Insider Akka

Konrad 'ktoso' Malawski

Table of Contents

Foreword

About Akka

Jonas / Roland?

Acknowlagements

I would like to thank the entire Akka community, for making it what it is today, and what it will grow to become in the future. You are an amazing, welcoming, and professional community, thank you.

I would like to thank the core Akka team for being the best team I've ever worked with, and can hardly imagine finding another team as skilled, well oiled and focused on improving the project as this one. Thanks: all names

About the author

Konrad 'ktoso' Malawski graduated from the AGH University of Science and Technology. He has worked in small startups, as well as large companies such as eBay, and has been part of the core Akka team at Typesafe/Lightbend since 2014 (just before Akka's 5th anniversary! [1: Akka's 5th anniversary infographic https://www.lightbend.com/akka-five-year-anniversary]). He has contributed to most, if not all, modules of Akka, maintaining a strong 2nd committer ranking across its main projects during that time. He has taken care of maintaining Akka HTTP while we were waiting to staff up its team as well as been the helping hand of the teams Tech Lead. He concurrently holds the 2nd most commits in either of these projects, and has lead to founding a number of satelite projects around the Akka "core" ecosystem.

He participated in the Reactive Streams initiative, by both working on the extensive Technology Compatibility Kit as well as contributing in discussions (on- and off-line), over the last years. He also authored a short, independent, report ("mini book") called *Why Reactive?* for O'Reilly in 2015.

Outside of the Akka and Reactive ecosystem he has been part of the PolishJUG and GeeCON conference's leadership group, which since 10 years gathers more than 1,200 developers each year in the heart of Krakow, Poland. He (co-)founded: KrakowScala, Lambda Lounge Krakow, The PaperCup London and was leading the Krakow GTUG during its initial years and while it's later been renamed GDG (Google Developers Group).

— Tokyo, July 2018

Chapter 1. Intro

I started writing this book out of a personal need to explain and share some of the internals that Akka has accumulated. While there is plenty books about Akka out there which go over the basics, as well as the reference documentation and guides, that we in the team continue to work on and improve. None of these resources are really going to take you from "Okey, I know how to use Akka", to "Now I fully understand how it works".

Such knowladge can be obtained by watching talks, reading Akka source-code and contributing to Akka itself. But still, it is difficult to extract rationale and process that led to some of the decisions in "one go". As all of core Akka development is made in the open [2: Yes, there exist commercial addons, after all Lightbend is a company targeting enterprises, and those have specific "enterprisy" needs that those addons address. They are just that however — addons — and none of them really "core" things are developed closed.] so you'd of course be able to find all this information by spending days reviewing pull requests, issues and design discussions in there, however that would be so time consuming, that rarely anyone does so.

By writing this book, I hope to share wth you knowladge that would be really hard to gather by someone who's entire all-around-the-clock isn't dedicated to building and improving Akka. This book is not intended to teach Akka, however if you had exposure to Akka's APIs, or concurrent and distributed programming in the past, you may find it a refreshing (and hopefully at least a bit enlightening) read about internals of one of the most popular library for building reactive distributed systems.

1.1. Versions referred to in this book

Details about the `ActorSystem` are referring to the `2.x` lifecycle of Akka. The `1.x` version of Akka was quite different, and we will not spend time on it in this book as the 2.x series is vastly superior, and been the de facto standard for more than 5 years by now. More specifically, whenever we'll talk about specific code, you may assume that we are talking about the Akka `2.5.x` series.

The same versions assumption, unless stated otherwise, hold for Akka Streams. Though we may refer to previous implementations of it as well, which will be highlighted explicitly when we do so.

For some discussion of an completely new `ActorSystem[T]` implementation for Akka Typed we will mostly refer to Roland Kuhn's attempt to implementing such system. However please note that this system implementation has not been merged and by the time a full "native" (rather than emulated on top of "Akka classic" (Untyped)) implementation of a Typed actor system would be merged, we could expect a number of changes, improvements and "hardening" be applied to it.

Whenever talking about Akka HTTP we will be talking about its `10.1.x` release cycle.

