Community Experience Distilled

Akka Essentials

A practical, step-by-step guide to learn and build Akka's actor-based, distributed, concurrent, and scalable Java applications

Munish K. Gupta

[PACKT] PUBLISHING open source* community experience distilled

Akka Essentials

A practical, step-by-step guide to learn and build Akka's actor-based, distributed, concurrent, and scalable Java applications

Munish K. Gupta

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Akka Essentials

First published: October 2012

Production Reference: 1171012

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84951-828-4

www.packtpub.com

Cover Image by Eleanor Bennett (eleanor.ellieonline@gmail.com)

Credits

Author
Munish K. Gupta

Reviewers
Jonas Bonér
David Y. Ross
Domingo Suarez Torres

Acquisition Editor
Usha Iyer

Lead Technical Editor
Unnati Shah

Technical Editors
Mayur Hule
Devdutt Kulkarni
Ankita Shashi

Copy Editor
Insiya Morbiwala

Project Coordinator
Joel Goveya

Proofreader
Julie Jackson

Indexer
Monica Ajmera

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Munish K. Gupta is a Senior Architect working for Wipro Technologies. Based in Bangalore, India, his day-to-day work involves solution architecture for applications with stringent **non-functional requirements** (**NFRs**), Application Performance Engineering, and exploring the readiness of cutting-edge, open source technologies for enterprise adoption.

He advises enterprise customers to help them solve performance and scalability issues, and implement innovative differentiating solutions to achieve business and technology goals. He believes that technology is meant to enable the business, and technology by itself is not a means to an end.

He is very passionate about software programming and craftsmanship. He is always looking for patterns in solving problems, writing code, and making optimum use of tools and frameworks. He blogs about technology trends and Application Performance Engineering at `http://www.techspot.co.in` and about Akka at `http://www.akkaessentials.in`

Acknowledgement

Writing a book is never a single person's job. During the course of this journey, I have relied on many people, both directly and indirectly. I would like to thank the Akka community, from whom I have learned, and continue to learn, every day.

I would like to especially thank Jonas Bonér and David Ross, who reviewed and contributed many helpful suggestions and improvements to my drafts.

I am grateful to my editors, Usha Iyer, Unnati Shah, and Joel Goveya at Packt Publishing for their help in preparing this book. I would like to thank all my colleagues at Wipro Technologies, especially Hari Burle, Sridhar PV, and Aravind Ajad for all their support and encouragement. I have learned so much from each one of you, and for that I am grateful.

Last but not the least, I would like to thank my family. My wife Kompal, who has been a source of constant support throughout this journey. She single-handedly managed the kids and other chores around the house while I was working late nights and on weekends. She was the constant motivator who egged me on to go that extra mile whenever I felt that it was too big a task. I also want to thank my parents and brother Nitin, who provided the moral support throughout this journey. I love you all.

To my children, Dale and Sabal, I am sorry I couldn't be around with you as much as we all wanted, and many times had to get you away from the laptop. I love you very much.

About the Reviewers

Jonas Bonér is a geek, programmer, speaker, musician, writer, and Java champion. He is the CTO and co-founder of Typesafe, and is an active contributor to the open source community. Most notably, he founded the Akka project and the AspectWerkz AOP compiler (now AspectJ). You can know more about him at `http://jonasboner.com`.

David Y. Ross is a Scala enthusiast and Software Engineer at Klout, the social media startup that empowers its users to discover and be recognized for how they influence the world. As a member of Klout's platform team, David uses Scala and Akka to scale the Klout API to over a billion requests per day. Having previously worked on enterprise Java systems at a large tech company, he is constantly amazed by the productivity and elegance of Scala and Akka.

David attends Bay Area Scala meetups and has given a talk on Klout's use of Akka. He is a fan of Boston's sports teams and esoteric Jazz guitar players.

Domingo Suarez Torres is a Software Developer from Mexico City. He is always looking for tools that can make him a more productive developer. He likes to adopt frameworks that are in their early stages. In Mexico, he has been a pioneer in adopting several languages for the JVM, such as Groovy and Scala, programming languages that are used to build successful businesses. He has founded several user groups to spread the word about new technology.

In the professional field, he has worked for big companies as well as small ones in different sectors, such as financial, health, media, sales, and e-commerce. Currently, he is the CTO for a succesful e-commerce company in Mexico (clickOnero).

