Writing assignment 2, AKKA

Authors:	Rein Spanjer, Bob Schijf, Jorrit Stutterheim
Group number:	10
Student ID Jorrit Stutterheim	13957899
Student ID Rein Spanjer	13558307
Student ID Bob Schijf:	14076357
Study:	Premaster software engineering
Course:	Programmeertalen

Table of Contents

Akka's actor model

Q1: In your own words, explain how message-passing in Akka avoids the problem of corrupted state, without having to rely on the programmer for avoiding the problem (e.g. by introducing locks to the code). In your answer, make explicit what is meant by the state of an actor and how messages are processed by an actor.

As stated by the Akka documentation[1], sending messages has the following two guarantees:

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

An 'actor' in Akka is a newly created separate process (Akka lightweight process, not a full system thread), which defines it's own state and behavior. The only way to communicate with an actor is to send it messages on its message queue, in general messages are handled FIFO (First In First Out). Thus, since each actor has its own state (no shared memory), and messages are handled one by one, state can not be corrupted by concurrent writes to the same memory block. Developers can change the order of message handling, by for example implementing a priority queue mailbox, still messages are handled one-by-one, which prevents memory corruption.

It is a rule of thumb in Akka to only send immutable messages, which prevents Actors writing in the same memory. A developer *could* send mutable objects though and through this create race conditions or memory issues, as demonstrated by the below Scala code taken from the Akka documentation[1]

```
    var state = ""
    val mySet = mutable.Set[String]()

    def onMessage(cmd: MyActor.Command) = cmd match {
      case Message(text, otherActor) =>
        // Very bad: shared mutable object allows
        // the other actor to mutate your own state,
        // or worse, you might get weird race conditions
        otherActor ! mySet
```

Q2: According to Akka's documentation, its message-delivery system guarantees two properties about the delivery of messages. In your own words, explain these properties and their relevance.

According to Akka's documentation these two properties are:

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

at-most-once delivery means that an actor will send a message only once and in a 'fire and forget' manner, this means a message might not be delivered. An advantage of this method is that it is the least overhead, which makes for the greatest possible performance in this model. In short, applications that do not need 100% message delivery are not bothered by complicated implementations that try to do this, applications that do need more guarantees need to be build keeping this in mind. (Proper failure propagation, business logic that can handle failures etc.)

message ordering per sender–receiver pair means that Akka guarantees message order between a single sender and receiver, however if multiple actors send to one actor messages can arrive in a 'mixed' order. An exception to this rule is failure communication between a parent a child, which goes through a separate system mailbox, as shown in the following example taken from the Akka documentation [2]

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

Child actor fails with failure F

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

Q3: In your own words, why does Akka not guarantee the delivery of messages? What are the arguments made to this design decision by the documentation? Mention at least three reasons.

As passingly mentioned above Akka documentation proposes several reasons for not guaranteeing message delivery. First of all it is ambiguous what guaranteed messaging delivery means, does it mean sending the message? making sure the recipient receives it in it's mailbox? That the recipient processes the message? No single messaging system would be able to comply to all of these interpretations. Furthermore it is stated that the only real guarantee that an interaction was successful is by receiving a business logic level acknowledgment from the receiving party, this is not something that distributed programming language wants to guarantee or is easily able to provide. Lastly, as mentioned in question 2, 'fire and forget' messaging reduces overhead and increases performance. Any program written that does not need stronger guarantees also does not suffer from an in-build implementation that provides stronger delivery guarantees. Systems that do need it can build with this in mind.

Q4: In your own words, what mechanism(s) does Akka provide to handle failures? In your answer, mention the hierarchy formed by Akka actors and different possible responses to an observed failure. Be specific.

Before going into failure lets have a quick look at how actors are interrelated. When creating an Akka application the 'ActorContext' already creates two actors as per the docs[3]:

- / the so-called root guardian. This is the parent of all actors in the system, and the last one to stop when the system itself is terminated.
- /system the system guardian. Akka or other libraries built on top of Akka may create actors in the system namespace.

And also the:

- /user the user guardian. This is the top level actor that you provide to start all other actors in your application.

Every actor spawned now will be descendant from the user guardian. We have established that every actor has a parent en possibly children. Actors usually implement a stop message to stop themselves, if an actor is stopped al its children are stopped recursively, meaning that the farthest descendant will be stopped first and work its way up the tree.