1.2. Resource recommendations to *learn* Akka

This book is not an introduction to Akka.

Far from it; it assumes knowladge (at least on a conceptual level) of Akka and its modules (such as Actors, Cluster, Streams, HTTP etc.), as well as concurrency and distributed systems knowladge at

least on a level that will allow you to track the implemenation decisions explained in this book.

As alternative books to start out with Akka, I strongly recommend the following books:

- If you are looking to start your journey with reactive systems and Akka, I would instead recommend to start out with Raymond Roestenburg's Akka in Action,
- If you are looking to understand reactive architectures and patterns, which happen to be nicely implementable using Akka, I recommend Roland Kuhn's (ex Akka Tech Lead) Reactive Design Patterns,
- Last but not least, do not be shy to refer to the official Reference Documentation. Which is available with examples in both Java and Scala (with an equal level of attention and quality).

While this book indeed is not going to teach you about using Akka, you are absolutely welcome to continue reading if you are curious about how one of the most popular reactive systems library (pretty much having coined the Reactive Manifesto and the term Reactive Systems) is implemented and want to learn about algorithms and data structures used by Akka internally, do bear in mind though that having some exposure to using Akka will be very helpful in understanding why some of the crazy things in the internals are the way they are - since they benefit the end users of the API.

Most, if not all, of the details outlined in this book should be mostly irrelevant to developers *using* Akka, however if you are an advanced user it may help you understand how to make better use of it (i.e. avoiding APIs which are "costly", and understanding why they are).

With that being said… welcome to *Insider Akka*.

Chapter 2. Guiding Principles

2.1. Islands of sanity, in a sea of parallelism

2.1.1. The "single-threaded illusion" of Actors

2.2. Immutability everywhere

One might wonder why immutability is a guiding principle in Akka which is all about Actors, which after all are **all about** encapsulating mutable state (that's the one thing they're best at!)

2.2.1. ByteString — immutable bytes

2.2.2. Immutability ⇒ Sanity

2.3. High-performance, but not at all costs

Actors are often referred to in the context of "high performance" messaging, and while this certainly can be true, it should also be put into perspective. It may be high performance "enough" for a person coming from an usual single-threaded and/or single-node processing paradigm, someone who's been working on high frequency trading would not call actors high performance, even though those systems share many similarities with Actors (processing messages from a queue in order, having addressable entities etc).

In the context of Akka it is important to keep in mind the "not at all costs" performance motto. Akka realises that for the vast majority of use cases its actors implementation is aboslutely "high performance enough" (esp. since Akka can also fan out the processing across many nodes), and does not aim to serve the 10% cases where Actors and the GC of short lived objects they cause is not a good fit.

A specific example could be discussing if Akka should or should not carry messages between actors in so-called Envelopes.

Chapter 3. Actor internals

3.1. Why the historical type-unsafety?

3.2. Protocol Design

3.2.1. Typed Protocol Design

While we have not yet talked much about Akka Typed (which is an type-safe API on to

3.3. The dungeon — what makes the Actors run

3.3.1. Making of a Cyborg: Starting Actors

As well versed Akka developer you likely know that top-level Actors are more "expensive" than child Actors. Have you ever wondered why that is? In this section we'll explore how Actors are created, and this will also explain why and how top level actors are more difficult for the infrastructure to start than child actors.

Hint: it relates to "from the outside of system" initiated starting of them vs. from "the inside of the system/actor".

Note

> You are highly encouraged to loop the fantastic track "Making of a Cyborg" by Kenji Kawai, from the 1995 movie 'Ghost in the Shell' while reading this chapter. Here is a youtube search for it: `https://www.youtube.com/results?search_query=kenji+kawai+making+of+a+cyborg`

Guardian Actors

The Untyped `ActorSystem` has 2 main guardians, they are maintained by Akka itself.