He has helped as a technical reviewer for other books, such as *Camel In Action, Claus Ibsen and Jonathan Anstey, Manning Publications* and *Making Java Groovy, Kenneth A. Kousen, Manning Publications.*

Table of Contents

Preface

Akka Essentials is meant as a guide for architects, solution providers, consultants, engineers, and anyone planning to design and implement a distributed, concurrent application based on Akka. It will refer to easy-to-explain concept examples, as they are likely to be the best teaching aids. It will explain the logic, code, and configurations needed to build a successful, distributed, concurrent application, as well as the reason behind those decisions.

This book covers the core concepts to design and create a distributed, concurrent application, but it is not meant to be a replacement for the official documentation guide for Akka published at Typesafe.

The driving force of Akka's Actor Model

The existing, Java-based concurrency model does not lend well to the underlying, hardware multiprocessor model. This leads to the Java application not being able to scale up and scale out, to handle the demands of a distributed, scalable, concurrent application.

The Akka framework has taken the "Actor Model" concept to build an event-driven, middleware framework that allows the building of concurrent, scalable, and distributed systems. Akka uses the Actor Model to raise the abstraction level that decouples the business logic from the low-level constructs of threads, locks, and non-blocking I/O.

The Akka framework provides the following features:

- **Concurrency**: The Akka Actor Model abstracts concurrency handling and allows the programmer to focus on the business logic
- **Scalability**: The Akka Actor Model's asynchronous message passing allows applications to scale up on multicore servers

- **Fault tolerance**: Akka borrows the concepts and techniques from Erlang to build the "Let It Crash", fault tolerance model
- **Event-driven architecture**: Akka provides an asynchronous messaging platform for building event-driven architectures
- **Transaction support**: Akka implements transactors that combine the actors and **software transactional memory** (**STM**) into transactional actors
- **Location transparency**: Akka provides a unified programming model for multicore and distributed computing needs
- **Scala/Java APIs**: Akka supports both Java and Scala APIs for building applications

The Akka framework is envisioned as a toolkit and runtime for building highly concurrent, distributed, and fault-tolerant, event-driven applications on the JVM.

What this book covers

Chapter 1, *Introduction to Akka*, covers the background on the evolution of the microprocessor, the current problems met in the building of concurrent applications, and the Actor Model. We will then jump into what Akka provides, and the high-level features of the Akka framework.

Chapter 2, *Starting with Akka*, covers the motions of the installation of the development environment and the writing of the first Akka application.

Chapter 3, *Actors*, covers the overview of the actors. The chapter covers the lifecycle of an actor, how to create actors, how to pass and process messages, and how to stop or kill the actor.

Chapter 4, *Typed Actors*, covers the overview of the typed actors. It also covers the lifecycle of a typed actor, how to create actors, how to pass and process messages, and how to stop or kill the actor.

Chapter 5, *Dispatchers and Routers*, covers dispatchers and their workings. The chapter covers the various types of dispatchers and their usage and configuration settings, and the different types of mailboxes and their usage and configuration. This chapter also covers routers, and their different types and usage.

Chapter 6, *Supervision and Monitoring*, covers fault tolerance, the lifecycle, supervision strategies, and linking strategies when writing large-scale, concurrent programs. The chapter covers the "Let It Crash" paradigm, and how it is managed in the Actor Model using the various supervision strategies.

Chapter 7, *Software Transactional Memory*, covers the various Akka constructs provided for the transactional concepts (begin/commit/rollback semantics). The chapter walks us through the basics of transaction management and explores the Akka constructs provided for STM—transactors and agents.

Chapter 8, *Deployment Ready*, covers the three, critical gating criteria that an application needs to pass in order to go into production. This chapter covers the unit and integration testing employed for the Akka application, how to manage environment-specific configuration, and the deployment strategies.

Chapter 9, *Remote Actors*, covers the requirements of a distributed computing environment and how Akka implements these. It also covers the various methods of creating remote actors, how object serialization happens in Akka, the various serializers provided by Akka, and how you can write your own serializers.

Chapter 10, *Management*, covers the monitoring capabilities provided by the Typesafe console—the Akka monitoring tool, various graphical dashboards, and real-time statistics. The chapter also covers the key JMX and REST interfaces.

Chapter 11, *Advanced Topics*, covers topics such as durable mailboxes, the integration of Akka with the play framework, and actor integration with ZeroMQ.

What you need for this book

The book is technical in nature, so the reader needs to have a basic understanding of the following:

- Java/Scala programming language
- Java's thread and concurrency model

Who this book is for

This book is aimed at developers and architects, who are building large distributed, concurrent, and scalable applications using Java/Scala. The book requires the reader to have a knowledge of Java/JEE concepts, but a knowledge of the Actor Model is not necessary.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows:"The `/lib` folder holds the `scala-library.jar` file."

