examples

A bigger FSM example contrasted with Actor's `become`/`unbecome` can be found in the Typesafe Activator template named Akka FSM in Scala

Another reason for marking a module as experimental is that it's too early to tell if the module has a maintainer that can take the responsibility of the module over time. These modules live in the `akka-contrib` subproject:

8.5 External Contributions

This subproject provides a home to modules contributed by external developers which may or may not move into the officially supported code base over time. The conditions under which this transition can occur include:

- there must be enough interest in the module to warrant inclusion in the standard distribution,
- the module must be actively maintained and
- code quality must be good enough to allow efficient maintenance by the Akka core development team

If a contributions turns out to not "take off" it may be removed again at a later time.

8.5.1 Caveat Emptor

A module in this subproject doesn't have to obey the rule of staying binary compatible between minor releases. Breaking API changes may be introduced in minor releases without notice as we refine and simplify based on your feedback. A module may be dropped in any release without prior deprecation. The Typesafe subscription does not cover support for these modules.

8.5.2 The Current List of Modules

Reliable Proxy Pattern

Looking at *Message Delivery Reliability* one might come to the conclusion that Akka actors are made for blue-sky scenarios: sending messages is the only way for actors to communicate, and then that is not even guaranteed to work. Is the whole paradigm built on sand? Of course the answer is an emphatic "No!".

A local message send—within the same JVM instance—is not likely to fail, and if it does the reason was one of

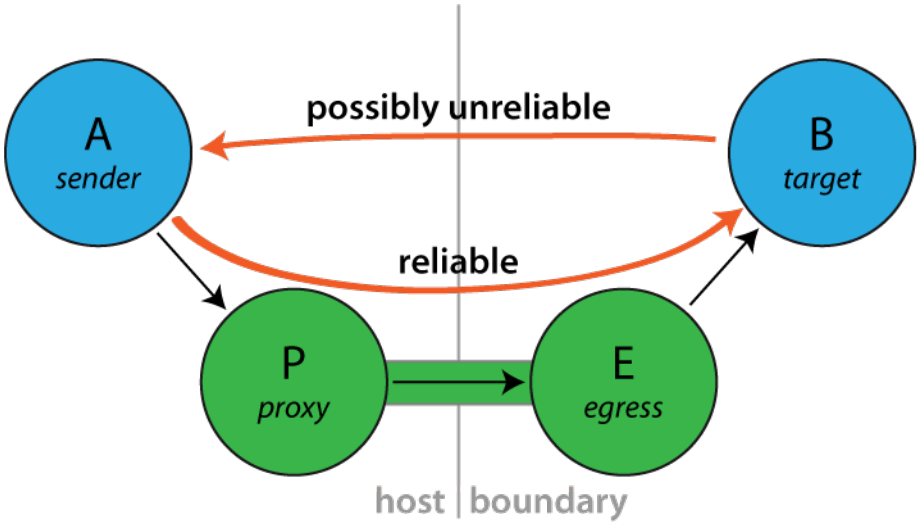
- it was meant to fail (due to consciously choosing a bounded mailbox, which upon overflow will have to drop messages)
- or it failed due to a catastrophic VM error, e.g. an `OutOfMemoryError`, a memory access violation ("segmentation fault", GPF, etc.), JVM bug—or someone calling `System.exit()`.

In all of these cases, the actor was very likely not in a position to process the message anyway, so this part of the non-guarantee is not problematic.

It is a lot more likely for an unintended message delivery failure to occur when a message send crosses JVM boundaries, i.e. an intermediate unreliable network is involved. If someone unplugs an ethernet cable, or a power failure shuts down a router, messages will be lost while the actors would be able to process them just fine.

Note: This does not mean that message send semantics are different between local and remote operations, it just means that in practice there is a difference between how good the "best effort" is.

Introducing the Reliable Proxy



To bridge the disparity between "local" and "remote" sends is the goal of this pattern. When sending from A to B must be as reliable as in-JVM, regardless of the deployment, then you can interject a reliable tunnel and send through that instead. The tunnel consists of two end-points, where the ingress point P (the "proxy") is a child of A and the egress point E is a child of P, deployed onto the same network node where B lives. Messages sent to P will be wrapped in an envelope, tagged with a sequence number and sent to E, who verifies that the received envelope has the right sequence number (the next expected one) and forwards the contained message to B. When B receives this message, the `sender()` will be a reference to the sender() of the original message to P. Reliability is added by E replying to orderly received messages with an ACK, so that P can tick those messages off its resend list. If ACKs do not come in a timely fashion, P will try to resend until successful.

Exactly what does it guarantee?

Sending via a `ReliableProxy` makes the message send exactly as reliable as if the represented target were to live within the same JVM, provided that the remote actor system does not terminate. In effect, both ends (i.e. JVM and actor system) must be considered as one when evaluating the reliability of this communication channel. The benefit is that the network in-between is taken out of that equation.

Connecting to the target The `proxy` tries to connect to the `target` using the mechanism outlined in *Identifying Actors via Actor Selection*. Once connected, if the `tunnel` terminates the `proxy` will optionally try to reconnect to the target using using the same process.

Note that during the reconnection process there is a possibility that a message could be delivered to the `target` more than once. Consider the case where a message is delivered to the `target` and the target system crashes before the ACK is sent to the `sender`. After the `proxy` reconnects to the `target` it will start resending all of the messages that it has not received an ACK for, and the message that it never got an ACK for will be redelivered. Either this possibility should be considered in the design of the `target` or reconnection should be disabled.

How to use it

Since this implementation does not offer much in the way of configuration, simply instantiate a proxy wrapping a target `ActorPath`. From Java it looks like this:

```
import akka.contrib.pattern.ReliableProxy;


  public class ProxyParent extends UntypedActor {
    private final ActorRef proxy;

    public ProxyParent(ActorPath targetPath) {
      proxy = getContext().actorOf(
          ReliableProxy.props(targetPath,
              Duration.create(100, TimeUnit.MILLISECONDS)));
    }

    public void onReceive(Object msg) {
      if ("hello".equals(msg)) {
        proxy.tell("world!", getSelf());
      }
    }
  }
```

And from Scala like this:

```
import akka.contrib.pattern.ReliableProxy

class ProxyParent(targetPath: ActorPath) extends Actor {
  val proxy = context.actorOf(ReliableProxy.props(targetPath, 100.millis))

  def receive = {
    case "hello" ⇒ proxy ! "world!"
  }
}
```

Since the `ReliableProxy` actor is an *FSM*, it also offers the capability to subscribe to state transitions. If you need to know when all enqueued messages have been received by the remote end-point (and consequently been forwarded to the target), you can subscribe to the FSM notifications and observe a transition from state `ReliableProxy.Active` to state `ReliableProxy.Idle`.

```
public class ProxyTransitionParent extends UntypedActor {
  private final ActorRef proxy;
  private ActorRef client = null;

  public ProxyTransitionParent(ActorPath targetPath) {
    proxy = getContext().actorOf(
        ReliableProxy.props(targetPath,
            Duration.create(100, TimeUnit.MILLISECONDS)));
    proxy.tell(new FSM.SubscribeTransitionCallBack(getSelf()), getSelf());
  }

  public void onReceive(Object msg) {
    if ("hello".equals(msg)) {
      proxy.tell("world!", getSelf());
      client = getSender();
    } else if (msg instanceof FSM.CurrentState<?>) {
      // get initial state
    } else if (msg instanceof FSM.Transition<?>) {
      @SuppressWarnings("unchecked")
      final FSM.Transition<ReliableProxy.State> transition =
        (FSM.Transition<ReliableProxy.State>) msg;
      assert transition.fsmRef().equals(proxy);
```

```
      if (transition.from().equals(ReliableProxy.active()) &&
              transition.to().equals(ReliableProxy.idle())) {
        client.tell("done", getSelf());
      }
    }
  }
}
```

From Scala it would look like so:

```
class ProxyTransitionParent(targetPath: ActorPath) extends Actor {
  val proxy = context.actorOf(ReliableProxy.props(targetPath, 100.millis))
  proxy ! FSM.SubscribeTransitionCallBack(self)

  var client: ActorRef = _

  def receive = {
    case "go" ⇒
      proxy ! 42
      client = sender()
    case FSM.CurrentState(`proxy`, initial) ⇒
    case FSM.Transition(`proxy`, from, to) ⇒
      if (to == ReliableProxy.Idle)
        client ! "done"
  }
}
```

Configuration

- Set `akka.reliable-proxy.debug` to `on` to turn on extra debug logging for your `ReliableProxy` actors.
- `akka.reliable-proxy.default-connect-interval` is used only if you create a `ReliableProxy` with no reconnections (that is, `reconnectAfter == None`). The default value is the value of the configuration property `akka.remote.retry-gate-closed-for`. For example, if `akka.remote.retry-gate-closed-for` is `5 s` case the `ReliableProxy` will send an `Identify` message to the *target* every 5 seconds to try to resolve the `ActorPath` to an `ActorRef` so that messages can be sent to the *target*.

The Actor Contract

Message it Processes

- `FSM.SubscribeTransitionCallBack` and `FSM.UnsubscribeTransitionCallBack`, see *FSM*
- `ReliableProxy.Unsent`, see the API documentation for details.
- any other message is transferred through the reliable tunnel and forwarded to the designated target actor

Messages it Sends

- `FSM.CurrentState` and `FSM.Transition`, see *FSM*
- `ReliableProxy.TargetChanged` is sent to the FSM transition subscribers if the proxy reconnects to a new target.
- `ReliableProxy.ProxyTerminated` is sent to the FSM transition subscribers if the proxy is stopped.

Exceptions it Escalates

- no specific exception types
- any exception encountered by either the local or remote end-point are escalated (only fatal VM errors)

Arguments it Takes

- *target* is the `ActorPath` to the actor to which the tunnel shall reliably deliver messages, `B` in the above illustration.
- *retryAfter* is the timeout for receiving ACK messages from the remote end-point; once it fires, all outstanding message sends will be retried.
- *reconnectAfter* is an optional interval between connection attempts. It is also used as the interval between receiving a `Terminated` for the tunnel and attempting to reconnect to the target actor.
- *maxConnectAttempts* is an optional maximum number of attempts to connect to the target while in the `Connecting` state.

Throttling Actor Messages

Introduction

Suppose you are writing an application that makes HTTP requests to an external web service and that this web service has a restriction in place: you may not make more than 10 requests in 1 minute. You will get blocked or need to pay if you don't stay under this limit. In such a scenario you will want to employ a *message throttler*.

This extension module provides a simple implementation of a throttling actor, the `TimerBasedThrottler`.

How to use it

You can use a `TimerBasedThrottler` as follows. From Java it looks like this:

```
  // A simple actor that prints whatever it receives
  ActorRef printer = system.actorOf(Props.create(Printer.class));
  // The throttler for this example, setting the rate
  ActorRef throttler = system.actorOf(Props.create(TimerBasedThrottler.class,
      new Throttler.Rate(3, Duration.create(1, TimeUnit.SECONDS))));
  // Set the target
  throttler.tell(new Throttler.SetTarget(printer), null);
  // These three messages will be sent to the target immediately
  throttler.tell("1", null);
  throttler.tell("2", null);
  throttler.tell("3", null);
  // These two will wait until a second has passed
  throttler.tell("4", null);
  throttler.tell("5", null);

//A simple actor that prints whatever it receives
public class Printer extends UntypedActor {
  @Override
  public void onReceive(Object msg) {
    System.out.println(msg);
  }
}
```

And from Scala like this:

```
// A simple actor that prints whatever it receives
class PrintActor extends Actor {
  def receive = {
```

```
    case x ⇒ println(x)
  }
}

    val printer = system.actorOf(Props[PrintActor])
    // The throttler for this example, setting the rate
    val throttler = system.actorOf(Props(classOf[TimerBasedThrottler],
      3 msgsPer 1.second))
    // Set the target
    throttler ! SetTarget(Some(printer))
    // These three messages will be sent to the target immediately
    throttler ! "1"
    throttler ! "2"
    throttler ! "3"
    // These two will wait until a second has passed
    throttler ! "4"
    throttler ! "5"
```

Please refer to the JavaDoc/ScalaDoc documentation for the details.

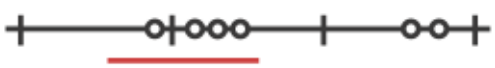
The guarantees

`TimerBasedThrottler` uses a timer internally. When the throttler's rate is 3 msg/s, for example, the throttler will start a timer that triggers every second and each time will give the throttler exactly three "vouchers"; each voucher gives the throttler a right to deliver a message. In this way, at most 3 messages will be sent out by the throttler in each interval.