The `/user` guardian is where we "attach" all actors started by users, e.g. by `system.actorOf`, and the `/system` guardian is where we "attach" all actors started using the special `system: ExtendedActorSystem.systemActorOf` actors. They don't really differ in behavior so much; The user guardians supervision strategy can be configured by config by users, but the system one not. During shutting down the system, the system actors are shut down LAST (CHECK THIS), after all user actors are stopped. That's about it.

system.actorOf

This is how we start (or "spawn") "top-level" actors. Note that we are not IN the `/user` guardian, we are after all "somewhere on the side" yet we need to attach the new child to the `/user` guardian. This means we have to send a message to the user guardian asynchronously for it to perform the spawning, yet at the same time we do not want to block while we do this.

```
val ref = system.actorOf(act, "major")
  // sync: notice we got our ActorRef back immediately (!)
ref ! "Reporting for duty!" // we immediately send
  // async: dear `/user`, please attach child `major`
  // async: `/user`: reserved name "major"
  // async: `/user`: initializing `/user/major`
  // async: `/user`: repoint (!) `ref` to `/user/major`!
```

The main principle here is that a) we do not want to block until the actor is fully initialized in `actorOf`, and b) we are bound to perform some async work during initialization since the `/user` guardian is an actor, so the way to communicate with it is via async operations.

Initial steps of starting a top level actor are "reserving" the name in the user guardians `ChildrenContainer`. This is a special datastructure in which we keep all the child actors. The first thing to do **synchronously (!)** is to reserve the new name, in order to be able to predictably be able to handle attempts of starting actors with the same name from different threads, like this:

```
// Thread 1:
  system.actorOf(..., name = "major")

// Thread 2, concurrently:
  system.actorOf(..., name = "major")

// one of those calls is guaranteed to throw
```

The **guarantee** given by the actor system is that always exactly one of the calls will succeed and the other will fail with an `InvalidActorNameException(s"actor name [$name] is not unique!")`. This is why spawning

Actors are identified not only by path but also UUID.

```
val childPath = new ChildActorPath(cell.self.path, name, ActorCell.newUid())
cell.provider.actorOf(cell.systemImpl, props, cell.self, childPath,
            systemService = systemService, deploy = None, lookupDeploy = true, async =
async)
```

This brings us to another abstraction inside the core of Akka: `ActorRefProvider`. Currently two implementations of it exist: the `LocalActorRefProvider` and the `RemoteActorRefProvider`. The remote one is only needed for deploying actors on remote nodes, which is a terrible hack and an idea the team would like to get rid of. TODO CHECK THIS.

We will focus only on the `LocalActorRefProvider` in this section. Not only does it actually create the ActorRef that we'll be able to hand out, it also does look at configuration of the system, since depending on the actor's path there may be configuration which makes it use some specific dispatcher. This works in tandem with settings passed in using `Props`. One can configure actor props using the `Props` object or by configuring it in `application.conf` in the deployment section of it.

Finally, the `LocalActorRefProvider` applies all config and creates our ref:

```
if (async) new RepointableActorRef(system, props2, dispatcher, mailboxType,
supervisor, path).initialize(async)
else new LocalActorRef(system, props2, dispatcher, mailboxType, supervisor, path)
```

Notice that when using the async init mode (we are using this mode for top-level actors, so we will focus on that), we create a special `RepointableActorRef`. This is another reason why top-level refs are more "expensive", they initialize asynchronously and while that init is in flight, they don't actually have a mailbox ready yet (!). This means the ref itself is buffering messages until the mailbox is initialized and the repointable ref is "pointed at" the ready and live cell with its mailbox.

To understand this a bit more, let's look at the `def !(msg)(sender): Unit` implementation of the `RepointableActorRef`:

```
underlying.sendMessage() // TODO
```