A block of code is set as follows:

```
package akka.first.app.mapreduce.messages;
import java.util.List;
public final class MapData {
    private final List<WordCount> dataList;
    public List<WordCount> getDataList() {
        return dataList;
    }
    public MapData(List<WordCount> dataList) {
        this.dataList = dataList;
    }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
package akka.first.app.mapreduce.actors;
import akka.actor.ActorRef;
import akka.actor.UntypedActor;
import akka.first.app.mapreduce.messages.Result;

public class MasterActor extends UntypedActor {
    ActorRef mapActor;
    ActorRef reduceActor;
    ActorRef aggregateActor;
    @Override
    public void onReceive(Object message) throws Exception {

    }
}
```

Any command-line input or output is written as follows:

```
$ cd HttpActors
$ ls
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this:"Open Eclipse and go to **File** | **New** | **Project....**"

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Introduction to Akka

Akka is one of the most popular Actor Model frameworks that provide a complete toolkit and runtime for designing and building highly concurrent, distributed, and fault-tolerant, event-driven applications on the JVM. This chapter will walk you through the motivation and need for building an Akka toolkit.

As Java/Scala developers, we will see the usage of creating applications using the Akka Actor Model, which scales up and scales out seamlessly, and provides levels of concurrency, which is simply difficult to achieve with the standard Java libraries.

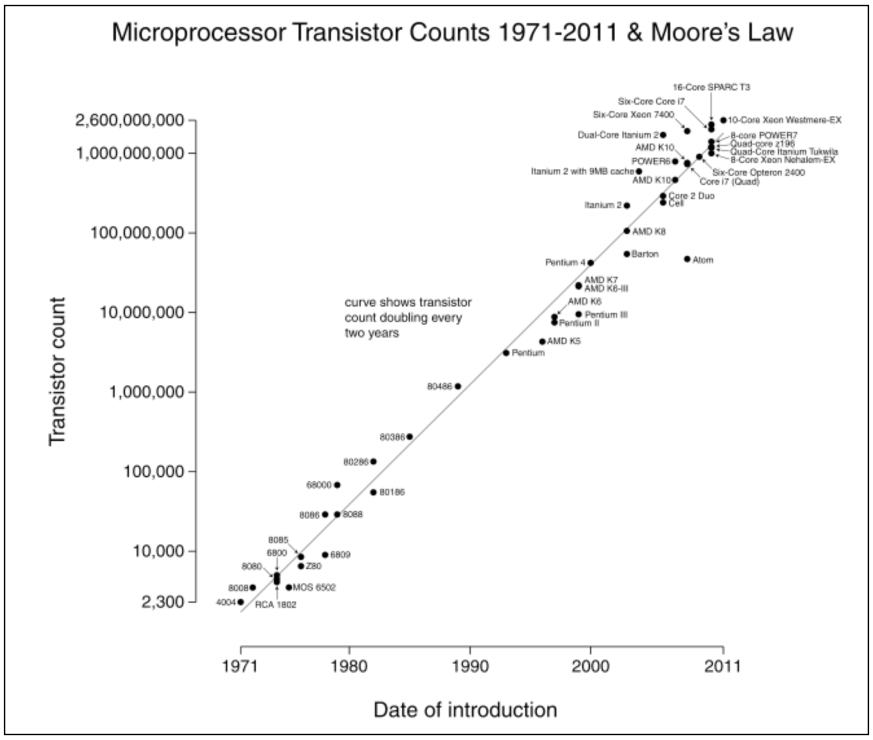
Background

Before we delve into what Akka is, let us take a step back to understand how the concept of concurrent programming has evolved in the application development world. The applications have always been tied to the underlying hardware resource capacity. The whole concept of building large, scalable, distributed applications needs to be looked at from the perspective of the underlying hardware resources where the application runs and the language support provided for concurrent programming.

Microprocessor evolution

The advancement of the microprocessor architecture meant the CPU kept becoming faster and faster with doubling of the transistors every 18 months (Moore's law). But soon, the chip design hit the physical limits in terms of how many transistors could be squeezed on to the **printed circuit board** (**PCB**). Subsequently, we moved to multicore processor architecture that has two or more identical processors or processor cores physically close to each other, sharing the underlying bus interface and the cache.

These microprocessors having two or more cores effectively increased the processor's performance by the same factor as the number of cores, limited only by the amount of serial code (Amdahl's law).



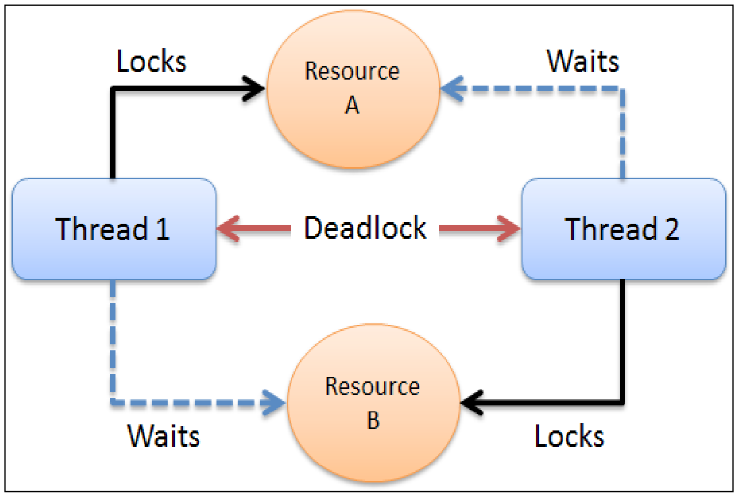
The preceding diagram from wiki, `http://en.wikipedia.org/wiki/Transistor_count` shows how the transistor count was doubled initially over the period of 18 months (following the Moore's Law) and how the multiprocessor architecture for consumer machines has evolved over the last 6-7 years.

Clock speeds are not increasing; processors are getting more parallel and not faster.

Concurrent systems

When writing large concurrent systems, the traditional model of shared state concurrency makes use of changing shared memory locations. The system uses multithreaded programming coupled with synchronization monitors to guard against potential deadlocks. The entire multithreading programming model is based on how to manage and control the concurrent access to the shared, mutable state.