It should be noted that such timer-based throttlers provide relatively **weak guarantees**:

- Only *start times* are taken into account. This may be a problem if, for example, the throttler is used to throttle requests to an external web service. If a web request takes very long on the server then the rate *observed on the server* may be higher.
- A timer-based throttler only makes guarantees for the intervals of its own timer. In our example, no more than 3 messages are delivered within such intervals. Other intervals on the timeline, however, may contain more calls.

The two cases are illustrated in the two figures below, each showing a timeline and three intervals of the timer. The message delivery times chosen by the throttler are indicated by dots, and as you can see, each interval contains at most 3 point, so the throttler works correctly. Still, there is in each example an interval (the red one) that is problematic. In the first scenario, this is because the delivery times are merely the start times of longer requests (indicated by the four bars above the timeline that start at the dots), so that the server observes four requests during the red interval. In the second scenario, the messages are centered around one of the points in time where the timer triggers, causing the red interval to contain too many messages.



For some application scenarios, the guarantees provided by a timer-based throttler might be too weak. Charles Cordingley's blog post discusses a throttler with stronger guarantees (it solves problem 2 from above). Future versions of this module may feature throttlers with better guarantees.

Java Logging (JUL)

This extension module provides a logging backend which uses the *java.util.logging* (j.u.l) API to do the endpoint logging for *akka.event.Logging*.

Provided with this module is an implementation of *akka.event.LoggingAdapter* which is independent of any *ActorSystem* being in place. This means that j.u.l can be used as the backend, via the Akka Logging API, for both Actor and non-Actor codebases.

To enable j.u.l as the *akka.event.Logging* backend, use the following Akka config:

loggers = ["akka.contrib.jul.JavaLogger"]

To access the *akka.event.Logging* API from non-Actor code, mix in *akka.contrib.jul.JavaLogging*.

This module is preferred over SLF4J with its JDK14 backend, due to integration issues resulting in the incorrect handling of *threadId*, *className* and *methodName*.

This extension module was contributed by Sam Halliday.

Mailbox with Explicit Acknowledgement

When an Akka actor is processing a message and an exception occurs, the normal behavior is for the actor to drop that message, and then continue with the next message after it has been restarted. This is in some cases not the desired solution, e.g. when using failure and supervision to manage a connection to an unreliable resource; the actor could after the restart go into a buffering mode (i.e. change its behavior) and retry the real processing later, when the unreliable resource is back online.

One way to do this is by sending all messages through the supervisor and buffering them there, acknowledging successful processing in the child; another way is to build an explicit acknowledgement mechanism into the mailbox. The idea with the latter is that a message is reprocessed in case of failure until the mailbox is told that processing was successful.

The pattern is implemented here. A demonstration of how to use it (although for brevity not a perfect example) is the following:

```
class MyActor extends Actor {
  def receive = {
    case msg ⇒
      println(msg)
      doStuff(msg) // may fail
      PeekMailboxExtension.ack()
  }

  // business logic elided ...
}

object MyApp extends App {
  val system = ActorSystem("MySystem", ConfigFactory.parseString("""
    peek-dispatcher {
      mailbox-type = "akka.contrib.mailbox.PeekMailboxType"
      max-tries = 2
    }
    """))

  val myActor = system.actorOf(Props[MyActor].withDispatcher("peek-dispatcher"),
    name = "myActor")

  myActor ! "Hello"
  myActor ! "World"
  myActor ! PoisonPill
}
```

Running this application (try it in the Akka sources by saying `sbt akka-contrib/test:run`) may produce the following output (note the processing of "World" on lines 2 and 16):

```
Hello
World
[ERROR] [12/17/2012 16:28:36.581] [MySystem-peek-dispatcher-5] [akka://MySystem/user/myActor] DON
java.lang.Exception: DONTWANNA
     at akka.contrib.mailbox.MyActor.doStuff(PeekMailbox.scala:105)
     at akka.contrib.mailbox.MyActor$$anonfun$receive$1.applyOrElse(PeekMailbox.scala:98)
     at akka.actor.ActorCell.receiveMessage(ActorCell.scala:425)
     at akka.actor.ActorCell.invoke(ActorCell.scala:386)
     at akka.dispatch.Mailbox.processMailbox(Mailbox.scala:230)
     at akka.dispatch.Mailbox.run(Mailbox.scala:212)
     at akka.dispatch.ForkJoinExecutorConfigurator$MailboxExecutionTask.exec(AbstractDispatcher.s
     at scala.concurrent.forkjoin.ForkJoinTask.doExec(ForkJoinTask.java:262)
     at scala.concurrent.forkjoin.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:975)
     at scala.concurrent.forkjoin.ForkJoinPool.runWorker(ForkJoinPool.java:1478)
     at scala.concurrent.forkjoin.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:104)
World
```

Normally one would want to make processing idempotent (i.e. it does not matter if a message is processed twice) or `context.become` a different behavior upon restart; the above example included the `println(msg)` call just to demonstrate the re-processing.

Cluster Singleton

For some use cases it is convenient and sometimes also mandatory to ensure that you have exactly one actor of a certain type running somewhere in the cluster.

Some examples:

- single point of responsibility for certain cluster-wide consistent decisions, or coordination of actions across the cluster system
- single entry point to an external system
- single master, many workers
- centralized naming service, or routing logic

Using a singleton should not be the first design choice. It has several drawbacks, such as single-point of bottleneck. Single-point of failure is also a relevant concern, but for some cases this feature takes care of that by making sure that another singleton instance will eventually be started.

The cluster singleton pattern is implemented by `akka.contrib.pattern.ClusterSingletonManager`. It manages one singleton actor instance among all cluster nodes or a group of nodes tagged with a specific role. `ClusterSingletonManager` is an actor that is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The actual singleton actor is started by the `ClusterSingletonManager` on the oldest node by creating a child actor from supplied `Props`. `ClusterSingletonManager` makes sure that at most one singleton instance is running at any point in time.

The singleton actor is always running on the oldest member, which can be determined by `Member#isOlderThan`. This can change when removing that member from the cluster. Be aware that there is a short time period when there is no active singleton during the hand-over process.

The cluster failure detector will notice when oldest node becomes unreachable due to things like JVM crash, hard shut down, or network failure. Then a new oldest node will take over and a new singleton actor is created. For these failure scenarios there will not be a graceful hand-over, but more than one active singletons is prevented by all reasonable means. Some corner cases are eventually resolved by configurable timeouts.

You can access the singleton actor by using the provided `akka.contrib.pattern.ClusterSingletonProxy`, which will route all messages to the current instance of the singleton. The proxy will keep track of the oldest node in the cluster and resolve the singleton's `ActorRef` by explicitly sending the singleton's `actorSelection` the `akka.actor.Identify` message and waiting for it to reply. This is performed periodically if the

singleton doesn't reply within a certain (configurable) time. Given the implementation, there might be periods of time during which the `ActorRef` is unavailable, e.g., when a node leaves the cluster. In these cases, the proxy will stash away all messages until it is able to identify the singleton. It's worth noting that messages can always be lost because of the distributed nature of these actors. As always, additional logic should be implemented in the singleton (acknowledgement) and in the client (retry) actors to ensure at-least-once message delivery.

An Example

Assume that we need one single entry point to an external system. An actor that receives messages from a JMS queue with the strict requirement that only one JMS consumer must exist to be make sure that the messages are processed in order. That is perhaps not how one would like to design things, but a typical real-world scenario when integrating with external systems.

On each node in the cluster you need to start the `ClusterSingletonManager` and supply the `Props` of the singleton actor, in this case the JMS queue consumer.

In Scala:

```
system.actorOf(ClusterSingletonManager.props(
  singletonProps = Props(classOf[Consumer], queue, testActor),
  singletonName = "consumer",
  terminationMessage = End,
  role = Some("worker")),
  name = "singleton")
```

Here we limit the singleton to nodes tagged with the `"worker"` role, but all nodes, independent of role, can be used by specifying `None` as `role` parameter.

The corresponding Java API for the `singeltonProps` function is `akka.contrib.pattern.ClusterSingletonPropsFactory`. The Java API takes a plain String for the role parameter and `null` means that all nodes, independent of role, are used.

In Java:

```
system.actorOf(ClusterSingletonManager.defaultProps(Props.create(Consumer.class, queue, testActor)
    new End(), "worker"), "singleton");
```

Note: The `singletonProps`/`singletonPropsFactory` is invoked when creating the singleton actor and it must not use members that are not thread safe, e.g. mutable state in enclosing actor.

Here we use an application specific `terminationMessage` to be able to close the resources before actually stopping the singleton actor. Note that `PoisonPill` is a perfectly fine `terminationMessage` if you only need to stop the actor.

Here is how the singleton actor handles the `terminationMessage` in this example.

```
case End ⇒
  queue ! UnregisterConsumer
case UnregistrationOk ⇒
  context stop self
case Ping ⇒
  sender ! Pong
```

Note that you can send back current state to the `ClusterSingletonManager` before terminating. This message will be sent over to the `ClusterSingletonManager` at the new oldest node and it will be passed to the `singletonProps` factory when creating the new singleton instance.

With the names given above, access to the singleton can be obtained from any cluster node using a properly configured proxy.

In Scala:

```
system.actorOf(ClusterSingletonProxy.props(
  singletonPath = "/user/singleton/consumer",
  role = Some("worker")),
  name = "consumerProxy")
```

In Java:

```
system.actorOf(ClusterSingletonProxy.defaultProps("user/singleton/consumer", "worker"), "consumer
```

A more comprehensive sample is available in the Typesafe Activator tutorial named Distributed workers with Akka and Scala! and Distributed workers with Akka and Java!.

Cluster Sharding

Cluster sharding is useful when you need to distribute actors across several nodes in the cluster and want to be able to interact with them using their logical identifier, but without having to care about their physical location in the cluster, which might also change over time.

It could for example be actors representing Aggregate Roots in Domain-Driven Design terminology. Here we call these actors "entries". These actors typically have persistent (durable) state, but this feature is not limited to actors with persistent state.

Cluster sharding is typically used when you have many stateful actors that together consume more resources (e.g. memory) than fit on one machine. If you only have a few stateful actors it might be easier to run them on a *Cluster Singleton* node.

In this context sharding means that actors with an identifier, so called entries, can be automatically distributed across multiple nodes in the cluster. Each entry actor runs only at one place, and messages can be sent to the entry without requiring the sender() to know the location of the destination actor. This is achieved by sending the messages via a `ShardRegion` actor provided by this extension, which knows how to route the message with the entry id to the final destination.

An Example in Java

This is how an entry actor may look like:

```
public class Counter extends UntypedEventsourcedProcessor {

  public static enum CounterOp {
    INCREMENT, DECREMENT
  }

  public static class Get {
    final public long counterId;

    public Get(long counterId) {
      this.counterId = counterId;
    }
  }

  public static class EntryEnvelope {
    final public long id;
    final public Object payload;

    public EntryEnvelope(long id, Object payload) {
      this.id = id;
      this.payload = payload;
    }
  }

  public static class CounterChanged {
```

```
    final public int delta;

    public CounterChanged(int delta) {
      this.delta = delta;
    }
  }

  int count = 0;

  @Override
  public void preStart() throws Exception {
    super.preStart();
    context().setReceiveTimeout(Duration.create(120, TimeUnit.SECONDS));
  }

  void updateState(CounterChanged event) {
    count += event.delta;
  }

  @Override
  public void onReceiveRecover(Object msg) {
    if (msg instanceof CounterChanged)
      updateState((CounterChanged) msg);
    else
      unhandled(msg);
  }

  @Override
  public void onReceiveCommand(Object msg) {
    if (msg instanceof Get)
      getSender().tell(count, getSelf());

    else if (msg == CounterOp.INCREMENT)
      persist(new CounterChanged(+1), new Procedure<CounterChanged>() {
        public void apply(CounterChanged evt) {
          updateState(evt);
        }
      });

    else if (msg == CounterOp.DECREMENT)
      persist(new CounterChanged(-1), new Procedure<CounterChanged>() {
        public void apply(CounterChanged evt) {
          updateState(evt);
        }
      });

    else if (msg.equals(ReceiveTimeout.getInstance()))
      getContext().parent().tell(
          new ShardRegion.Passivate(PoisonPill.getInstance()), getSelf());

    else
      unhandled(msg);
  }
}
```

The above actor uses event sourcing and the support provided in `UntypedEventsourcedProcessor` to store its state. It does not have to be a processor, but in case of failure or migration of entries between nodes it must be able to recover its state if it is valuable.