Internally this ends up executing in the so-called `UnstartedCell`. The `ActorCell` in general is the "heart" of the Actor, yet here it is not yet started... so it has to keep acting as if it works before it is started and ready for scheduling... It also has no mailbox ready yet, which means it has to internally implement one as "best effort" which will be used for *the few milliseconds until the mailbox initialization completes and the cell is started*. This brings us to one of the very few locks in the entirety of Akka, in the implementation of `sendMessage` of the `UnstartedCell` we see:

```
private[akka] class UnstartedCell(
  val systemImpl: ActorSystemImpl,
  val self:       RepointableActorRef,
  val props:      Props,
  val supervisor: InternalActorRef) extends Cell {

  /*
   * This lock protects all accesses to this cell's queues. It also ensures
   * safe switching to the started ActorCell.
   */
  private[this] final val lock = new ReentrantLock

  // use Envelope to keep on-send checks in the same place ACCESS MUST BE PROTECTED BY
THE LOCK
  private[this] final val queue = new JLinkedList[Envelope]()

  // ACCESS MUST BE PROTECTED BY THE LOCK
  private[this] var sysmsgQueue: LatestFirstSystemMessageList = SystemMessageList.LNil
```

```
  def sendMessage(msg: Envelope): Unit = {
    if (lock.tryLock(timeout.length, timeout.unit)) {
      try {
        val cell = self.underlying
        if (cellIsReady(cell)) {
          cell.sendMessage(msg)
        } else if (!queue.offer(msg)) {
          system.eventStream.publish(Warning(self.path.toString, getClass, "dropping
message of type " + msg.message.getClass + " due to enqueue failure"))
          system.deadLetters.tell(DeadLetter(msg.message, msg.sender, self),
msg.sender)
        } else if (Mailbox.debug) println(s"$self temp queueing ${msg.message} from
${msg.sender}")
      } finally lock.unlock()
    } else {
      system.eventStream.publish(Warning(self.path.toString, getClass, "dropping
message of type" + msg.message.getClass + " due to lock timeout"))
      system.deadLetters.tell(DeadLetter(msg.message, msg.sender, self), msg.sender)
    }
  }
```

context.actorOf

This method allows spawning child actors when you "are" an actor. This spawns them "under" yourself and also makes you their supervisor. In other words, in Akka (Untyped) the one who makes child actors, is automatically responsible (supervising) for them. This is unlike Erlang, where supervision is completely separate. Later on we'll discuss `watch` as well, but that is different than supervision.

Note also that since we are now attaching new children to our own `ChildrenContainer` which is held by the actor we're in, it means that we need to additional synchronization during startup of the new actor (!).

3.4. Differences between System Actors and normal ones

System actors ignore the deployment section; they may only be local.

Remote deployment is a messa anyway…

3.5. Do the Guardian actors have a parent?

An interesting question is… "Do the Guardian actors have a parent?"

After all, they are actors as well, and as we know, in Akka "every actor has a parent".

Indeed, the guardians do have a parent Actor. It is `theOneWhoWalksTheBubblesOfSpaceTime` [3: theOneWhoWalksTheBubblesOfSpaceTime - https://github.com/akka/akka/blob/

e6633f17fac9b2fe1100af73b18add3ac24ad0df/akka-actor/src/main/scala/akka/actor/ActorRefProvider.scala#L519-L554].

The unstarted cell waits for the parent to get the `Supervise` message and only then the real cell with user code is started. This is because the parent needs to know about the child in order to be able to apply the supervision things.

3.6. Restarting Actors

This is why we can do `newCell` — we restart it "in place" while the mailbox remains untouched.

3.7. Sending (Local) Messages

3.7.1. Sending user messages

All messages which are send by users in an Akka application between actors are referred to as "user messages".

The delivery guarantee provided for such messages is *at-most-once*, which holds true for either the remote or local case. After all, evne in a local setting the JVM may crash at any time (well, in theory at least!), thus the "at most once" part holding true even in local applications. In practice though one often assumes that messages will be delivered unless we are thinking about critical for correctness things, where we can apply at-least-once delivery to messages, which is slower and most costly however can achieve this by also applying persistence before.

note

Akka provides *at-most-once delivery* guarantees for *plain actor messaging*, and on top of this is able to implement *at-least-once delivery* when it is required. One might be suprised how often at-most once is most of the time enough for most applications.