Manipulating shared, mutable state via threads makes it hard at times to debug problems. Usage of locks may guarantee the correct behavior, but it is likely to lead to the effect of threads running into a deadlock problem, with each acquiring locks in a different order and waiting for each other, as shown in the following diagram:



Working with threads requires a much higher level of programming skills and it is very difficult to predict the behavior of the threads in a runtime environment.

Java provides shared memory threads with locks as the primary form of concurrency abstractions. However, shared memory threads are quite heavyweight and incur severe performance penalties from context-switching overheads.

A newer Java API around fork/join, based on work-stealing algorithms, makes the task easier, but it still takes a fair bit of expertise and tuning to write the application.

> Writing multithreaded applications that can take advantage of the underlying hardware is very error-prone and not easy to build.
>
> Scaling up Java programs is difficult; scaling out Java programs is even more difficult.

Container-based applications

Java Platform, Enterprise Edition (**JEE**) was introduced as a platform to develop and run distributed multitier Java applications. The entire multitier architecture is based on the concept of breaking down the application into specialized layers that process the smaller pieces of logic. These multitier applications are deployed in containers (called application servers) provided by vendors, such as IBM or Oracle, which host and provide the infrastructure to run the application. The application server is tuned to run the application and utilize the underlying hardware.

The container-based model allows the applications to be distributed across nodes and allows them to be scaled. The runtime model of the application servers has its own share of issues, as follows:

- In case of runtime failures, the entire request call fails. It is very difficult to retry any method execution or recovery from failures.
- The application scalability is tagged to the underlying application container settings. An application cannot make use of different threading models to account for different workloads within the same application.
- Using the container-based model to scale out the applications requires a large set of resources, and overheads of managing the application across the application server nodes are very high.

Container-based applications are bounded by the rules of the container's ability to scale up and scale out, resulting in suboptimal performance.

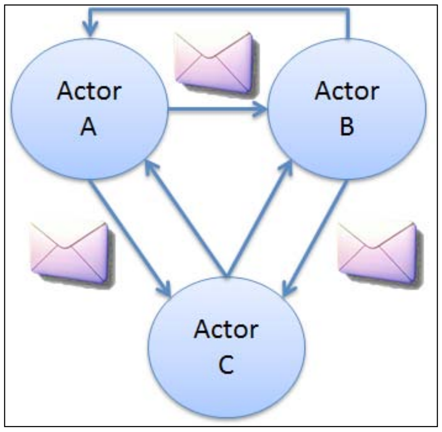
The JEE programming model of writing distributed applications is not the best fit for a scale-out application model.

Given that the processors are becoming more parallel, the applications are getting more distributed, and traditional JVM programming techniques are not helpful. So, there is a need for a different paradigm to solve the problem.

Actor Model

In 1973, Carl Hewitt, Peter Bishop, and Richard Steiger wrote a paper—*A Universal Modular ACTOR Formalism for Artificial Intelligence*, which introduced the concept of Actors. Subsequently, the Actor Model was implemented in the Erlang language by Joe Armstrong and Ericsson implemented the AXD 301 telecom switch that went onto achieve reliability of 99.9999999 percent (nine 9's).