When using the sharding extension you are first, typically at system startup on each node in the cluster, supposed to register the supported entry types with the `ClusterSharding.start` method.

```
ClusterSharding.get(system).start("Counter", Props.create(Counter.class),
    messageExtractor);
```

The `messageExtractor` defines application specific methods to extract the entry identifier and the shard identifier from incoming messages.

```
ShardRegion.MessageExtractor messageExtractor = new ShardRegion.MessageExtractor() {

  @Override
  public String entryId(Object message) {
    if (message instanceof Counter.EntryEnvelope)
      return String.valueOf(((Counter.EntryEnvelope) message).id);
    else if (message instanceof Counter.Get)
      return String.valueOf(((Counter.Get) message).counterId);
    else
      return null;
  }

  @Override
  public Object entryMessage(Object message) {
    if (message instanceof Counter.EntryEnvelope)
      return ((Counter.EntryEnvelope) message).payload;
    else
      return message;
  }

  @Override
  public String shardId(Object message) {
    if (message instanceof Counter.EntryEnvelope) {
      long id = ((Counter.EntryEnvelope) message).id;
      return String.valueOf(id % 10);
    } else if (message instanceof Counter.Get) {
      long id = ((Counter.Get) message).counterId;
      return String.valueOf(id % 10);
    } else {
      return null;
    }
  }

};
```

This example illustrates two different ways to define the entry identifier in the messages:

- The `Get` message includes the identifier itself.
- The `EntryEnvelope` holds the identifier, and the actual message that is sent to the entry actor is wrapped in the envelope.

Note how these two messages types are handled in the `entryId` and `entryMessage` methods shown above.

A shard is a group of entries that will be managed together. The grouping is defined by the `shardResolver` function shown above. Creating a good sharding algorithm is an interesting challenge in itself. Try to produce a uniform distribution, i.e. same amount of entries in each shard. As a rule of thumb, the number of shards should be a factor ten greater than the planned maximum number of cluster nodes.

Messages to the entries are always sent via the local `ShardRegion`. The `ShardRegion` actor for a named entry type can be retrieved with `ClusterSharding.shardRegion`. The `ShardRegion` will lookup the location of the shard for the entry if it does not already know its location. It will delegate the message to the right node and it will create the entry actor on demand, i.e. when the first message for a specific entry is delivered.

```
ActorRef counterRegion = ClusterSharding.get(system).shardRegion("Counter");
counterRegion.tell(new Counter.Get(100), getSelf());

counterRegion.tell(new Counter.EntryEnvelope(100,
```

```
    Counter.CounterOp.INCREMENT), getSelf());
counterRegion.tell(new Counter.Get(100), getSelf());
```

An Example in Scala

This is how an entry actor may look like:

```
case object Increment
case object Decrement
case class Get(counterId: Long)
case class EntryEnvelope(id: Long, payload: Any)

case object Stop
case class CounterChanged(delta: Int)

class Counter extends EventsourcedProcessor {
  import ShardRegion.Passivate

  context.setReceiveTimeout(120.seconds)

  var count = 0

  def updateState(event: CounterChanged): Unit =
    count += event.delta

  override def receiveRecover: Receive = {
    case evt: CounterChanged ⇒ updateState(evt)
  }

  override def receiveCommand: Receive = {
    case Increment      ⇒ persist(CounterChanged(+1))(updateState)
    case Decrement      ⇒ persist(CounterChanged(-1))(updateState)
    case Get(_)         ⇒ sender() ! count
    case ReceiveTimeout ⇒ context.parent ! Passivate(stopMessage = Stop)
    case Stop           ⇒ context.stop(self)
  }
}
```

The above actor uses event sourcing and the support provided in `EventsourcedProcessor` to store its state. It does not have to be a processor, but in case of failure or migration of entries between nodes it must be able to recover its state if it is valuable.

When using the sharding extension you are first, typically at system startup on each node in the cluster, supposed to register the supported entry types with the `ClusterSharding.start` method.

```
ClusterSharding(system).start(
  typeName = "Counter",
  entryProps = Some(Props[Counter]),
  idExtractor = idExtractor,
  shardResolver = shardResolver)
```

The `idExtractor` and `shardResolver` are two application specific functions to extract the entry identifier and the shard identifier from incoming messages.

```
val idExtractor: ShardRegion.IdExtractor = {
  case EntryEnvelope(id, payload) ⇒ (id.toString, payload)
  case msg @ Get(id)              ⇒ (id.toString, msg)
}

val shardResolver: ShardRegion.ShardResolver = msg ⇒ msg match {
  case EntryEnvelope(id, _) ⇒ (id % 12).toString
```

```
  case Get(id)              ⇒ (id % 12).toString
}
```

This example illustrates two different ways to define the entry identifier in the messages:

- The `Get` message includes the identifier itself.
- The `EntryEnvelope` holds the identifier, and the actual message that is sent to the entry actor is wrapped in the envelope.

Note how these two messages types are handled in the `idExtractor` function shown above.

A shard is a group of entries that will be managed together. The grouping is defined by the `shardResolver` function shown above. Creating a good sharding algorithm is an interesting challenge in itself. Try to produce a uniform distribution, i.e. same amount of entries in each shard. As a rule of thumb, the number of shards should be a factor ten greater than the planned maximum number of cluster nodes.

Messages to the entries are always sent via the local `ShardRegion`. The `ShardRegion` actor for a named entry type can be retrieved with `ClusterSharding.shardRegion`. The `ShardRegion` will lookup the location of the shard for the entry if it does not already know its location. It will delegate the message to the right node and it will create the entry actor on demand, i.e. when the first message for a specific entry is delivered.

```
val counterRegion: ActorRef = ClusterSharding(system).shardRegion("Counter")
counterRegion ! Get(100)
expectMsg(0)

counterRegion ! EntryEnvelope(100, Increment)
counterRegion ! Get(100)
expectMsg(1)
```

A more comprehensive sample is available in the Typesafe Activator tutorial named Akka Cluster Sharding with Scala!.

How it works

The `ShardRegion` actor is started on each node in the cluster, or group of nodes tagged with a specific role. The `ShardRegion` is created with two application specific functions to extract the entry identifier and the shard identifier from incoming messages. A shard is a group of entries that will be managed together. For the first message in a specific shard the `ShardRegion` request the location of the shard from a central coordinator, the `ShardCoordinator`.

The `ShardCoordinator` decides which `ShardRegion` that owns the shard. The `ShardRegion` receives the decided home of the shard and if that is the `ShardRegion` instance itself it will create a local child actor representing the entry and direct all messages for that entry to it. If the shard home is another `ShardRegion` instance messages will be forwarded to that `ShardRegion` instance instead. While resolving the location of a shard incoming messages for that shard are buffered and later delivered when the shard home is known. Subsequent messages to the resolved shard can be delivered to the target destination immediately without involving the `ShardCoordinator`.

Scenario 1:

1. Incoming message M1 to `ShardRegion` instance R1.
2. M1 is mapped to shard S1. R1 doesn't know about S1, so it asks the coordinator C for the location of S1.
3. C answers that the home of S1 is R1.
4. R1 creates child actor for the entry E1 and sends buffered messages for S1 to E1 child
5. All incoming messages for S1 which arrive at R1 can be handled by R1 without C. It creates entry children as needed, and forwards messages to them.

Scenario 2:

1. Incoming message M2 to R1.

2. M2 is mapped to S2. R1 doesn't know about S2, so it asks C for the location of S2.
3. C answers that the home of S2 is R2.
4. R1 sends buffered messages for S2 to R2
5. All incoming messages for S2 which arrive at R1 can be handled by R1 without C. It forwards messages to R2.
6. R2 receives message for S2, ask C, which answers that the home of S2 is R2, and we are in Scenario 1 (but for R2).

To make sure that at most one instance of a specific entry actor is running somewhere in the cluster it is important that all nodes have the same view of where the shards are located. Therefore the shard allocation decisions are taken by the central `ShardCoordinator`, which is running as a cluster singleton, i.e. one instance on the oldest member among all cluster nodes or a group of nodes tagged with a specific role.

The logic that decides where a shard is to be located is defined in a pluggable shard allocation strategy. The default implementation `ShardCoordinator.LeastShardAllocationStrategy` allocates new shards to the `ShardRegion` with least number of previously allocated shards. This strategy can be replaced by an application specific implementation.

To be able to use newly added members in the cluster the coordinator facilitates rebalancing of shards, i.e. migrate entries from one node to another. In the rebalance process the coordinator first notifies all `ShardRegion` actors that a handoff for a shard has started. That means they will start buffering incoming messages for that shard, in the same way as if the shard location is unknown. During the rebalance process the coordinator will not answer any requests for the location of shards that are being rebalanced, i.e. local buffering will continue until the handoff is completed. The `ShardRegion` responsible for the rebalanced shard will stop all entries in that shard by sending `PoisonPill` to them. When all entries have been terminated the `ShardRegion` owning the entries will acknowledge the handoff as completed to the coordinator. Thereafter the coordinator will reply to requests for the location of the shard and thereby allocate a new home for the shard and then buffered messages in the `ShardRegion` actors are delivered to the new location. This means that the state of the entries are not transferred or migrated. If the state of the entries are of importance it should be persistent (durable), e.g. with `akka-persistence`, so that it can be recovered at the new location.

The logic that decides which shards to rebalance is defined in a pluggable shard allocation strategy. The default implementation `ShardCoordinator.LeastShardAllocationStrategy` picks shards for handoff from the `ShardRegion` with most number of previously allocated shards. They will then be allocated to the `ShardRegion` with least number of previously allocated shards, i.e. new members in the cluster. There is a configurable threshold of how large the difference must be to begin the rebalancing. This strategy can be replaced by an application specific implementation.

The state of shard locations in the `ShardCoordinator` is persistent (durable) with `akka-persistence` to survive failures. Since it is running in a cluster `akka-persistence` must be configured with a distributed journal. When a crashed or unreachable coordinator node has been removed (via down) from the cluster a new `ShardCoordinator` singleton actor will take over and the state is recovered. During such a failure period shards with known location are still available, while messages for new (unknown) shards are buffered until the new `ShardCoordinator` becomes available.

As long as a sender() uses the same `ShardRegion` actor to deliver messages to an entry actor the order of the messages is preserved. As long as the buffer limit is not reached messages are delivered on a best effort basis, with at-most once delivery semantics, in the same way as ordinary message sending. Reliable end-to-end messaging, with at-least-once semantics can be added by using channels in `akka-persistence`.

Some additional latency is introduced for messages targeted to new or previously unused shards due to the round-trip to the coordinator. Rebalancing of shards may also add latency. This should be considered when designing the application specific shard resolution, e.g. to avoid too fine grained shards.

Proxy Only Mode

The `ShardRegion` actor can also be started in proxy only mode, i.e. it will not host any entries itself, but knows how to delegate messages to the right location. A `ShardRegion` starts in proxy only mode if the roles

of the node does not include the node role specified in `akka.contrib.cluster.sharding.role` config property or if the specified *entryProps* is `None` / `null`.

Passivation

If the state of the entries are persistent you may stop entries that are not used to reduce memory consumption. This is done by the application specific implementation of the entry actors for example by defining receive timeout (`context.setReceiveTimeout`). If a message is already enqueued to the entry when it stops itself the enqueued message in the mailbox will be dropped. To support graceful passivation without loosing such messages the entry actor can send `ShardRegion.Passivate` to its parent `ShardRegion`. The specified wrapped message in `Passivate` will be sent back to the entry, which is then supposed to stop itself. Incoming messages will be buffered by the `ShardRegion` between reception of `Passivate` and termination of the entry. Such buffered messages are thereafter delivered to a new incarnation of the entry.

Configuration

The `ClusterSharding` extension can be configured with the following properties:

```
# Settings for the ClusterShardingExtension
akka.contrib.cluster.sharding {
  # The extension creates a top level actor with this name in top level user scope,
  # e.g. '/user/sharding'
  guardian-name = sharding
  # If the coordinator can't store state changes it will be stopped
  # and started again after this duration.
  coordinator-failure-backoff = 10 s
  # Start the coordinator singleton manager on members tagged with this role.
  # All members are used if undefined or empty.
  # ShardRegion actor is started in proxy only mode on nodes that are not tagged
  # with this role.
  role = ""
  # The ShardRegion retries registration and shard location requests to the
  # ShardCoordinator with this interval if it does not reply.
  retry-interval = 2 s
  # Maximum number of messages that are buffered by a ShardRegion actor.
  buffer-size = 100000
  # Timeout of the shard rebalancing process.
  handoff-timeout = 60 s
  # Rebalance check is performed periodically with this interval.
  rebalance-interval = 10 s
  # How often the coordinator saves persistent snapshots, which are
  # used to reduce recovery times
  snapshot-interval = 3600 s
  # Setting for the default shard allocation strategy
  least-shard-allocation-strategy {
    # Threshold of how large the difference between most and least number of
    # allocated shards must be to begin the rebalancing.
    rebalance-threshold = 10
    # The number of ongoing rebalancing processes is limited to this number.
    max-simultaneous-rebalance = 3
  }
}
```

Custom shard allocation strategy can be defined in an optional parameter to `ClusterSharding.start`. See the API documentation of `ShardAllocationStrategy` (Scala) or `AbstractShardAllocationStrategy` (Java) for details of how to implement a custom shard allocation strategy.

