Akka framework

Martin Thurau
University of Lübeck
Email: martin.thurau@gmail.com

***Abstract*—Akka is a toolkit for building concurrent, distributed application on the JVM using the actor model. It provides the developer with a well defined API to develop large concurrent systems and allow for easy scaling out of a single machine.**

We will discuss the underlying actor model of actor and show how Akka implements it, additionally highlighting some of the other features and tools Akka provides.

I. Introduction

Building concurrent applications seems to be hard even with modern tools. On multi core machines one has to carefully places locks around shared data-structures while avoiding deadlocks and race conditions which lead to hard to reproduce and debug failures. And even if you manage all the problems that arise and your system grows, you will eventually reach a point where you need to scale out of a single machine towards many of them. If you haven't foreseen the need to scale to a multi-machine setup you will most likely end up rewriting large parts of your application. Akka tries to solve some of these problems by building on the actor model to create an abstract model of computation that nicely fits on todays many-core systems while additionally providing a seamless way of integration multiple machines to a complex system of computation.

To provide a better understanding of Akka, we will first introduce the actor model in general and will afterwards explain how Akka implements this model. In the next section we will outline some of the additional features Akka provides that are not part of the actor model itself. After that we will compare Akka to a few other actor implementation a highlight differences.

II. Actor model

Akka is based on the Actor model which was introduced in 1973 [1]. The actor model is, like a Turing machine, a model to describe and reason about a computational process in an theoretical environment. As the Actor model is based on small, independent and concurrent primitives (called "actors") it turned out that it is an useful basis for implementing concurrent systems.

As *everything is an object* in object-oriented programming in an actor based system *everything is an actor*. An actor is an entity of computation that communicates with other actors on the base of messages[2] that are send between the actors. Essentially and actor is the encapsulation of some behavior and (not necessarily) some internal state. As stated these actors exchange message between each other. Whenever an actor receives a message it can act on it using a very limited set of operations.

A. Actions of an actor

If a actor receives a message it can do any of the following things:

- send a (finite) number of messages to other actors
- create a (finite) number of new actors
- alter the own behavior for the next received message (i.e. change some internal states that are used the next time a message if received)

This list is finite and no other actions may be taken. Note that their are no restrictions on order and concurrency, so an actor may concurrently create new actors and send messages to them. The only limitations is that the process must be finite, so the actor *must* eventually finish his work.

B. Message passing

These messages the actors send are exchanged through an non specified medium and the communication my be direct, which means that the messages are not "buffered" in any way. In order to exchange messages it is obvious that their needs to be a way

of specifying the receiver of a message. The Actor model places no special restrictions on the kind of addresses other that they exist so a message can be appropriately labeled. Addresses of other actors can be obtained either from a received message (i.e. if the receiver wants to reply to the sender) or from creating a actor (i.e. to delegate work to it). Sending message is an asynchronous process. A sender simply sends a message, without any prior handshake or other protocol in before. Additionally it is not required that the sender sends something back in response and when it does the response message is not part of some special request-reply semantic but a separate message on its own. The order of message delivery is undefined, so if actor A sends two messages M_1 and afterwards M_2 it is not guaranteed that M_1 is received before M_2. While the ordering is not strictly required by the actor model, most implementations (like Akka) choose to implement it to some point.

C. Concurrency and scalability

Since actors may carry out their actions in parallel and messages may be reordered, actor based systems are entirely concurrent. Each actors lives totally separated from all other actors and only messages, which are conceptually different from actors, are passed between them. So two actors may be executed by a single thread on a single chip that alternates execution between the two of them or they may be executed parallel within two data centers on both ends of the world. The actors themselves can and should not know, they simply exchange messages. This property makes it easy to scale actor based systems out of a single core to millions of them in multiple data centers. In an ideal case you simply change the configuration of the system and you are done without any need to rewrite code.

D. No shared data

Since actors only communicate by messages they share no common data. Additionally all messages are worked on independently and the internal state of an actor is only changed when the processing of a single message is done. So when an actors processes a single message it can make the very convenient assumption that the world does not change until the message is processed. This makes programming actor a linear process with no need to use locking, semaphores or mutexes while accessing internal data. Unfortunately that comes with a trade off in therms of memory consumption: Since actors may not share data it is necessarily two duplicate information be copying them (this problem can be alleviated by allowing sharing of immutable data between actors which is, what some actor systems are trying to do).