3.7.2. Sending system messages

System messages are a special, they are (best effort) guaranteed to be delivered. This is because many invariants of your system depend on those messages. This means that they are internally buffered and re-delivered in thr distributed setting, and in the local setting it means that even if you pick a mailbox that is bounded, system messages actually use a separate queue (which we'll investigate in a second) as they must not bbe dropped on the floor to keep correctness of core akka things, such as death watch, lifecycle events and supervision.

As a short reminder, system messages are for example `Terminated(ref)` which you may have seen (as it is sent as effect of a watched actor terminating), or deeply internal messages like `Supervise` which is sent from asynchronously spawned child actor to it's parent so the parent can become its supervisor (this is during child actor starting).

Unlike user messages which are any kind of message that users send within an Akka application system messages. Let's first see what the `SystemMessage` trait is implemented as:

```
/**
 * INTERNAL API
 * ...
 * <b>NEVER SEND THE SAME SYSTEM MESSAGE OBJECT TO TWO ACTORS</b>
 */
private[akka] sealed trait SystemMessage extends PossiblyHarmful with Serializable {
  // Next fields are only modifiable via the SystemMessageList value class
  @transient
  private[sysmsg] var next: SystemMessage = _

  // ... queue operations ...
}
```

You may be surprised to see the "never send the same system message" information here. This is because system messages ARE the message queue (!). This is to save space in the Actor Mailbox so we don't have two complicated queues but only one, and the simplified one for system messages which is simply a single linked list of the messages.

This allows us to implement the system message queue in the actor mailbox as:

```
 * INTERNAL API
 */
private[akka] abstract class Mailbox(val messageQueue: MessageQueue)
  extends ForkJoinTask[Unit] with SystemMessageQueue with Runnable {

  // ...

  @volatile
  protected var _statusDoNotCallMeDirectly: Status = _ //0 by default

  @volatile
  protected var _systemQueueDoNotCallMeDirectly: SystemMessage = _ //null by default
```

3.8. Sending Messages to distant nodes (Remoting)

3.8.1. Sending user messages

3.8.2. Sending system messages

3.9. Processing messages

First system messages are processed — all of them.

Next user messages are processed, until the `throughput` limit. We also experimented with a time limit, but in reality this was never used — calculating time costs after all.

3.10. Death Watch: `watch`-ing Actors for lifecycle events

Death watch is modeled after Erlang's `monitor` [4: Erlang's monitor function: http://erlang.org/doc/reference_manual/processes.html#monitors] API, in which processes bind their lifecycles to one another.

Lifecycle monitoring of other Actors is quite an important part of Actor systems, since thanks to this one can easily tear down entire groups of actors that "only work if others work as well". The concept itself is also present in Erlang, where it is called `monitor/2`

3.10.1. Dying together, with `DeathPactException`

When you `context.watch(ref)` an actor and *do not* handle the resulting `Terminated(ref)` messages, this results in an `DeathPactException` being thrown by the watching actor.

In other words, if you only watch, and don't implement logic that handles termination, the default behavior is for the watcher to kill itself if the watched actor terminates. This is referred to as the "death pact".

A good way to visualize death pact's default behavior is Shakespeare's play Romeo and Juliet, in which during the plays finale Romeo finds Juliet "dead", so he decides to kill himself as he can not imagine continuing life without her. Shortly after, once Juliet wakes up and sees Romeo really dead, she decides to kill herself *for real* this time. In the end, both actors are dead.

Note that, unlike in Shakespeare's play, the `Terminated()` message will never fire prematurely. Once it has been sent we know for certain that the terminated actor is indeed dead. This is somewhat more interesting in clusterd environment, where `Terminated` can fire when an entire node is marked as `Down` — since the actor itself may not have actually terminated… however, since the entire node is declared as `Down`, we *know* that we will never receive a message from it ever again.

In clustering, this effect has the simple mnemonic of: "We do not talk to zombies."

One more note about the Romeo & Juliet example: You may have noticed that the terminating oneself is somewhat "mutual" in the play. Once Romeo notices Juliet dead, he kills himself, and likewise once Juliet notices Romeo dead, she kills herself. In Akka terms, this could be seen as two actors, which watch *eachother*, which means that whichever actor terminates first, the other one will terminate itself in response to this lifecycle event.