Distributed Publish Subscribe in Cluster

How do I send a message to an actor without knowing which node it is running on?

How do I send messages to all actors in the cluster that have registered interest in a named topic?

This pattern provides a mediator actor, `akka.contrib.pattern.DistributedPubSubMediator`, that manages a registry of actor references and replicates the entries to peer actors among all cluster nodes or a group of nodes tagged with a specific role.

The *DistributedPubSubMediator* is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The mediator can be started with the `DistributedPubSubExtension` or as an ordinary actor.

Changes are only performed in the own part of the registry and those changes are versioned. Deltas are disseminated in a scalable way to other nodes with a gossip protocol. The registry is eventually consistent, i.e. changes are not immediately visible at other nodes, but typically they will be fully replicated to all other nodes after a few seconds.

You can send messages via the mediator on any node to registered actors on any other node. There is three modes of message delivery.

1. DistributedPubSubMediator.Send

The message will be delivered to one recipient with a matching path, if any such exists in the registry. If several entries match the path the message will be sent via the supplied `RoutingLogic` (default random) to one destination. The sender() of the message can specify that local affinity is preferred, i.e. the message is sent to an actor in the same local actor system as the used mediator actor, if any such exists, otherwise route to any other matching entry. A typical usage of this mode is private chat to one other user in an instant messaging application. It can also be used for distributing tasks to registered workers, like a cluster aware router where the routees dynamically can register themselves.

2. DistributedPubSubMediator.SendToAll

The message will be delivered to all recipients with a matching path. Actors with the same path, without address information, can be registered on different nodes. On each node there can only be one such actor, since the path is unique within one local actor system. Typical usage of this mode is to broadcast messages to all replicas with the same path, e.g. 3 actors on different nodes that all perform the same actions, for redundancy. You can also optionally specify a property (`allButSelf`) deciding if the message should be sent to a matching path on the self node or not.

3. DistributedPubSubMediator.Publish

Actors may be registered to a named topic instead of path. This enables many subscribers on each node. The message will be delivered to all subscribers of the topic. For efficiency the message is sent over the wire only once per node (that has a matching topic), and then delivered to all subscribers of the local topic representation. This is the true pub/sub mode. A typical usage of this mode is a chat room in an instant messaging application.

You register actors to the local mediator with `DistributedPubSubMediator.Put` or `DistributedPubSubMediator.Subscribe`. `Put` is used together with `Send` and `SendToAll` message delivery modes. The `ActorRef` in `Put` must belong to the same local actor system as the mediator. `Subscribe` is used together with `Publish`. Actors are automatically removed from the registry when they are terminated, or you can explicitly remove entries with `DistributedPubSubMediator.Remove` or `DistributedPubSubMediator.Unsubscribe`.

Successful `Subscribe` and `Unsubscribe` is acknowledged with `DistributedPubSubMediator.SubscribeAck` and `DistributedPubSubMediator.UnsubscribeAck` replies.

A Small Example in Java

A subscriber actor:

```
public class Subscriber extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);
```

```
  public Subscriber() {
    ActorRef mediator =
      DistributedPubSubExtension.get(getContext().system()).mediator();
    // subscribe to the topic named "content"
    mediator.tell(new DistributedPubSubMediator.Subscribe("content", getSelf()),
      getSelf());
  }

  public void onReceive(Object msg) {
    if (msg instanceof String)
      log.info("Got: {}", msg);
    else if (msg instanceof DistributedPubSubMediator.SubscribeAck)
      log.info("subscribing");
    else
      unhandled(msg);
  }
}
```

Subscriber actors can be started on several nodes in the cluster, and all will receive messages published to the "content" topic.

```
system.actorOf(Props.create(Subscriber.class), "subscriber1");
//another node
system.actorOf(Props.create(Subscriber.class), "subscriber2");
system.actorOf(Props.create(Subscriber.class), "subscriber3");
```

A simple actor that publishes to this "content" topic:

```
public class Publisher extends UntypedActor {

  // activate the extension
  ActorRef mediator =
    DistributedPubSubExtension.get(getContext().system()).mediator();

  public void onReceive(Object msg) {
    if (msg instanceof String) {
      String in = (String) msg;
      String out = in.toUpperCase();
      mediator.tell(new DistributedPubSubMediator.Publish("content", out),
        getSelf());
    } else {
      unhandled(msg);
    }
  }
}
```

It can publish messages to the topic from anywhere in the cluster:

```
//somewhere else
ActorRef publisher = system.actorOf(Props.create(Publisher.class), "publisher");
// after a while the subscriptions are replicated
publisher.tell("hello", null);
```

A Small Example in Scala

A subscriber actor:

```
class Subscriber extends Actor with ActorLogging {
  import DistributedPubSubMediator.{ Subscribe, SubscribeAck }
  val mediator = DistributedPubSubExtension(context.system).mediator
  // subscribe to the topic named "content"
  mediator ! Subscribe("content", self)
```

```
  def receive = {
    case SubscribeAck(Subscribe("content", `self`)) ⇒
      context become ready
  }

  def ready: Actor.Receive = {
    case s: String ⇒
      log.info("Got {}", s)
  }
}
```

Subscriber actors can be started on several nodes in the cluster, and all will receive messages published to the "content" topic.

```
runOn(first) {
  system.actorOf(Props[Subscriber], "subscriber1")
}
runOn(second) {
  system.actorOf(Props[Subscriber], "subscriber2")
  system.actorOf(Props[Subscriber], "subscriber3")
}
```

A simple actor that publishes to this "content" topic:

```
class Publisher extends Actor {
  import DistributedPubSubMediator.Publish
  // activate the extension
  val mediator = DistributedPubSubExtension(context.system).mediator

  def receive = {
    case in: String ⇒
      val out = in.toUpperCase
      mediator ! Publish("content", out)
  }
}
```

It can publish messages to the topic from anywhere in the cluster:

```
runOn(third) {
  val publisher = system.actorOf(Props[Publisher], "publisher")
  later()
  // after a while the subscriptions are replicated
  publisher ! "hello"
}
```

A more comprehensive sample is available in the Typesafe Activator tutorial named Akka Clustered PubSub with Scala!.

DistributedPubSubExtension

In the example above the mediator is started and accessed with the `akka.contrib.pattern.DistributedPubSubExtension`. That is convenient and perfectly fine in most cases, but it can be good to know that it is possible to start the mediator actor as an ordinary actor and you can have several different mediators at the same time to be able to divide a large number of actors/topics to different mediators. For example you might want to use different cluster roles for different mediators.

The `DistributedPubSubExtension` can be configured with the following properties:

```
# Settings for the DistributedPubSubExtension
akka.contrib.cluster.pub-sub {
  # Actor name of the mediator actor, /user/distributedPubSubMediator
  name = distributedPubSubMediator
```

```
  # Start the mediator on members tagged with this role.
  # All members are used if undefined or empty.
  role = ""

  # The routing logic to use for 'Send'
  # Possible values: random, round-robin, consistent-hashing, broadcast
  routing-logic = random

  # How often the DistributedPubSubMediator should send out gossip information
  gossip-interval = 1s

  # Removed entries are pruned after this duration
  removed-time-to-live = 120s

  # Maximum number of elements to transfer in one message when synchronizing the registries.
  # Next chunk will be transferred in next round of gossip.
  max-delta-elements = 3000

}
```

It is recommended to load the extension when the actor system is started by defining it in `akka.extensions` configuration property. Otherwise it will be activated when first used and then it takes a while for it to be populated.

```
akka.extensions = ["akka.contrib.pattern.DistributedPubSubExtension"]
```

Cluster Client

An actor system that is not part of the cluster can communicate with actors somewhere in the cluster via this `ClusterClient`. The client can of course be part of another cluster. It only needs to know the location of one (or more) nodes to use as initial contact points. It will establish a connection to a `ClusterReceptionist` somewhere in the cluster. It will monitor the connection to the receptionist and establish a new connection if the link goes down. When looking for a new receptionist it uses fresh contact points retrieved from previous establishment, or periodically refreshed contacts, i.e. not necessarily the initial contact points. Also, note it's necessary to change `akka.actor.provider` from `akka.actor.LocalActorRefProvider` to `akka.remote.RemoteActorRefProvider` or `akka.cluster.ClusterActorRefProvider` when using the cluster client.

The receptionist is supposed to be started on all nodes, or all nodes with specified role, in the cluster. The receptionist can be started with the `ClusterReceptionistExtension` or as an ordinary actor.

You can send messages via the `ClusterClient` to any actor in the cluster that is registered in the `DistributedPubSubMediator` used by the `ClusterReceptionist`. The `ClusterReceptionistExtension` provides methods for registration of actors that should be reachable from the client. Messages are wrapped in `ClusterClient.Send`, `ClusterClient.SendToAll` or `ClusterClient.Publish`.

1. ClusterClient.Send

The message will be delivered to one recipient with a matching path, if any such exists. If several entries match the path the message will be delivered to one random destination. The sender() of the message can specify that local affinity is preferred, i.e. the message is sent to an actor in the same local actor system as the used receptionist actor, if any such exists, otherwise random to any other matching entry.

2. ClusterClient.SendToAll

The message will be delivered to all recipients with a matching path.

3. ClusterClient.Publish

The message will be delivered to all recipients Actors that have been registered as subscribers to the named topic.

Response messages from the destination actor are tunneled via the receptionist to avoid inbound connections from other cluster nodes to the client, i.e. the `sender()`, as seen by the destination actor, is not the client itself. The

`sender()` of the response messages, as seen by the client, is preserved as the original sender(), so the client can choose to send subsequent messages directly to the actor in the cluster.

While establishing a connection to a receptionist the `ClusterClient` will buffer messages and send them when the connection is established. If the buffer is full the `ClusterClient` will throw `akka.actor.StashOverflowException`, which can be handled in by the supervision strategy of the parent actor. The size of the buffer can be configured by the following `stash-capacity` setting of the mailbox that is used by the `ClusterClient` actor.

```
akka.contrib.cluster.client {
  mailbox {
    mailbox-type = "akka.dispatch.UnboundedDequeBasedMailbox"
    stash-capacity = 1000
  }
}
```

An Example

On the cluster nodes first start the receptionist. Note, it is recommended to load the extension when the actor system is started by defining it in the `akka.extensions` configuration property:

```
akka.extensions = ["akka.contrib.pattern.ClusterReceptionistExtension"]
```

Next, register the actors that should be available for the client.

```
runOn(host1) {
  val serviceA = system.actorOf(Props[Service], "serviceA")
  ClusterReceptionistExtension(system).registerService(serviceA)
}

runOn(host2, host3) {
  val serviceB = system.actorOf(Props[Service], "serviceB")
  ClusterReceptionistExtension(system).registerService(serviceB)
}
```

On the client you create the `ClusterClient` actor and use it as a gateway for sending messages to the actors identified by their path (without address information) somewhere in the cluster.

```
runOn(client) {
  val c = system.actorOf(ClusterClient.props(initialContacts))
  c ! ClusterClient.Send("/user/serviceA", "hello", localAffinity = true)
  c ! ClusterClient.SendToAll("/user/serviceB", "hi")
}
```

The `initialContacts` parameter is a `Set[ActorSelection]`, which can be created like this:

```
val initialContacts = Set(
  system.actorSelection("akka.tcp://OtherSys@host1:2552/user/receptionist"),
  system.actorSelection("akka.tcp://OtherSys@host2:2552/user/receptionist"))
```

You will probably define the address information of the initial contact points in configuration or system property.

A more comprehensive sample is available in the Typesafe Activator tutorial named Distributed workers with Akka and Scala! and Distributed workers with Akka and Java!.

ClusterReceptionistExtension

In the example above the receptionist is started and accessed with the `akka.contrib.pattern.ClusterReceptionistExtension`. That is convenient and perfectly fine in most cases, but it can be good to know that it is possible to start the `akka.contrib.pattern.ClusterReceptionist` actor as an ordinary actor and you can have several different receptionists at the same time, serving different types of clients.