III. The actor model in Akka

The actor model is implemented in Akka with respect to the definition above. However, their are certain parts of this implementation that have subtle differences in them. Before we take a look at them, lets first start with a simple code example on how an actor actually looks if you implement it with Akka. After all, that concept is abstract enough so an example will certainly help to build a mental model of it. Listing 1 shows an example of "Hello World" using a single actor. The example is written in Scala since the Scala API for Akka produces a smaller code overhead. We don't assume the reader is familiar with Scala but think the examples are even understandable with no prior knowledge of Scala.

All basic classes of Akka actors are available with an single import in line 1. The actual Actor is implemented in the lines 3 to 8. We only need to extend the class `Actor` and override the `receive` method. This method is called each time a message is received by the actor. The method does pattern matching[3] on the received message to decide what to do. In this case, whenever a message is received that contains a single string a message is printed to the console.

The `Main` object in lines 10-19 is what would be the code within the `main` method in a Java application and is executed when the application starts. We first create an `ActorSystem` in line 12 which is responsible for creating and running the actors. The `ActorSystem` is then asked to create a new actor of the class `Greeter` in line 14. In line 16 we send the `Greeter` actor a message (in this case a simple string). Notice that the call in line 14 will not block and return before the actor is actually created (which is done asynchronously in the background) but we can send messages to is

```scala
import akka.actor._

class Greeter extends Actor {
  def receive = {
    // if we receive a String we print a greeting
    case name:String => println(s"Hello $name")
  }
}

object Main extends App {
  // create the actor system
  val system = ActorSystem("GreeterSystem")
  // create an actor
  val greeter = system.actorOf(Props[Greeter], name="greeter")
  // send greeter a message
  greeter ! "World"
  //shutdown the actor system
  system.shutdown()
}
```

Listing 1. "Hello World in Akka"

immediately - they will be processes whenever the actor is ready.

A. Message passing

An Actors in Akka is modeled as single object of functionality. Each actor has an event driven message inbox (called mailbox) that holds all incoming message until they are processed. Within the actor their is an function that takes out one message and acts accordingly. Each actor is identified by a so called `ActorRef`. This references are created whenever one actor creates another actor and are additionally added automatically to each message that es send between two actors identifying the sender of the message.

Messages in Akka can be from any type. You should however be aware that these object will *not* be copied when to actors run on the same physical system so it is possible to create a situation where two actors share some kind of data.

As mentioned in II-B Akka chooses to implement some message ordering guarantees for certain kinds of messages. Specifically Akka provides a guaranteed ordering for any two pairs of actors. So if actor A sends two messages M_1 and M_2 to actor B, M_1 will be received before M_2. However, this only is the case if the receivers uses a simple FIFO mailbox and not one of the other mailboxes that allow prioritizing certain messages so this featured should be used carefully[4].

B. Shared data

Since Akka runs on top of the JVM and is implemented as a simply library it is not possible for Akka to strictly enforce a true data separation between actors (other then inspecting every single message which would certainly hurt performance). If an actor creates a mutable data structure and sends a reference to this structure to another actor (using a message that contains the reference to the structure) and both actors run on the same VM these two actors will share the same underlying data structure, thus breaking the encapsulation of the actors. This will most certainly create hard to reproduce bugs (since all actors run concurrent and the process could likely become indeterministic) so this is strongly discouraged. Additionally, this problem goes away if two communicating actors don't run within the same JVM, since the data in the messages is then actually serialized and send over to the other JVM.

C. How actors are run concurrently

Actors in Akka run event-driven. Within the system there a certain number of threads that switch between executing the different actors. As noted, each actor has an mailbox, that holds incoming messages. When an actor has a non empty mailbox and it is executed by a thread the message is taken out of the mailbox, processed by the actors code,

maybe modifying some internal state, creating actors, sending messages, etc. and finally the execution finishes (remember that the actor model requires that an actor will finish after a finite amount of time). After this, the executing thread ends the execution of this particular actor and activates another one with a non empty mailbox (which may in fact be the same as before).

D. Scalability

The way Akka processes the actors as describe in the last section is only the default case where Akka uses a number of threads that are managed by a parent `ActorSystem`. The way the actors are executed can be customized to allow adjustment to certain use-cases. This allows for application that scale up to a huge number of parallel threads on a huge number of CPUs within the same system.

Additionally Akka is designed to scale out to a large number of dedicated machines that work together. To allow this Akka is designed to be as location transparent as possible. In fact, there is nearly no API for the actual remoting layer of Akka. Instead the remoting is purely driven by configuration. This way it is possible to simply switch a part of the actors to a different host, this way scaling out the application. The location of an actor is completely encapsulated within an `ActorRef` so the user doesn't need to care about the actual location of a specific actor. However, he needs to be aware that a message send to an actor may take a few seconds to reach its target (depending on layout and size of the network).