The Actor Model takes a different approach to solving the problem of concurrency, by avoiding the issues caused by threads and locks. In the Actor Model, all objects are modeled as independent, computational entities that only respond to the messages received. There is no shared state between actors, as follows:



Actors change their state only when they receive a stimulus in the form of a message. So unlike the object-oriented world where the objects are executed sequentially, the actors execute concurrently.

The Actor Model is based on the following principles:

- The immutable messages are used to communicate between actors. Actors do not share state, and if any information is shared, it is done via message only. Actors control the access to the state and nobody else can access the state. This means there is no shared, mutable state.
- Each actor has a queue attached where the incoming messages are enqueued. Messages are picked from the queue and processed by the actor, one at a time. An actor can respond to the received message by sending immutable messages to other actors, creating a new set of actors, updating their own state, or designating the computational logic to be used when the next message arrives (behavior change).
- Messages are passed between actors asynchronously. It means that the sender does not wait for the message to be received and can go back to its execution immediately. Any actor can send a message to another actor with no guarantee on the sequence of the message arrival and execution.
- Communication between the sender and receiver is decoupled and asynchronous, allowing them to execute in different threads. By having invocation and execution in separate threads coupled with no shared state, allows actors to provide a concurrent and scalable model.

Akka framework

The Akka framework has taken the Actor Model concept to build an event-driven, middleware framework that allows building concurrent, scalable, distributed systems. Akka uses the Actor Model to raise the abstraction level that decouples the business logic from low-level constructs of threads, locks, and non-blocking I/O.

The Akka framework provides the following features:

- **Concurrency**: Akka Actor Model abstracts the concurrency handling and allows the programmer to focus on the business logic.
- **Scalability**: Akka Actor Model's asynchronous message passing allows applications to scale up on multicore servers.
- **Fault tolerance**: Akka borrows the concepts and techniques from Erlang to build a "Let It Crash" fault-tolerance model using supervisor hierarchies to allow applications to fail fast and recover from the failure as soon as possible.
- **Event-driven architecture**: Asynchronous messaging makes Akka a perfect platform for building event-driven architectures.
- **Transaction support**: Akka implements transactors that combine actors and **software transactional memory** (**STM**) into transactional actors. This allows composition of atomic message flows with automatic retry and rollback.
- **Location transparency**: Akka treats remote and local process actors the same, providing a unified programming model for multicore and distributed computing needs.
- **Scala/Java APIs**: Akka supports both Java and Scala APIs for building applications.

The Akka framework is envisaged as a toolkit and runtime for building highly concurrent, distributed, and fault-tolerant, event-driven applications on the JVM.

[Akka is open source and available under the Apache License, Version 2 at `http://akka.io`.]

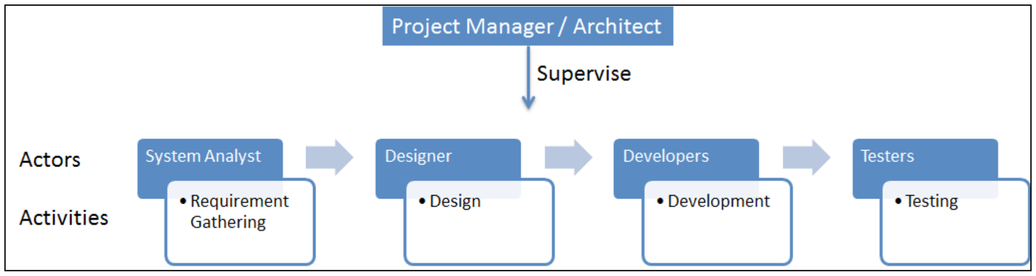
Akka was originally created by Jonas Bonér and is currently available as part of the open source Typesafe Stack.