The `ClusterReceptionistExtension` can be configured with the following properties:

```
# Settings for the ClusterReceptionistExtension
akka.contrib.cluster.receptionist {
  # Actor name of the ClusterReceptionist actor, /user/receptionist
  name = receptionist

  # Start the receptionist on members tagged with this role.
  # All members are used if undefined or empty.
  role = ""

  # The receptionist will send this number of contact points to the client
  number-of-contacts = 3

  # The actor that tunnel response messages to the client will be stopped
  # after this time of inactivity.
  response-tunnel-receive-timeout = 30s
}
```

Note that the `ClusterReceptionistExtension` uses the `DistributedPubSubExtension`, which is described in *Distributed Publish Subscribe in Cluster*.

It is recommended to load the extension when the actor system is started by defining it in the `akka.extensions` configuration property:

```
akka.extensions = ["akka.contrib.pattern.ClusterReceptionistExtension"]
```

Aggregator Pattern

The aggregator pattern supports writing actors that aggregate data from multiple other actors and updates its state based on those responses. It is even harder to optionally aggregate more data based on the runtime state of the actor or take certain actions (sending another message and get another response) based on two or more previous responses.

A common thought is to use the ask pattern to request information from other actors. However, ask creates another actor specifically for the ask. We cannot use a callback from the future to update the state as the thread executing the callback is not defined. This will likely close-over the current actor.

The aggregator pattern solves such scenarios. It makes sure we're acting from the same actor in the scope of the actor receive.

Introduction

The aggregator pattern allows match patterns to be dynamically added to and removed from an actor from inside the message handling logic. All match patterns are called from the receive loop and run in the thread handling the incoming message. These dynamically added patterns and logic can safely read and/or modify this actor's mutable state without risking integrity or concurrency issues.

Usage

To use the aggregator pattern, you need to extend the `Aggregator` trait. The trait takes care of `receive` and actors extending this trait should not override `receive`. The trait provides the `expect`, `expectOnce`, and `unexpect` calls. The `expect` and `expectOnce` calls return a handle that can be used for later de-registration by passing the handle to `unexpect`.

`expect` is often used for standing matches such as catching error messages or timeouts.

```
expect {
  case TimedOut ⇒ collectBalances(force = true)
}
```

`expectOnce` is used for matching the initial message as well as other requested messages

```
expectOnce {
  case GetCustomerAccountBalances(id, types) ⇒
    new AccountAggregator(sender(), id, types)
  case _ ⇒
    sender() ! CantUnderstand
    context.stop(self)
}
```

```
def fetchCheckingAccountsBalance() {
  context.actorOf(Props[CheckingAccountProxy]) ! GetAccountBalances(id)
  expectOnce {
    case CheckingAccountBalances(balances) ⇒
      results += (Checking -> balances)
      collectBalances()
  }
}
```

`unexpect` can be used for expecting multiple responses until a timeout or when the logic dictates such an `expect` no longer applies.

```
val handle = expect {
  case Response(name, value) ⇒
    values += value
    if (values.size > 3) processList()
  case TimedOut ⇒ processList()
}

def processList() {
  unexpect(handle)

  if (values.size > 0) {
    context.actorSelection("/user/evaluator") ! values.toList
    expectOnce {
      case EvaluationResults(name, eval) ⇒ processFinal(eval)
    }
  } else processFinal(List.empty[Int])
}
```

As the name eludes, `expect` keeps the partial function matching any received messages until `unexpect` is called or the actor terminates, whichever comes first. On the other hand, `expectOnce` removes the partial function once a match has been established.

It is a common pattern to register the initial expectOnce from the construction of the actor to accept the initial message. Once that message is received, the actor starts doing all aggregations and sends the response back to the original requester. The aggregator should terminate after the response is sent (or timed out). A different original request should use a different actor instance.

As you can see, aggregator actors are generally stateful, short lived actors.

Sample Use Case - AccountBalanceRetriever

This example below shows a typical and intended use of the aggregator pattern.

```
import scala.collection._
import scala.concurrent.duration._
import scala.math.BigDecimal.int2bigDecimal

import akka.actor._
/**
 * Sample and test code for the aggregator patter.
 * This is based on Jamie Allen's tutorial at
```

```
 * http://jaxenter.com/tutorial-asynchronous-programming-with-akka-actors-46220.html
 */

sealed trait AccountType
case object Checking extends AccountType
case object Savings extends AccountType
case object MoneyMarket extends AccountType

case class GetCustomerAccountBalances(id: Long, accountTypes: Set[AccountType])
case class GetAccountBalances(id: Long)

case class AccountBalances(accountType: AccountType,
                           balance: Option[List[(Long, BigDecimal)]])

case class CheckingAccountBalances(balances: Option[List[(Long, BigDecimal)]])
case class SavingsAccountBalances(balances: Option[List[(Long, BigDecimal)]])
case class MoneyMarketAccountBalances(balances: Option[List[(Long, BigDecimal)]])

case object TimedOut
case object CantUnderstand

class SavingsAccountProxy extends Actor {
  def receive = {
    case GetAccountBalances(id: Long) ⇒
      sender() ! SavingsAccountBalances(Some(List((1, 150000), (2, 29000))))
  }
}
class CheckingAccountProxy extends Actor {
  def receive = {
    case GetAccountBalances(id: Long) ⇒
      sender() ! CheckingAccountBalances(Some(List((3, 15000))))
  }
}
class MoneyMarketAccountProxy extends Actor {
  def receive = {
    case GetAccountBalances(id: Long) ⇒
      sender() ! MoneyMarketAccountBalances(None)
  }
}

class AccountBalanceRetriever extends Actor with Aggregator {

  import context._

  expectOnce {
    case GetCustomerAccountBalances(id, types) ⇒
      new AccountAggregator(sender(), id, types)
    case _ ⇒
      sender() ! CantUnderstand
      context.stop(self)
  }

  class AccountAggregator(originalSender: ActorRef,
                          id: Long, types: Set[AccountType]) {

    val results =
      mutable.ArrayBuffer.empty[(AccountType, Option[List[(Long, BigDecimal)]])]

    if (types.size > 0)
      types foreach {
        case Checking    ⇒ fetchCheckingAccountsBalance()
        case Savings     ⇒ fetchSavingsAccountsBalance()
        case MoneyMarket ⇒ fetchMoneyMarketAccountsBalance()
```

```
      }
    else collectBalances() // Empty type list yields empty response

    context.system.scheduler.scheduleOnce(1.second, self, TimedOut)
    expect {
      case TimedOut ⇒ collectBalances(force = true)
    }

    def fetchCheckingAccountsBalance() {
      context.actorOf(Props[CheckingAccountProxy]) ! GetAccountBalances(id)
      expectOnce {
        case CheckingAccountBalances(balances) ⇒
          results += (Checking -> balances)
          collectBalances()
      }
    }

    def fetchSavingsAccountsBalance() {
      context.actorOf(Props[SavingsAccountProxy]) ! GetAccountBalances(id)
      expectOnce {
        case SavingsAccountBalances(balances) ⇒
          results += (Savings -> balances)
          collectBalances()
      }
    }

    def fetchMoneyMarketAccountsBalance() {
      context.actorOf(Props[MoneyMarketAccountProxy]) ! GetAccountBalances(id)
      expectOnce {
        case MoneyMarketAccountBalances(balances) ⇒
          results += (MoneyMarket -> balances)
          collectBalances()
      }
    }

    def collectBalances(force: Boolean = false) {
      if (results.size == types.size || force) {
        originalSender ! results.toList // Make sure it becomes immutable
        context.stop(self)
      }
    }
  }
}
```

Sample Use Case - Multiple Response Aggregation and Chaining

A shorter example showing aggregating responses and chaining further requests.

```
case class InitialRequest(name: String)
case class Request(name: String)
case class Response(name: String, value: String)
case class EvaluationResults(name: String, eval: List[Int])
case class FinalResponse(qualifiedValues: List[String])

/**
 * An actor sample demonstrating use of unexpect and chaining.
 * This is just an example and not a complete test case.
 */
class ChainingSample extends Actor with Aggregator {

  expectOnce {
```

```
    case InitialRequest(name) ⇒ new MultipleResponseHandler(sender(), name)
  }

  class MultipleResponseHandler(originalSender: ActorRef, propName: String) {

    import context.dispatcher
    import collection.mutable.ArrayBuffer

    val values = ArrayBuffer.empty[String]

    context.actorSelection("/user/request_proxies") ! Request(propName)
    context.system.scheduler.scheduleOnce(50.milliseconds, self, TimedOut)

    val handle = expect {
      case Response(name, value) ⇒
        values += value
        if (values.size > 3) processList()
      case TimedOut ⇒ processList()
    }

    def processList() {
      unexpect(handle)

      if (values.size > 0) {
        context.actorSelection("/user/evaluator") ! values.toList
        expectOnce {
          case EvaluationResults(name, eval) ⇒ processFinal(eval)
        }
      } else processFinal(List.empty[Int])
    }

    def processFinal(eval: List[Int]) {
      // Select only the entries coming back from eval
      originalSender ! FinalResponse(eval map values)
      context.stop(self)
    }
  }
}
```

Pitfalls

- The current implementation does not match the sender of the message. This is designed to work with `ActorSelection` as well as `ActorRef`. Without the sender(), there is a chance a received message can be matched by more than one partial function. The partial function that was registered via `expect` or `expectOnce` first (chronologically) and is not yet de-registered by `unexpect` takes precedence in this case. Developers should make sure the messages can be uniquely matched or the wrong logic can be executed for a certain message.
- The `sender` referenced in any `expect` or `expectOnce` logic refers to the sender() of that particular message and not the sender() of the original message. The original sender() still needs to be saved so a final response can be sent back.
- `context.become` is not supported when extending the `Aggregator` trait.
- We strongly recommend against overriding `receive`. If your use case really dictates, you may do so with extreme caution. Always provide a pattern match handling aggregator messages among your `receive` pattern matches, as follows:

```
    case msg if handleMessage(msg) ⇒ // noop
    // side effects of handleMessage does the actual match
```

Sorry, there is not yet a Java implementation of the aggregator pattern available.

8.5.3 Suggested Way of Using these Contributions

Since the Akka team does not restrict updates to this subproject even during otherwise binary compatible releases, and modules may be removed without deprecation, it is suggested to copy the source files into your own code base, changing the package name. This way you can choose when to update or which fixes to include (to keep binary compatibility if needed) and later releases of Akka do not potentially break your application.

8.5.4 Suggested Format of Contributions

Each contribution should be a self-contained unit, consisting of one source file or one exclusively used package, without dependencies to other modules in this subproject; it may depend on everything else in the Akka distribution, though. This ensures that contributions may be moved into the standard distribution individually. The module shall be within a subpackage of `akka.contrib`.

Each module must be accompanied by a test suite which verifies that the provided features work, possibly complemented by integration and unit tests. The tests should follow the *Developer Guidelines* and go into the `src/test/scala` or `src/test/java` directories (with package name matching the module which is being tested). As an example, if the module were called `akka.contrib.pattern.ReliableProxy`, then the test suite should be called `akka.contrib.pattern.ReliableProxySpec`.

Each module must also have proper documentation in reStructured Text format. The documentation should be a single `<module>.rst` file in the `akka-contrib/docs` directory, including a link from `index.rst` (this file).

CHAPTER

NINE

INFORMATION FOR AKKA DEVELOPERS

9.1 Building Akka

This page describes how to build and run Akka from the latest source code.

9.1.1 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
```

9.1.2 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`.

9.1.3 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
```

Long Running and Time Sensitive Tests

By default are the long running tests (mainly cluster tests) and time sensitive tests (dependent on the performance of the machine it is running on) disabled. You can enable them by adding one of the flags:

```
-Dakka.test.tags.include=long-running
-Dakka.test.tags.include=timing
```

Or if you need to enable them both:

```
-Dakka.test.tags.include=long-running,timing
```

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
```

Note: Akka generates class diagrams for the API documentation using ScalaDoc. This needs the `dot` command from the Graphviz software package to be installed to avoid errors. You can disable the diagram generation by adding the flag `-Dakka.scaladoc.diagrams=false`

---

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
```

9.1.4 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.

9.2 Multi JVM Testing

Supports running applications (objects with main methods) and ScalaTest tests in multiple JVMs at the same time. Useful for integration testing where multiple systems communicate with each other.

9.2.1 Setup

The multi-JVM testing is an sbt plugin that you can find at http://github.com/typesafehub/sbt-multi-jvm.