Chapter 4. Remoting internals

4.1. Streaming remoting pipeline

4.1.1. Parallelizing multiple "lanes"

4.2. Message Metadata Compression Table Protocol

4.3. Swappable transports Aeron (UDP) / Akka IO (TCP+TLS)

Chapter 5. APIs that guide and protect

5.1. Streams' GraphStage — design analysis

5.2. Typed Actors — design analysis

Chapter 6. Stream API levels of power and abstraction

6.1. Operators, the "safe heaven"

6.2. Graphs, for arbitrary shapes

6.3. Stages, for arbitrary logic protected from concurrency issues

6.4. Reactive Streams SPI, all concurrency issues and hell set lose

Chapter 7. The depths of Stream Materialization

7.1. Materialization, or "Traversing the blueprints"

https://github.com/akka/akka/blob/master/akka-stream/src/main/scala/akka/stream/impl/package.scala

7.2. Examining the Engine running it all: ActorInterpreter

Chapter 8. Scheduling (Timers)

8.1. No wall-clock time in a Cluster

8.2. The LightArrayRevolverScheduler

8.3. The un-deniable existence of wall-clock time

8.3.1. Timers and the Scheduler

Chapter 9. Cluster Failure-detection

9.1. Transport failures and Cluster-level semantics

Quarantines — transport

Cluster-level unreachability

9.2. Fail-over scenarios exposed

https://support.lightbend.com/agent/case/10623 and others by him

Chapter 10. Networks and Types

10.1. A Receptionist for thy Actors

Imagine you're on a work trip in some foreign country, you're jet lagged, batteries in all your electronic devices have drained, and yet you somehow manage to arrive at the hotel you booked. You want to meet your friend who's (maybe), already checked-in at the same hotel (though you're not quite sure, your flights were arriving concurrently after all, and traffic was terrible on the way to the hotel!) You do want to meet your friend, however you're not sure how to find them, you don't know the room number (the "Address") after all.

You wouldn't (I hope, at least) barge into random people's hotel rooms, trying to find your friend's room, by vaguely remembering something they told you before ("*... was it that they only stay in rooms that end with a 7?*"). You'd ask at the *reception*, if your friend has arrived, and if they could either call them down for you or tell you their room number.

Similarily, one should not rely on randomly guessing an `Actor` 's address! This, in fact, describes the difference between using `.actorSelection(String)` and a proper `Receptionist` to locate an `Actor`. Far too often, perhaps tempted by scars of the past of using service locators which return instances if only you knew the right identifier, I've seen developers resolve to exclusively using selection to locate "well known actors". This however is increadibly brittle, as the **path** of an Actor really should be it's own thing, you should not try to remember and reconstruct by concating strings and guessing where the Actor may live — similarily how looking for our friends hotel room by guessing, is probably a bad idea, you may, after all, end up in someone else's room, and cause everyone quite a hassle.

In Actor systems this is about not integrating over the paths, which could be dynamically allocated, or even changing since there is a rollout ongoing, and in the new version of the software that specific actor has moved somewhere.

By using an Receptionist, we share a piece of information, the `ServiceKey[M]`, by which we can ask the receptionist if the Actor has already, checked-in or not yet. Notice that this also deals nicer with typing, as here the Actor, while checking in, lets the receptionist know which protocol (messages of type `M`) it is able to speak. This way, even if there's two actors called "bob", but only one of them speaks the protocol you want to communicate with, you'd be able to get the right `ActorRef[M]` from the Receptionist.

Chapter 11. Type-safe HTTP

11.1. Akka HTTP's Domain Model

A fully type-safe representation of the HTTP domain is difficult to pull off. HTTP is notorious for its corner cases and ambiguities.

JSON example

11.2. Spraying the Routes — a note on type-safe routing DSLs

Chapter 12. The people crazy enough to think they can change the world…

In this section we will explore random crazy ideas that may change how actor systems look like in "magical dream land" implementations.

https://www.goodreads.com/author/show/14907567.Rob_Siltanen

12.1. Async/await within receive blocks