Additionally, the programmer must be aware of the fact that messages between actors may actually be send over a socket. As long es all actors run within the same physical memory, sending messages is nearly free of cost. However, when the application is scaled out this may very well become a factor.

IV. Advanced Akka Features

In this section we will discuss some features of Akka that are not strictly part of the actor model itself, but are rather additions to ease the development of real world applications and to make them more scalable. Most of these features can even be used without actors. After all, actors are only one tool to solve problems and for certain problems they might not be the optimal choice.

A. Supervision and Monitoring

In Akka each running actor has a supervisor which is the parent actor that created the current one. So each given actor is part of a hierarchy of supervisor: the one parent that created the actor and all child actors, that were created by itself were it is the supervisor. But what does supervision mean? A supervisor created the children so it either delegated some kind of work to them or is a least somehow interested in them. This means it is responsible for watching its subordinates and handling problem that they themselves can not handle.

If an actor detects a failure (i.e. if an exception is thrown) it suspends itself and all its children and signals a failure to its supervisor. The supervisor can now choose how to respond this particular failure by

- simply resuming the subordinate. In this case the subordinate keeps all its internals state and will continue were it left of.
- restarting the subordinate. The subordinate will in this case clear all it's internal state.
- terminate the subordinate permanently.
- escalate the failure to its own supervisor.

It has to be noted that in Akka there is no way to put a message back into the inbox. So if a actor signals a failure the current message may be lost. Also it is important to note that restarting an actor will reset all it's internal state but not it's mailbox. So the "new" actor will continue to process all messages in the mailbox after the restart is complete. Additionally, restarting an actor will create a new actor behind the same `ActorRef` so other actors that had a reference to the restarted actor can and will not know, that the actor has been restarted. If this wouldn't be the case you would have to notify all other actors whenever a single actor crashed so that they could update eventually stored references.

However, there might by certain situations where some actor wants to be notified an other actor is permanently stopped. For this cases Akka provides a feature called `DeathWatch` were the watching actor is notified by a special message type, that is delivered to its inbox.

If we recap that each actor must have a parent actor the supervises it, we notice a chicken-and-egg problem: Who starts the first actor? This is done by a special actor (that is in fact not a real actor) called the "bubble-walker" (because he lives "outside the bubble") which in turn will receive all failures that were escalated to the top level. If the bubble walker ever gets such a failure he will terminate all actors which effectively stops the system. [5]

B. Routers

Routers are a special type of actor that route incoming messages to other actors. You can either implement your one ones or use some of the implementation that Akka provides. There are in example some simple ones, that ca be used to load balance messages to a number of actors (on different hosts) this way distributing the workload [1]. Additionally to these simple "distributing" routers there are ones that broadcast messages to all actors or broadcast them and wait for the first result to complete[2].

Routers can also be used to automatically adjust the number of actors depending on the number of messages by spawning or stopping actors.

C. Software Transactional Memory

If you problem can easy be splitted into relatively independent subtask, the actor model is fine. Each actor can solve its own part of the problem, either directly or by reducing it into smaller parts and distributing it to sub-actors and send the results back. But what if you really *need* a truly shared state within those actors (i.e. a bank account where people deposit and withdraw and where each operation needs to be atomic). STM solves this by providing an abstraction layer above lock-based concurrency where you can define certain blocks of code to be atomic towards a certain data structure. STM in Akka is provided be the ScalaSTM library [3] and can i.e. be used to share a data structure between a number of actors.

Another way to use STM in Akka is with the use of Transactors. Transactors are used when you have a certain set of actors, each having some internal state, by sending a message to them, that needs to be updates in one transaction. So if one of the actors fails to update its state all other actions need to be undone.

D. Event Bus

Additionally to the whole message passing Akka provides an event bus[6] that allows parts of the code to publish events to the event bus and other parts may independently subscribe to certain events and get notified if these events occur.

This features can be used to decouple actors from each other but have them be able to pass information around using the event bus.

E. Futures

Future in Akka are included from the Scala Standard Library. Futures are a way to access the result of an concurrent operation. They are in particular useful for Akka if you want to your actors to return some data to non-actor code. I.e. if you have an actor system that does a complicated calculation it could normally return the result by sending a message back to the original requester. However, this is only possible if the requester was itself an actor. If you want to invoke this actor system from a non-actor context you can use the so called "ask"-Pattern which sends a message to and actor and returns a `Future` which you can later use to retrieve the result the actor sent back.