You can add it as a plugin by adding the following to your project/plugins.sbt:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-multi-jvm" % "0.3.8")
```

You can then add multi-JVM testing to `build.sbt` or `project/Build.scala` by including the `MultiJvm` settings and config. Please note that MultiJvm test sources are located in `src/multi-jvm/...`, and not in `src/test/...`.

Here is an example `build.sbt` file for sbt 0.13 that uses the MultiJvm plugin:

```
import com.typesafe.sbt.SbtMultiJvm
import com.typesafe.sbt.SbtMultiJvm.MultiJvmKeys.MultiJvm

val akkaVersion = "2.3.2"

val project = Project(
  id = "akka-sample-multi-node-scala",
  base = file("."),
  settings = Project.defaultSettings ++ SbtMultiJvm.multiJvmSettings ++ Seq(
    name := "akka-sample-multi-node-scala",
    version := "1.0",
    scalaVersion := "2.10.3",
    libraryDependencies ++= Seq(
      "com.typesafe.akka" %% "akka-remote" % akkaVersion,
      "com.typesafe.akka" %% "akka-multi-node-testkit" % akkaVersion,
      "org.scalatest" %% "scalatest" % "2.0" % "test"),
    // make sure that MultiJvm test are compiled by the default test compilation
    compile in MultiJvm <<= (compile in MultiJvm) triggeredBy (compile in Test),
    // disable parallel tests
    parallelExecution in Test := false,
    // make sure that MultiJvm tests are executed by the default test target,
    // and combine the results from ordinary test and multi-jvm tests
    executeTests in Test <<= (executeTests in Test, executeTests in MultiJvm) map {
      case (testResults, multiNodeResults)  =>
        val overall =
```

```
          if (testResults.overall.id < multiNodeResults.overall.id)
            multiNodeResults.overall
          else
            testResults.overall
        Tests.Output(overall,
          testResults.events ++ multiNodeResults.events,
          testResults.summaries ++ multiNodeResults.summaries)
    }
  )
) configs (MultiJvm)
```

You can specify JVM options for the forked JVMs:

```
jvmOptions in MultiJvm := Seq("-Xmx256M")
```

9.2.2 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-tests/multi-jvm:test
```

Or one can change to the `akka-remote-tests` project first, and then run the tests:

```
project akka-remote-tests
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
```

9.2.3 Creating application tests

The tests are discovered, and combined, through a naming convention. MultiJvm test sources are located in `src/multi-jvm/...`. 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] * sample.Sample
[JVM-1] Hello from node 1
[JVM-2] Hello from node 2
[JVM-3] Hello from node 3
[success] Total time: ...
```

9.2.4 Changing Defaults

You can change the name of the multi-JVM test source directory by adding the following configuration to your project:

```
unmanagedSourceDirectories in MultiJvm <<=
   Seq(baseDirectory(_ / "src/some_directory_here")).join
```

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}`.

9.2.5 Configuration of the JVM instances

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` and `-Xmx256m` to the `SampleMultiJvmNode1` let's create three `*.opts` files and add the options to them. Separate multiple options with space.

`SampleMultiJvmNode1.opts`:

```
-Dakka.remote.port=9991 -Xmx256m
```

`SampleMultiJvmNode2.opts`:

```
-Dakka.remote.port=9992 -Xmx256m
```

`SampleMultiJvmNode3.opts`:

```
-Dakka.remote.port=9993 -Xmx256m
```

9.2.6 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.

9.2.7 Multi Node Additions

There has also been some additions made to the `SbtMultiJvm` plugin to accommodate the *experimental* module *multi node testing*, described in that section.

9.3 I/O Layer Design

The `akka.io` package has been developed in collaboration between the Akka and spray.io teams. Its design incorporates the experiences with the `spray-io` module along with improvements that were jointly developed for more general consumption as an actor-based service.

9.3.1 Requirements

In order to form a general and extensible IO layer basis for a wide range of applications, with Akka remoting and spray HTTP being the initial ones, the following requirements were established as key drivers for the design:

- scalability to millions of concurrent connections
- lowest possible latency in getting data from an input channel into the target actor's mailbox
- maximal throughput
- optional back-pressure in both directions (i.e. throttling local senders as well as allowing local readers to throttle remote senders, where allowed by the protocol)
- a purely actor-based API with immutable data representation
- extensibility for integrating new transports by way of a very lean SPI; the goal is to not force I/O mechanisms into a lowest common denominator but instead allow completely protocol-specific user-level APIs.

9.3.2 Basic Architecture

Each transport implementation will be made available as a separate Akka extension, offering an `ActorRef` representing the initial point of contact for client code. This "manager" accepts requests for establishing a communications channel (e.g. connect or listen on a TCP socket). Each communications channel is represented by one dedicated actor, which is exposed to client code for all interaction with this channel over its entire lifetime.

The central element of the implementation is the transport-specific "selector" actor; in the case of TCP this would wrap a `java.nio.channels.Selector`. The channel actors register their interest in readability or writability of their channel by sending corresponding messages to their assigned selector actor. However, the actual channel reading and writing is performed by the channel actors themselves, which frees the selector actors from time-consuming tasks and thereby ensures low latency. The selector actor's only responsibility is the management of the underlying selector's key set and the actual select operation, which is the only operation to typically block.

The assignment of channels to selectors is performed by the manager actor and remains unchanged for the entire lifetime of a channel. Thereby the management actor "stripes" new channels across one or more selector actors based on some implementation-specific distribution logic. This logic may be delegated (in part) to the selectors actors, which could, for example, choose to reject the assignment of a new channel when they consider themselves to be at capacity.

The manager actor creates (and therefore supervises) the selector actors, which in turn create and supervise their channel actors. The actor hierarchy of one single transport implementation therefore consists of three distinct actor levels, with the management actor at the top-, the channel actors at the leaf- and the selector actors at the mid-level.

Back-pressure for output is enabled by allowing the user to specify within its `Write` messages whether it wants to receive an acknowledgement for enqueuing that write to the O/S kernel. Back-pressure for input is enabled by sending the channel actor a message which temporarily disables read interest for the channel until reading is re-enabled with a corresponding resume command. In the case of transports with flow control—like TCP—the act of not consuming data at the receiving end (thereby causing them to remain in the kernels read buffers) is propagated back to the sender, linking these two mechanisms across the network.

9.3.3 Design Benefits

Staying within the actor model for the whole implementation allows us to remove the need for explicit thread handling logic, and it also means that there are no locks involved (besides those which are part of the underlying transport library). Writing only actor code results in a cleaner implementation, while Akka's efficient actor messaging does not impose a high tax for this benefit. In fact the event-based nature of I/O maps so well to the actor model that we expect clear performance and especially scalability benefits over traditional solutions with explicit thread management and synchronization.

Another benefit of supervision hierarchies is that clean-up of resources comes naturally: shutting down a selector actor will automatically clean up all channel actors, allowing proper closing of the channels and sending the appropriate messages to user-level client actors. DeathWatch allows the channel actors to notice the demise of their user-level handler actors and terminate in an orderly fashion in that case as well; this naturally reduces the chances of leaking open channels.

The choice of using `ActorRef` for exposing all functionality entails that these references can be distributed or delegated freely and in general handled as the user sees fit, including the use of remoting and life-cycle monitoring (just to name two).

9.3.4 How to go about Adding a New Transport

The best start is to study the TCP reference implementation to get a good grip on the basic working principle and then design an implementation, which is similar in spirit, but adapted to the new protocol in question. There are vast differences between I/O mechanisms (e.g. compare file I/O to a message broker) and the goal of this I/O layer is explicitly **not** to shoehorn all of them into a uniform API, which is why only the basic architecture ideas are documented here.

9.4 Developer Guidelines

Note: First read The Akka Contributor Guidelines .

9.4.1 Code Style

The Akka code style follows the Scala Style Guide . The only exception is the style of block comments:

```
/**
  * Style mandated by "Scala Style Guide"
  */

/**
 * Style adopted in the Akka codebase
 */
```

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.

9.4.2 Process

- Make sure you have signed the Akka CLA, if not, sign it online.
- 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.

9.4.3 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. Fixes #XXX

  * Details 1
  * Details 2
  * Details 3
```

9.4.4 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.

Actor TestKit

There is a useful test kit for testing actors: akka.util.TestKit. It enables assertions concerning replies received and their timing, there is more documentation in the *Testing Actor Systems* module.

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.

9.5 Documentation Guidelines

The Akka documentation uses reStructuredText as its markup language and is built using Sphinx.

9.5.1 Sphinx

For more details see The Sphinx Documentation

9.5.2 reStructuredText

For more details see The reST Quickref

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".
```

9.5.3 Build the documentation

First install Sphinx. See below.

Building

For the html version of the docs:

```
sbt sphinx:generate-html

open <project-dir>/akka-docs/target/sphinx/html/index.html
```

For the pdf version of the docs:

```
sbt sphinx:generate-pdf

open <project-dir>/akka-docs/target/sphinx/latex/AkkaJava.pdf
or
open <project-dir>/akka-docs/target/sphinx/latex/AkkaScala.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/2.7.5/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/2013basic/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
sudo tlmgr install multirow
```

If you get the error "unknown locale: UTF-8" when generating the documentation the solution is to define the following environment variables:

```
export LANG=en_US.UTF-8
export LC_ALL=en_US.UTF-8
```

9.6 Team

Name	Role
Jonas Bonér	Founder, Despot, Committer
Viktor Klang	Honorary Member
Roland Kuhn	Project Lead
Patrik Nordwall	Core Team
Björn Antonsson	Core Team
Endre Varga	Core Team
Mathias Doenitz	Committer
Johannes Rudolph	Committer
Raymond Roestenburg	Committer
Piotr Gabryanczyk	Committer
Helena Edelson	Committer
Martin Krasser	Committer
Henrik Engström	Alumnus
Peter Vlugter	Alumnus
Derek Williams	Alumnus
Debasish Ghosh	Alumnus
Ross McDonald	Alumnus
Eckhart Hertzler	Alumnus
Mikael Högqvist	Alumnus
Tim Perrett	Alumnus
Jeanfrancois Arcand	Alumnus
Jan Van Besien	Alumnus
Michael Kober	Alumnus
Peter Veentjer	Alumnus
Irmo Manie	Alumnus
Heiko Seeberger	Alumnus
Hiram Chirino	Alumnus
Scott Clasen	Alumnus

CHAPTER

TEN

PROJECT INFORMATION

10.1 Migration Guides

10.1.1 Migration Guide 1.3.x to 2.0.x

Migration from 1.3.x to 2.0.x is described in the documentation of 2.0.

10.1.2 Migration Guide 2.0.x to 2.1.x

Migration from 2.0.x to 2.1.x is described in the documentation of 2.1.

10.1.3 Migration Guide 2.1.x to 2.2.x

Migration from 2.1.x to 2.2.x is described in the documentation of 2.2.

10.1.4 Migration Guide 2.2.x to 2.3.x

The 2.3 release contains some structural changes that require some simple, mechanical source-level changes in client code.

When migrating from earlier versions you should first follow the instructions for migrating *1.3.x to 2.0.x* and then *2.0.x to 2.1.x* and then *2.1.x to 2.2.x*.

Removed hand over data in cluster singleton

The support for passing data from previous singleton instance to new instance in a graceful leaving scenario has been removed. Valuable state should be persisted in durable storage instead, e.g. using akka-persistence. The constructor/props parameters of `ClusterSingletonManager` has been changed to ordinary `Props` parameter for the singleton actor instead of the factory parameter.

Changed cluster auto-down configuration

`akka.cluster.auto-down` setting has been replaced by `akka.cluster.auto-down-unreachable-after`, which instructs the cluster to automatically mark unreachable nodes as DOWN after this configured time of unreachability. This feature is disabled by default, as it also was in 2.2.x.

During the deprecation phase `akka.cluster.auto-down=on` is interpreted at as instant auto-down.

Routers

The routers have been cleaned up and enhanced. The routing logic has been extracted to be usable within normal actors as well. Some usability problems have been have been solved, such as properly reject invalid configuration combinations. Routees can be dynamically added and removed by sending special management messages to the router.

The two types of routers have been named `Pool` and `Group` to make them more distinguishable and reduce confusion of their subtle differences:

- Pool - The router creates routees as child actors and removes them from the router if they terminate.
- Group - The routee actors are created externally to the router and the router sends messages to the specified path using actor selection, without watching for termination.

Configuration of routers is compatible with 2.2.x, but the `router` type should preferably be specified with `-pool` or `-group` suffix.

Some classes used for programmatic definition of routers have been renamed, but the old classes remain as deprecated. The compiler will guide you with deprecation warning. For example `RoundRobinRouter` has been renamed to `RoundRobinPool` or `RoundRobinGroup` depending on which type you are actually using.

There is no replacement for `SmallestMailboxRouter` combined with routee paths, i.e. a group, because that combination is not useful.

An optional API enhancement that makes the code read better is to use the `props` method instead of `withRouter`. `withRouter` has not been deprecated and you can continue to use that if you prefer that way of defining a router.

Example in Scala:

```
context.actorOf(FromConfig.props(Props[Worker]), "router1")
context.actorOf(RoundRobinPool(5).props(Props[Worker]), "router2")
```

Example in Java:

```
getContext().actorOf(FromConfig.getInstance().props(Props.create(Worker.class)),
  "router1");

getContext().actorOf(new RoundRobinPool(5).props(Props.create(Worker.class)),
    "router2");
```

To support multiple routee paths for a cluster aware router sending to paths the deployment configuration property `cluster.routees-path` has been changed to string list `routees.paths` property. The old `cluster.routees-path` is deprecated, but still working during the deprecation phase.

Example:

```
/router4 {
  router = round-robin-group
  nr-of-instances = 10
  routees.paths = ["/user/myserviceA", "/user/myserviceB"]
  cluster.enabled = on
}
```

The API for creating custom routers and resizers have changed without keeping the old API as deprecated. That should be a an API used by only a few users and they should be able to migrate to the new API without much trouble.

Read more about the new routers in the *documentation for Scala* and *documentation for Java*.

Akka IO is no longer experimental

The core IO layer introduced in Akka 2.2 is now a fully supported module of Akka.

Experimental Pipelines IO abstraction has been removed

Pipelines in the form introduced by 2.2 has been found unintuitive and are therefore discontinued. A new more flexible and easier-to-use abstraction will replace their role in the future. Pipelines will be still available in the 2.2 series.

Changed cluster expected-response-after configuration

Configuration property `akka.cluster.failure-detector.heartbeat-request.expected-response-after` has been renamed to `akka.cluster.failure-detector.expected-response-after`.

Removed automatic retry feature from Remoting in favor of retry-gate

The retry-gate feature is now the only failure handling strategy in Remoting. This change means that when remoting detects faulty connections it goes into a gated state where all buffered and subsequent remote messages are dropped until the configurable time defined by the configuration key `akka.remote.retry-gate-closed-for` elapses after the failure event. This behavior prevents reconnect storms and unbounded buffer growth during network instabilities. After the configured time elapses the gate is lifted and a new connection will be attempted when there are new remote messages to be delivered.

In concert with this change all settings related to the old reconnect behavior (`akka.remote.retry-window` and `akka.remote.maximum-retries-in-window`) were removed.

The timeout setting `akka.remote.gate-invalid-addresses-for` that controlled the gate interval for certain failure events is also removed and all gating intervals are now controlled by the `akka.remote.retry-gate-closed-for` setting instead.

Reduced default sensitivity settings for transport failure detector in Remoting

Since the most commonly used transport with Remoting is TCP, which provides proper connection termination events the failure detector sensitivity setting `akka.remote.transport-failure-detector.acceptable-heartbeat-pause` now defaults to 20 seconds to reduce load induced false-positive failure detection events in remoting. In case a non-connection-oriented protocol is used it is recommended to change this and the `akka.remote.transport-failure-detector.heartbeat-interval` setting to a more sensitive value.

Quarantine is now permanent

The setting that controlled the length of quarantine `akka.remote.quarantine-systems-for` has been removed. The only setting available now is `akka.remote.prune-quarantine-marker-after` which influences how long quarantine tombstones are kept around to avoid long-term memory leaks. This new setting defaults to 5 days.

Remoting uses a dedicated dispatcher by default

The default value of `akka.remote.use-dispatcher` has been changed to a dedicated dispatcher.

Dataflow is Deprecated

Akka dataflow is superseded by Scala Async.

Durable Mailboxes are Deprecated

Durable mailboxes are superseded by `akka-persistence`, which offers several tools to support reliable messaging.

Read more about `akka-persistence` in the *documentation for Scala* and *documentation for Java*.

Deprecated STM Support for Agents

Agents participating in enclosing STM transaction is a deprecated feature.

Transactor Module is Deprecated

The integration between actors and STM in the module `akka-transactor` is deprecated and will be removed in a future version.

Typed Channels has been removed

Typed channels were an experimental feature which we decided to remove: its implementation relied on an experimental feature of Scala for which there is no correspondence in Java and other languages and its usage was not intuitive.

Removed Deprecated Features

The following, previously deprecated, features have been removed:

- event-handlers renamed to loggers
- API changes to FSM and TestFSMRef
- DefaultScheduler superseded by LightArrayRevolverScheduler
- all previously deprecated construction and deconstruction methods for Props

publishCurrentClusterState is Deprecated

Use `sendCurrentClusterState` instead. Note that you can also retrieve the current cluster state with the new `Cluster(system).state`.

CurrentClusterState is not a ClusterDomainEvent

`CurrentClusterState` does not implement the `ClusterDomainEvent` marker interface any more.

Note the new `initialStateMode` parameter of `Cluster.subscribe`, which makes it possible to handle the initial state as events instead of `CurrentClusterState`. See *documentation for Scala* and *documentation for Java*.

BalancingDispatcher is Deprecated

Use `BalancingPool` instead of `BalancingDispatcher`. See *documentation for Scala* and *documentation for Java*.

During a migration period you can still use BalancingDispatcher by specifying the full class name in the dispatcher configuration:

```
type = "akka.dispatch.BalancingDispatcherConfigurator"
```

akka-sbt-plugin is Removed

`akka-sbt-plugin` for packaging of application binaries has been removed. Version 2.2.3 can still be used independent of Akka version of the application. Version 2.2.3 can be used with both sbt 0.12 and 0.13.

sbt-native-packager is the recommended tool for creating distributions of Akka applications when using sbt.

Parens Added to sender

Parens were added to the `sender()` method of the Actor Scala API to highlight that the `sender()` reference is not referentially transparent and must not be exposed to other threads, for example by closing over it when using future callbacks.

It is recommended to use this new convention:

```
sender() ! "reply"
```

However, it is not mandatory to use parens and you do not have to change anything.

ReliableProxy Constructor Changed

The constructor of `ReliableProxy` in `akka-contrib` has been changed to take an `ActorPath` instead of an `ActorRef`. Also it takes new parameters to support reconnection. Use the new props factory methods, `ReliableProxy.props`.

OSGi Changes

`akka-osgi` no longer contains the `akka-actor` classes, instead `akka-actor` is a bundle now. `akka-osgi` only contains a few OSGi helpers, most notably the `BundleDelegatingClassLoader` which resolves e.g. `reference.conf` files (hence these are not copied into the `akka-osgi` bundle any longer either).

`akka-osgi-aries` has been removed. Similar can be implemented outside of Akka if needed.

TestKit: reworked time dilation

`TestDuration` has been changed into an implicit value class plus a Java API in JavaTestKit. Please change:

```
import akka.testkit.duration2TestDuration
```

into:

```
import akka.testkit.TestDuration
```

10.1.5 Migration Guide Eventsourced to Akka Persistence 2.3.x

General notes

Eventsourced and Akka *Persistence* share many high-level concepts but strongly differ on design, implementation and usage level. This migration guide is more a higher-level comparison of Eventsourced and Akka Persistence rather than a sequence of low-level instructions how to transform Eventsourced application code into Akka Persistence application code. This should provide a good starting point for a migration effort. Development teams should consult the user documentation of both projects for further details.

Scope of this migration guide is code migration, not journal migration. Journals written by Eventsourced can neither be used directly Akka Persistence nor migrated to Akka Persistence compatible journals. Journal migration tools may be provided in the future but do not exist at the moment.

Extensions

Eventsourced and Akka Persistence are both *Akka Extensions*.

Eventsourced: `EventsourcingExtension`

- Must be explicitly created with an actor system and an application-defined journal actor as arguments. (see example usage).
- Processors and Channels must be created with the factory methods `processorOf` and `channelOf` defined on `EventsourcingExtension`.
- Is a central registry of created processors and channels.

Akka Persistence: `Persistence` extension

- Must **not** be explicitly created by an application. A `Persistence` extension is implicitly created upon first processor or channel creation. Journal actors are automatically created from a journal plugin configuration (see *Journal plugin API*).
- *Processors* and *Channels* can be created like any other actor with `actorOf` without using the `Persistence` extension.
- Is **not** a central registry of processors and channels.

Processors

Eventsourced: `Eventsourced`

- Stackable trait that adds journaling (write-ahead-logging) to actors (see processor definition and creation). Name `Eventsourced` caused some confusion in the past as many examples used `Eventsourced` processors for *command sourcing*. See also this FAQ entry for clarification.
- Must be explicitly recovered using recovery methods on `EventsourcingExtension`. Applications are responsible for avoiding an interference of replayed messages and new messages i.e. applications have to explicitly wait for recovery to complete. Recovery on processor re-start is not supported out-of-the box.
- Journaling-preserving behavior changes are only possible with special-purpose methods `become` and `unbecome`, in addition to non-journaling-preserving behavior changes with default methods `context.become` and `context.unbecome`.
- Writes messages of type `Message` to the journal (see processor usage). Sender references are not stored in the journal i.e. the sender reference of a replayed message is always `system.deadLetters`.
- Supports snapshots.
- Identifiers are of type `Int` and must be application-defined.
- Does not support batch-writes of messages to the journal.
- Does not support stashing of messages.

Akka Persistence: `Processor`

- Trait that adds journaling (write-ahead-logging) to actors (see *Processors*) and used by applications for *command sourcing*. Corresponds to `Eventsourced` processors in Eventsourced but is not a stackable trait.
- Automatically recovers on start and re-start, by default. *Recovery* can be customized or turned off by overriding actor life cycle hooks `preStart` and `preRestart`. `Processor` takes care that new messages never interfere with replayed messages. New messages are internally buffered until recovery completes.
- No special-purpose behavior change methods. Default behavior change methods `context.become` and `context.unbecome` can be used and are journaling-preserving.
- Writes messages of type `Persistent` to the journal (see *Persistent messages*). Corresponds to `Message` in Eventsourced. Sender references are written to the journal. A reply to senders must therefore be done via

a channel in order to avoid redundant replies during replay. Sender references of type `PromiseActorRef` are not journaled, they are `system.deadLetters` on replay.

- Supports *Snapshots*.
- *Identifiers* are of type `String`, have a default value and can be overridden by applications.
- Supports *Batch writes*.
- Supports stashing of messages.

Akka Persistence: `EventsourcedProcessor`

- Extension trait and pattern on top of `Processor` to support *Event sourcing*. Has no direct counterpart in Eventsourced. Can be considered as a replacement of two processors in Eventsourced where one processor processes commands and the other processes events that have been emitted by the command processor.

Channels

Eventsourced: `DefaultChannel`

- Prevents redundant delivery of messages to a destination (see usage example and default channel).
- Is associated with a single destination actor reference. A new incarnation of the destination is not automatically resolved by the channel. In this case a new channel must be created.
- Must be explicitly activated using methods `deliver` or `recover` defined on `EventsourcingExtension`.
- Must be activated **after** all processors have been activated. Depending on the recovery method, this is either done automatically or must be followed by the application (see non-blocking recovery). This is necessary for a network of processors and channels to recover consistently.
- Does not redeliver messages on missing or negative delivery confirmation.
- Cannot be used standalone.

Akka Persistence: `Channel`

- Prevents redundant delivery of messages to a destination (see *Channels*) i.e. serves the same primary purpose as in Eventsourced.
- Is not associated with a single destination. A destination can be specified with each `Deliver` request and is referred to by an actor path. A destination path is resolved to the current destination incarnation during delivery (via `actorSelection`).
- Must not be explicitly activated. Also, a network of processors and channels automatically recover consistently, even if they are distributed. This enhancement, together with improved processor recovery, makes recovery of complex Akka Persistence applications trivial. No special recovery procedures must be run by applications.
- Redelivers messages on missing delivery confirmation (see *Message re-delivery*). In contrast to Eventsourced, Akka Persistence doesn't distinguish between missing and negative confirmations. It only has a notion of missing confirmations using timeouts (which are closely related to negative confirmations as both trigger message redelivery).
- Can be used standalone.

Persistent channels

Eventsourced: `ReliableChannel`

- Provides `DefaultChannel` functionality plus persistence and recovery from sender JVM crashes (see ReliableChannel). Also provides message redelivery in case of missing or negative delivery confirmations.
- Delivers next message to a destination only if previous message has been successfully delivered (flow control is done by destination).

- Stops itself when the maximum number of redelivery attempts has been reached.
- Cannot reply on persistence.
- Can be used standalone.

Akka Persistence: `PersistentChannel`

- Provides `Channel` functionality plus persistence and recovery from sender JVM crashes (see *Persistent channels*). Same message redelivery features as `Channel`.
- Redelivers unconfirmed messages concurrently to newly delivered messages. Flow control is done by channel using a configurable minimum and maximum number of pending confirmations.
- Optionally notifies applications about messages for which the maximum number of delivery attempts has been reached (also offered by `Channel`).
- Can reply on persistence (= accept acknowledgement).
- Can be used standalone.

Views

Eventsourced:

- No direct support for views. Only way to maintain a view is to use a channel and a processor as destination.

Akka Persistence: `View`

- Receives the message stream written by a `Processor` or `EventsourcedProcessor` by reading it directly from the journal (see *Views*). Alternative to using channels. Useful in situations where actors shall receive a persistent message stream in correct order without duplicates.
- Can be used in combination with *Channels* for sending messages.
- Supports *Snapshots*.

Serializers

Eventsourced:

- Uses a protobuf serializer for serializing `Message` objects.
- Delegates to a configurable Akka serializer for serializing `Message` payloads.
- Delegates to a configurable, proprietary (stream) serializer for serializing snapshots.
- See Serialization.

Akka Persistence:

- Uses a protobuf serializer for serializing `Persistent` objects.
- Delegates to a configurable Akka serializer for serializing `Persistent` payloads.
- Delegates to a configurable Akka serializer for serializing snapshots.
- See *Custom serialization*.

Sequence numbers

Eventsourced:

- Generated on a per-journal basis.

Akka Persistence:

- Generated on a per-processor basis.

Storage plugins

Eventsourced:

- Plugin API: SynchronousWriteReplaySupport and AsynchronousWriteReplaySupport
- No separation between journal and snapshot storage plugins.
- All plugins pre-packaged with project (see journals and snapshot configuration)

Akka Persistence:

- Plugin API: `SyncWriteJournal`, `AsyncWriteJournal`, `AsyncRecovery`, `SnapshotStore` (see *Storage plugins*).
- Clear separation between journal and snapshot storage plugins.
- Limited number of *Pre-packaged plugins* (LevelDB journal and local snapshot store).
- Impressive list of community plugins.

10.2 Issue Tracking

Akka is using `Assembla` as its issue tracking system.

10.2.1 Browsing

Tickets

You can find the Akka tickets here

Roadmaps

The roadmap for each Akka milestone is here

10.2.2 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. *Please include the versions of Scala and Akka and relevant configuration files.*

10.3 Licenses

10.3.1 Akka License