Next, we will see all the key constructs provided by Akka that are used to build a concurrent, fault-tolerant, and scalable application.

Actor systems

Actor is an independent, concurrent computational entity that responds to messages. Before we jump into actor, we need to understand the role played by the actor in the overall scheme of things. Actor is the smallest unit in the grand scheme of things. Concurrent programs are split into separate entities that work on distinct subtasks. Each actor performs his quota of tasks (subtasks) and when all the actors have finished their individual subtasks, the bigger task gets completed.

Let's take an example of an IT project that needs to deliver a defined functionality to the business. The project is staffed with people who bring different skill sets to the table, mapped for the different phases of the project as follows:



The whole task of building something is divided into subtasks/activities that are handled by specialized actors adept in that subtask. The overall supervision is provided by another actor—project manager or architect.

In the preceding example, the project needs to exist and it should provide the structure for the various actors (project manager, architect, developer, and so on) to start playing their roles. In the absence of the project, the actor roles have no meaning and existence. In Akka world, the project is equivalent to the actor system.

[The **actor system** is the container that manages the actor behavior, lifecycle, hierarchy, and configuration among other things. The actor system provides the structure to manage the application.]

What is an actor?

Actor is modeled as the object that encapsulates state and behavior. All the messages intended for the actors are parked in a queue and actors process the messages from that queue.

Actors can change their state and behavior based on the message passed. This allows them to respond to changes in the messages coming in. An actor has the constituents that are listed in the following sections.

State

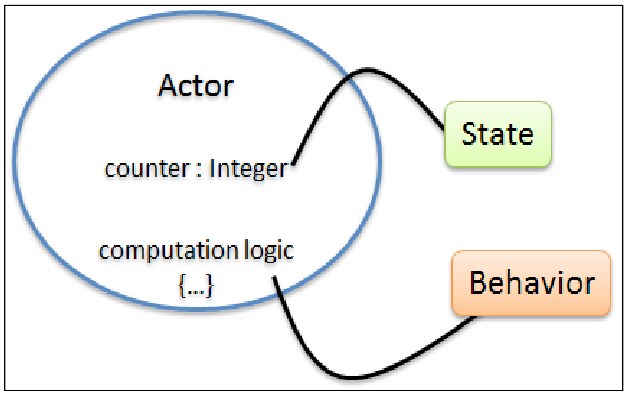
The actor objects hold instance variables that have certain state values or can be pure computational entities (stateless). These state values held by the actor instance variable define the state of the actor. The state can be characterized by counters, listeners, or references to resources or state machine. The actor state is changed only as a response to a message. The whole premise of the actor is to prevent the actor state getting corrupted or locked via concurrent access to the state variables.

Akka implements actors as a reactive, event-driven, lightweight thread that shields and protects the actor's state. Actors provide the concurrent access to the state allowing us to write programs without worrying about concurrency and locking issues.

When the actors fail and are restarted, the actors' state is reinitialized to make sure that the actors behave in a consistent manner with a consistent state.

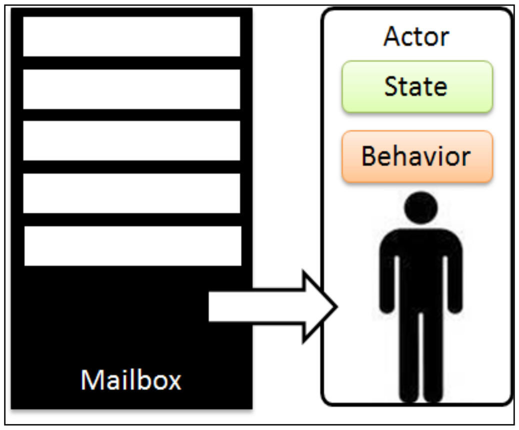
Behavior

Behavior is nothing but the computation logic that needs to be executed in response to the message received. The actor behavior might include changing the actor state. The actor behavior itself can undergo a change as a reaction to the message. It means the actor can swap the existing behavior with a new behavior when a certain message comes in. The actor defaults to the original behavior in case of a restart, when encountering a failure:



Mailbox