F. Performance

The overhead of an actors system is relatively low in terms of memory consumption. Each actor consumes about 300 bytes of memory, which allows for roughly 3 Million actors per GB of main memory (this of course only holds true if there are no messages send and each actors has no internal state which makes the hole system pretty useless). The Akka documentation states that you should view actors as abundant and simply use as many as you need. [7]

The maximum message throughput of an actor system is connected to the number of available CPU cores. Figure 1 shows an example benchmark for the Akka 2.0 release.

[1] i.e. `RoundRobinRouter` or `SmallestMailboxRouter` http://doc.akka.io/docs/akka/2.1.0/scala/routing.html

[2] `BroadcastRouter` and `ScatterGatherFirstCompletedRouter`

[3] http://nbronson.github.com/scala-stm/

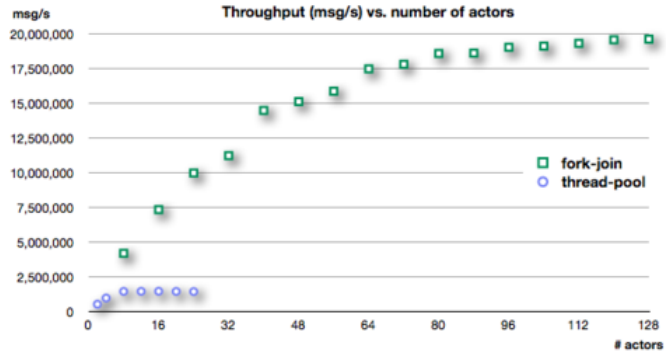

Fig. 1. Message throughput of an actor system comparing two different thread pool executors. The test machine was had 48 AMD Opteron cores with 128GB of ECC memory running Ubuntu 11.10 and OpenJDK 7. ©2011-2013 - Akka Team, http://letitcrash.com/

V. Akka compared to other actor implementations

In this section we will do a qualitative comparison of Akka and certain other implementations of the actor model. As Wikipedia currently lists 16 programming languages that employ the actor model and additionally 24 libraries that allow an actor-style programming model for 15 different languages that don't have a native actor implementation. Since comparing all of them is beyond the scope of this paper, we will limit ourselves to a subset of them.

The first limitation is, that we will only compare Java or Scala based solutions. Akka, as a framework for Java and Scala applications, is our baseline it wouldn't be a very fair to compare a actors library for C with a Java one.

The second limitation is, that we will only compare solutions that have a certain popularity. For this, we will have a look at the popular programming Q&A site Stack Overflow[4]. According to Alexa[5], Stack Overflow is ranked on place 86 in their 3-month worldwide traffic ranking and is very well known to programmers so there should be enough data available for us to estimate the popularity of each solution. For the ranking we will look at the number of user questions that were tagged with the solution in question and will additionally use the number of total posts that mention it.

[4]http://stackoverflow.com

[5]http://www.alexa.com/

TABLE I
Popularity comparison results

Solution	# questions tagged	# posts
Akka	606	2002
Scala built-in actors	-	1068
Jetlang	0	31
Kilim	0	30
Ateji PX	0	12
ActorFoundry	0	2
Korus	0	1
S4	0	-

Table I shows the result of this preliminary analysis. As you can see, Akka is the far most popular solution. In the following section, we will compare the top four of this list.

A. Scala built-in actors

The Scala standard library has already a built-in actor implementation. As well as Akka it provides a base class you can extend and a methods for receiving message in it. However, it allows two modes of execution. The first method is that each Actor owns a thread within the JVM that blocks whenever the actors waits for a message. This approach isn't a problem for a few dozen actors but if you scale this up you will certainly reach the limit of threads the underlying JVM can manage. The second method is an event-driven approach as in Akka, where a pool of threads switches between the execution of the actors and only executed the ones that have messages in their inbox.

As well as Akka their are a number of different schedulers that determine how the actors are executed (i.e. their is an `SingleThreaded Scheduler` that only uses one thread to execute all actors). Scala also has built-in remote actors. However, the configuration whether an actor is remote or not, is done within the source code so scaling the application may require certain changes to the codebase. In contrast, Akka solves this purely by configuration which makes scaling out less an issue[8].

While writing of this document, the new Version 2.10 of Scala was released. In that version the Scala actors are deprecated while the Akka actor implementation will be integrated into the Scala

core [9].

B. Jetlang

Jetlang[6] is a simple no-dependency Java library that is an implementation of message based concurrency for Java. It also features lightweight, actor-like constructs that receive message and are backed by an execution model that is based on threads or pools of threads. In contrast to Akka where an actor can only exchange a message with a single actor, Jetlang provides a mechanism where multiple actors can subscribe to certain messages on a so called `Channel` which acts as a transport medium for messages.