```
This software is licensed under the Apache 2 license, quoted below.

Copyright 2009-2014 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.
```

10.3.2 Akka Committer License Agreement

All committers have signed this CLA. It can be signed online.

10.3.3 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): AkkaBuild.scala

10.4 Sponsors

10.4.1 Typesafe

Typesafe is the company behind the Akka Project, Scala Programming Language, Play Web Framework, Scala IDE, Simple Build Tool and many other open source projects. It also provides the Typesafe Stack, a full-featured development stack consisting of AKka, Play and Scala. Learn more at typesafe.com.

10.4.2 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

10.5 Project

10.5.1 Commercial Support

Commercial support is provided by Typesafe. Akka is part of the Typesafe Reactive Platform.

10.5.2 Mailing List

Akka User Google Group

Akka Developer Google Group

10.5.3 Downloads

http://akka.io/downloads

10.5.4 Source Code

Akka uses Git and is hosted at Github.

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

10.5.5 Releases Repository

All Akka releases are published via Sonatype to Maven Central, see search.maven.org

10.5.6 Snapshots Repository

Nightly builds are available in http://repo.akka.io/snapshots/ as both `SNAPSHOT` and timestamped versions.

For timestamped versions, pick a timestamp from http://repo.akka.io/snapshots/com/typesafe/akka/akka-actor_2.10/. All Akka modules that belong to the same build have the same timestamp.

sbt definition of snapshot repository

Make sure that you add the repository to the sbt resolvers:

```
resolvers += "Typesafe Snapshots" at "http://repo.akka.io/snapshots/"
```

Define the library dependencies with the timestamp as version. For example:

```
libraryDependencies += "com.typesafe.akka" % "akka-remote_2.10" %
  "2.1-20121016-001042"
```

maven definition of snapshot repository

Make sure that you add the repository to the maven repositories in pom.xml:

```
<repositories>
  <repository>
    <id>akka-snapshots</id>
    <name>Akka Snapshots</name>
    <url>http://repo.akka.io/snapshots/</url>
    <layout>default</layout>
  </repository>
</repositories>
```

Define the library dependencies with the timestamp as version. For example:

```
<dependencies>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-remote_2.10</artifactId>
    <version>2.1-20121016-001042</version>
  </dependency>
</dependencies>
```

CHAPTER

ELEVEN

ADDITIONAL INFORMATION

11.1 Frequently Asked Questions

11.1.1 Akka Project

Where does the name Akka come from?

It is the name of a beautiful Swedish mountain up in the northern part of Sweden called Laponia. The mountain is also sometimes called 'The Queen of Laponia'.

Akka is also the name of a goddess in the Sámi (the native Swedish population) mythology. She is the goddess that stands for all the beauty and good in the world. The mountain can be seen as the symbol of this goddess.

Also, the name AKKA is the a palindrome of letters A and K as in Actor Kernel.

Akka is also:

- the name of the goose that Nils traveled across Sweden on in The Wonderful Adventures of Nils by the Swedish writer Selma Lagerlöf.
- the Finnish word for 'nasty elderly woman' and the word for 'elder sister' in the Indian languages Tamil, Telugu, Kannada and Marathi.
- a font
- a town in Morocco
- a near-earth asteroid

11.1.2 Actors in General

sender()/getSender() disappears when I use Future in my Actor, why?

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 breaks 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.

Read more about it in the docs for *Actors and shared mutable state*.

Why OutOfMemoryError?

It can be many reasons for OutOfMemoryError. For example, in a pure push based system with message consumers that are potentially slower than corresponding message producers you must add some kind of message flow control. Otherwise messages will be queued in the consumers' mailboxes and thereby filling up the heap memory.

Some articles for inspiration:

- Balancing Workload across Nodes with Akka 2.
- Work Pulling Pattern to prevent mailbox overflow, throttle and distribute work

11.1.3 Actors Scala API

How can I get compile time errors for missing messages in *receive*?

One solution to help you get a compile time warning for not handling a message that you should be handling is to define your actors input and output messages implementing base traits, and then do a match that the will be checked for exhaustiveness.

Here is an example where the compiler will warn you that the match in receive isn't exhaustive:

```
object MyActor {
  // these are the messages we accept
  sealed abstract trait Message
  case class FooMessage(foo: String) extends Message
  case class BarMessage(bar: Int) extends Message

  // these are the replies we send
  sealed abstract trait Reply
  case class BazMessage(baz: String) extends Reply
}

class MyActor extends Actor {
  import MyActor._
  def receive = {
    case message: Message => message match {
      case BarMessage(bar) => sender ! BazMessage("Got " + bar)
      // warning here:
      // "match may not be exhaustive. It would fail on the following input: FooMessage(_)"
    }
  }
}
```

11.1.4 Remoting

I want to send to a remote system but it does not do anything

Make sure that you have remoting enabled on both ends: client and server. Both do need hostname and port configured, and you will need to know the port of the server; the client can use an automatic port in most cases (i.e. configure port zero). If both systems are running on the same network host, their ports must be different

If you still do not see anything, look at what the logging of remote life-cycle events tells you (normally logged at INFO level) or switch on *logging-remote-java* to see all sent and received messages (logged at DEBUG level).

Which options shall I enable when debugging remoting issues?

Have a look at the *remote-configuration-java*, the typical candidates are:

- *akka.remote.log-sent-messages*
- *akka.remote.log-received-messages*
- *akka.remote.log-remote-lifecycle-events* (this also includes deserialization errors)

What is the name of a remote actor?

When you want to send messages to an actor on a remote host, you need to know its *full path*, which is of the form:

```
akka.protocol://system@host:1234/user/my/actor/hierarchy/path
```

Observe all the parts you need here:

- **`protocol` is the protocol to be used to communicate with the remote system.** Most of the cases this is *tcp*.
- `system` is the remote system's name (must match exactly, case-sensitive!)
- `host` is the remote system's IP address or DNS name, and it must match that system's configuration (i.e. *akka.remote.netty.hostname*)
- `1234` is the port number on which the remote system is listening for connections and receiving messages
- `/user/my/actor/hierarchy/path` is the absolute path of the remote actor in the remote system's supervision hierarchy, including the system's guardian (i.e. `/user`, there are others e.g. `/system` which hosts loggers, `/temp` which keeps temporary actor refs used with *ask()*, */remote* which enables remote deployment, etc.); this matches how the actor prints its own `self` reference on the remote host, e.g. in log output.

Why are replies not received from a remote actor?

The most common reason is that the local system's name (i.e. the `system@host:1234` part in the answer above) is not reachable from the remote system's network location, e.g. because `host` was configured to be `0.0.0.0`, `localhost` or a NAT'ed IP address.

How reliable is the message delivery?

The general rule is **at-most-once delivery**, i.e. no guaranteed delivery. Stronger reliability can be built on top, and Akka provides tools to do so.

Read more in *Message Delivery Reliability*.

11.1.5 Debugging

How do I turn on debug logging?

To turn on debug logging in your actor system add the following to your configuration:

```
akka.loglevel = DEBUG
```

To enable different types of debug logging add the following to your configuration:

- `akka.actor.debug.receive` will log all messages sent to an actor if that actors *receive* method is a `LoggingReceive`
- `akka.actor.debug.autoreceive` will log all *special* messages like `Kill`, `PoisonPill` e.t.c. sent to all actors
- `akka.actor.debug.lifecycle` will log all actor lifecycle events of all actors

Read more about it in the docs for *logging-java* and *Tracing Actor Invocations*.

11.2 Books

- Akka in Action, by Raymond Roestenburg and Rob Bakker, Manning Publications Co., ISBN: 9781617291012, est fall 2013
- Akka Concurrency, by Derek Wyatt, artima developer, ISBN: 0981531660, est April 2013
- Akka Essentials, by Munish K. Gupta, PACKT Publishing, ISBN: 1849518289, October 2012

11.3 Other Language Bindings

11.3.1 JRuby

Read more here: https://github.com/iconara/mikka.

11.3.2 Groovy/Groovy++

Read more here: https://gist.github.com/620439.

11.3.3 Clojure

Read more here: http://blog.darevay.com/2011/06/clojure-and-akka-a-match-made-in/.

11.4 Akka in OSGi

11.4.1 Configuring the OSGi Framework

To use Akka in an OSGi environment, the `org.osgi.framework.bootdelegation` property must be set to always delegate the `sun.misc` package to the boot classloader instead of resolving it through the normal OSGi class space.

11.4.2 Activator

To bootstrap Akka inside an OSGi environment, you can use the `akka.osgi.ActorSystemActivator` class to conveniently set up the ActorSystem.

```
import akka.actor.{ Props, ActorSystem }
import org.osgi.framework.BundleContext
import akka.osgi.ActorSystemActivator

class Activator extends ActorSystemActivator {

  def configure(context: BundleContext, system: ActorSystem) {
    // optionally register the ActorSystem in the OSGi Service Registry
    registerService(context, system)

    val someActor = system.actorOf(Props[SomeActor], name = "someName")
    someActor ! SomeMessage
  }

}
```

The `ActorSystemActivator` creates the actor system with a class loader that finds resources (`reference.conf` files) and classes from the application bundle and all transitive dependencies.

The `ActorSystemActivator` class is included in the `akka-osgi` artifact:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-osgi_2.10</artifactId>
  <version>2.3.2</version>
</dependency>
```

11.4.3 Sample

A complete sample project is provided in akka-sample-osgi-dining-hakkers.

11.5 Incomplete List of HTTP Frameworks

11.5.1 Play

The Play framework is built using Akka, and is well suited for building both full web applications as well as REST services.

11.5.2 Spray

The Spray toolkit is built using Akka, and is a minimalistic HTTP/REST layer.

11.5.3 Akka Mist

If you are using Akka Mist (Akka's old HTTP/REST module) with Akka 1.x and wish to upgrade to 2.x there is now a port of Akka Mist to Akka 2.x. You can find it here.

11.5.4 Other Alternatives

There are a bunch of other alternatives for using Akka with HTTP/REST. You can find some of them among the Community Projects.