An actor responds to messages. The connection wire between the sender sending a message and the receiver actor receiving the message is called the **mailbox**. Every actor is attached to exactly one mailbox. When the message is sent to the actor, the message gets enqueued in its mailbox, from where the message is dequeued for processing by the receiving actor. The order of arrival of the messages in the queue is determined in runtime based on the time order of the send operation. Messages from one sender actor to another definite receiver actor will be enqueued in the same order as they are sent:



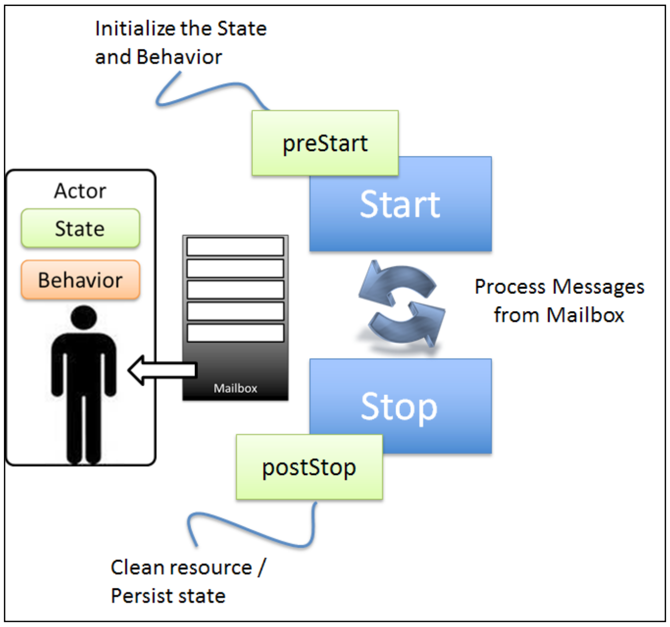
Akka provides multiple mailbox implementations. The mailboxes can be bounded or unbounded. A bounded mailbox limits the number of messages that can be queued in the mailbox, meaning it has a defined or fixed capacity for holding the messages.

At times, applications may want to prioritize a certain message over the other. To handle such cases, Akka provides a priority mailbox where the messages are enqueued based on the assigned priority. Akka does not allow scanning of the mailbox. Messages are processed in the same order as they are enqueued in the mailbox.

Akka makes use of dispatchers to pass the messages from the queue to the actors for processing. Akka supports different types of dispatchers. We will cover more about dispatchers and mailboxes in *Chapter 5, Dispatchers and Routers*.

Actor lifecycle

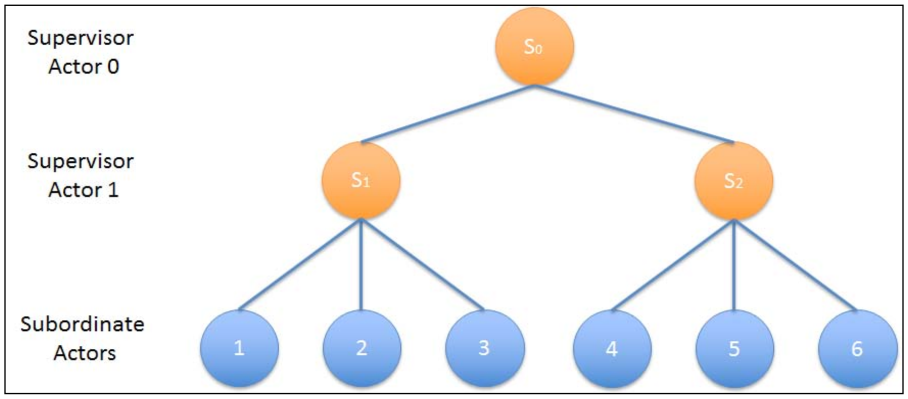
Every actor that is defined and created has an associated lifecycle. Akka provides hooks such as `preStart` that allow the actor's state and behavior to be initialized. When the actor is stopped, Akka disables the message queuing for the actor before `PostStop` is invoked. In the `postStop` hook, any persistence of the state or clean up of any hold-up resources can be done:



Further, Akka supports two types of actors—untyped actors and typed actors. We will cover untyped and typed actors in *Chapter 3*, *Actors*, and *Chapter 4*, *Typed Actors*, respectively.

Fault tolerance

Akka follows the premise of the actor hierarchy where we have specialized actors that are adept in handling or performing an activity. To manage these specialized actors, we have supervisor actors that coordinate and manage their lifecycle. As the complexity of the problem grows, the hierarchy also expands to manage the complexity. This allows the system to be as simple or as complex as required based on the tasks that need to be performed:



The whole idea is to break down the task into smaller tasks to the point where the task is granular and structured enough to be performed by one actor. Each actor knows which kind of message it will process and how he reacts in terms of failure. So, if the actor does not know how to handle a particular message or an abnormal runtime behavior, the actor asks its supervisor for help. The recursive actor hierarchy allows the problem to be propagated upwards to the point where it can be handled. Remember, every actor in Akka has one and only one supervisor.

This actor hierarchy forms the basis of the Akka's "Let It Crash" fault-tolerance model. Akka's fault-tolerance model is built using the actor hierarchy and supervisor model. We will cover more details about supervision in *Chapter 6, Supervision and Monitoring*.

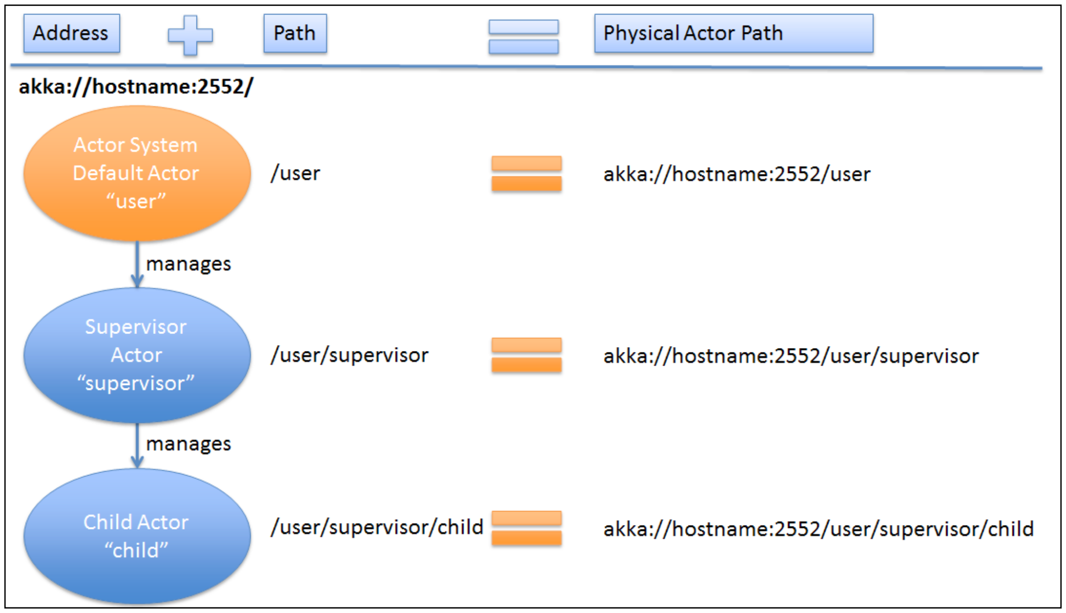
Location transparency

For a distributed application, all actor interactions need to be asynchronous and location transparent. Meaning, location of the actor (local or remote) has no impact on the application. Whether we are accessing an actor, or invoking or passing the message, everything remains the same.