It doesn't feature Futures or any other of the advanced constructs Akka provides - you would have to implement them on your own. While it has a small TCP based remoting feature it's optimized for high throughput in-memory messaging.

C. Kilim

Kilim[7] also provides a lightweight model with message passing but uses an entirely different approach as Akka or Jetlang. While Akka and Jetlang are implemented on a relatively high level using the normal language features, Kilim incorporates byte code manipulation, called *weaving*, in the post-compilation phase. Kilims light-weight threads, called fibers, use a cooperatively-scheduled execution model. Switching between these threads is done by an automatic stack-management (thus avoiding Java heavyweight threads) which allows Kilim to pause the execution of one fiber, save the current stack to an object, and continue to execute with another fiber. This feature requires a post-compilation byte-code processing and a runtime library to work. The authors claim that this allows Kilim to be extremly fast when using a large amount of fibers [10], [11].

Kilim also features fast, in-memory message passing by using a zero-copy approach: Messages are not actually send from one actor to another, theres is only a reference passed around. To ensure isolation of states between actors, the message are checked to be either immutable or only be "owned" by one actor at any given time. This is statically checked at compile time, using special annotation and a common base class for alls messages[10].

Kilim does include a limited networking support and remote message, however these features are not the core focus of Kilim.

[6]http://code.google.com/p/jetlang/

[7]http://www.malhar.net/sriram/kilim/

D. Comparison results

Compared to the other examined solution the Akka Library is by far the most feature rich actor implementation. It has a broad user base and community, a commercial company supporting the development (and providing commercial support) and and excellent documentation. However, it is a lot more "general purpose" than Jetlang and Kilim. They both focus a lot more on very fast, local concurrency wheres Akka focuses more on a unified local and remote actor implementation. Some of the features that Akka provides, like the Supervision and Monitoring features, are very useful if you want to build huge, self healing systems with it.

VI. Conclusion

It is certainly easier to write a large, concurrent application with the help of Akka as opposed to the traditional threads-and-shared-data model. However, this requires that you are familiar with the actor model. We found it very hard to wrap our heads around the whole "small communicating entities" idea in the first place. And if you manage this hurdle you face the problem to actually write your application with actors. In how many pieces should you break your problems? When is one problem small enough to be solved by a single actor? And how do you organize you system in general? This are all question one has to experiment with and learn the answers from the results. While the Akka documentation itself is really well written, their are only a few guides on how to solve a real problem. This section could really be improved by providing examples and case studies in the form of code and documentation.

However, we still believe that actors are certainly worth looking at and every developer should familiarize oneself with this model of programming.

REFERENCES

[1] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624775.1624804

[2] Wikipedia. (2012) Message passing — wikipedia, the free encyclopedia. [Online; accessed 2-January-2013]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Message_passing&oldid=522155441

[3] École Polytechnique Fédérale de Lausanne (EPFL). (2013) A tour of scala: Pattern matching. [accessed 13-January-2013]. [Online]. Available: http://www.scala-lang.org/node/120

[4] Typesafe Inc. (2013) Akka documentation "discussion: Message ordering". [Version 2.1; accessed 2-January-2013]. [Online]. Available: http://doc.akka.io/docs/akka/2.1.0/general/message-delivery-guarantees.html#Discussion__Message_Ordering

[5] ——. (2013) Akka documentation "supervision and monitoring". [Version 2.1; accessed 3-January-2013]. [Online]. Available: http://doc.akka.io/docs/akka/2.1.0/general/supervision.html

[6] Wikipedia. (2012) Publish–subscribe pattern — wikipedia, the free encyclopedia. [accessed 3-January-2013]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Publish%E2%80%93subscribe_pattern&oldid=525393535

[7] Typesafe Inc. (2013) Akka documentation "actor systems". [Version 2.1; accessed 6-January-2013]. [Online]. Available: http://doc.akka.io/docs/akka/2.1.0/general/actor-systems.html

[8] P. Haller and S. Tu. (2013) The scala actors api. [accessed 13-January-2013]. [Online]. Available: http://docs.scala-lang.org/overviews/core/actors.html

[9] V. Jovanovic and P. Haller. (2013) The scala actors migration guide. [accessed 16-January-2013]. [Online]. Available: http://docs.scala-lang.org/overviews/core/actors-migration-guide.html

[10] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for java," in *European Conference on Object Oriented Programming ECOOP 2008, Cyprus*, 2008.

[11] S. Srinivasan, "A thread of one's own," in *Workshop on New Horizons in Compilers*, December 2006.