To achieve this location transparency, the actors need to be identifiable and reachable. Under the hood, Akka uses configuration to indicate whether the actor is running locally or on a remote machine. Akka uses the actor hierarchy and combines it with the actor system address to make each actor identifiable and reachable.

Akka uses the same philosophy of the **World Wide Web** (**WWW**) to identify and locate resources on the Web. WWW makes use of the **uniform resource locator** (**URL**) to identify and locate resources on the Web. The URL consists of—`scheme://domain:port/path`, where `scheme` defines the protocol (HTTP or FTP), `domain` defines the server name or the IP address, `port` defines the port where the process listens for incoming requests, and `path` specifies the resource to be fetched.

Akka uses the similar URL convention to locate the actors. In case of an Akka application, the default values are `akka://hostname/` or `akka://hostname:2552/` depending upon whether the application uses remote actors or not, to identify the application. To identify the resource within the application, the actor hierarchy is used to identify the location of the actor:


Address
Path
Physical Actor Path
akka://hostname:2552/
Actor System Default Actor "user"
/user
akka://hostname:2552/user
manages
Supervisor Actor "supervisor"
/user/supervisor
akka://hostname:2552/user/supervisor
manages
Child Actor "child"
/user/supervisor/child
akka://hostname:2552/user/supervisor/child

The actor hierarchy allows the unique path to be created to reach any actor within the actor system. This unique path coupled with the address creates a unique address that identifies and locates an actor.

Within the application, each actor is accessed using an `ActorRef` class, which is based on the underlying actor path. `ActorRef` allows us to transparently access the actors without knowing their locations. Meaning, the location of the actor is transparent for the application. The location transparency allows you to build applications without worrying how the actors communicate underneath.

Akka treats remote and local process actors the same—all can be accessed by an address URL.

Transactors

To provide transaction capabilities to actors, Akka transactors combine actors with STM to form transactional actors. This allows actors to compose atomic message flows with automatic retry and rollback.

Working with threads and locks is hard and there is no guarantee that the application will not run into locking issues. To abstract the threading and locking hardships, STM, which is a concurrency control mechanism for managing access to shared memory in a concurrent environment, has gained a lot of acceptance.

STM is modeled on similar lines of database transaction handling. In the case of STM, the Java heap is the transactional data set with begin/commit and rollback constructs. As the objects hold the state in memory, the transaction only implements the characteristics—atomicity, consistency, and isolation.

For actors to implement a shared state model and provide a consistent, stable view of the state across the calling components, Akka transactors provide the way. Akka transactors combine the Actor Model and STM to provide the best of both worlds allowing you to write transactional, asynchronous, event-based message flow applications and gives you composed atomic arbitrary, deep message flows. We will cover transactors in more details in the *Chapter 7*, *Software Transactional Memory*.