Now this relation is understood its time to look at failures. Failures and exception propagate up to an actors parents, when an actor creates a child it can define a 'supervision strategy' to handle these failures. The default strategy is to stop a child after failure. The three supervision strategies are, according to the docs[4]:

- Resume the actor, keeping its accumulated internal state
- Restart the actor, clearing out its accumulated internal state, with a potential delay starting again
- Stop the actor permanently

Finally another tool is 'lifecycle monitoring', in which an actor monitors the 'the transition from alive to dead' from another specified actor, if an actor suddenly terminates the monitoring actor can take appropriate action.

Q5: A significant difference between a message being processed and a method/function/procedure being called, is that processing a message does not return a value. What does the Akka documentation suggest a programmer do instead? In your answer, refer to at least one of the 'interaction patterns' described in the documentation.

There are several methods to request a reply from another actor. Care needs to be taken when choosing a method for getting a return value since each method has certain advantages and disadvantages.

The simplest ones is a 'request-response' model, where an actor sends a special 'Request' class message to which another actor will send its response. An advantage of this that the actor sending the initial message will keep handling incoming messages, its not blocked.

If an actor needs to have a response before doing anything else the 'Request-Response with ask between two actors' model can be used. In this case the actor sets up a listener for a response and does not do any other work until the response is received or times out. Disadvantage here are that a blocked actor take up resources, do their work slower, and it is difficult to set a proper timeout, the message might come successfully just a little after a timeout.

Q6 Explain an additional interaction pattern and motivate it through an example of your own choosing.

The 'General purpose response aggregator' is an interesting pattern where a single actor needs to gather info from many other actors and sends back the total (aggregated) response to the requestor. An example that could use this pattern and we all know is Google flights. In which google flights will check many airlines for flight information and returns all results to the user.

Embedded nature of Akka

Q7:In either Java or Scala, is Akka a compiled or interpreted language? Explain how you concluded this. If interpreted, in what programming language is the interpreter written? If compiled, what is the target language instruction set of the compiler?

Akka can be used within Java and Scala. Java and Scala both compile to bytecode which is interpreted by the Java Virtual Machine. In this way you could argue that the toolkit Akka can be described as an compiled and interpreted language. Because the JVM is platform dependent it can be written in several languages. The Oracle JVM is written in C/C++[5].

Q8: In either Java or Scala, is Akka statically or dynamically typed? How did you conclude this?

Akka embedded in Java is statically typed. Java itself is statically typed because the type of the variable is known at compile time. If an Akka actor receives a message the type of the message is known. Because when you want to interact with an Akka actor you need to send the right message `ActorRef<T>` where T stands for type that de actor accepts.

Q9: Describe an application for which you think Akka's actor-oriented programming is particularly suitable. Argue whether for this application you would prefer to use the Java's Akka, Scala's Akka, or Erlang.

An actor-oriented programming language like Akka is best use for for applications that need to flexibly scale and perform. A good real world example would be money transactions. Banks usually need to handle millions of transactions a day, using the actor model will mean this can be done while keeping the performance high and running many transactions concurrently.

There are several factors that come in when choosing a specific language like:

- A programmers skill-set
- Market for a specific skill-set (as in, can I hire programmers with the required skill-set)
- Current IT eco-system

Within our group there already was some disagreement on choosing one of the three, some prefer a more functional approach so would opt for Erlang, others have more experience with OO and would go for Java or Scala.

Q10: How can Akka programs be debugged? Is this simpler or more difficult than in other languages you are familiar with? How so?

It is more difficult to debug Akka programs because of their async nature, a line-by-line debugger is therefore hard to do. The business logic can be debugged without using Akka in a more synchronous manner. Logging will be the main way of debugging in Akka programs. When using other languages that are not asynchronous like normal Java it is easy to attach a debugger to the process.

Additional bonus questions

Q11: The following page contains links to several additional Akka modules that extend the functionality of Akka even further. For one of these modules: explain what it is for by mentioning the kinds of applications that can benefit from using this module. Pick an example application of your own choosing. https://doc.akka.io/docs/akka/current/common/other-modules.html

Akka HTTP is an extra module which gives the functionality of exposing the Akka application through HTTP or calling own HTTP api's. HTTP is the main way how different systems talks to each other. If you want to integrate an Akka application into a diverse architecture then with this module it is possible. For example it can be implemented as an authorization server.

A user wants to do an payment and goes to an api gateway. This needs to know if this user is allowed to do that payment. The authorization server get information from the api gateway and will check in all kinds of sources in parallel if it is allowed. These sources can all be information from other systems that exposes there services in HTTP. Performance is really important in this setup because authorizations will be done for all api traffic. Akka can help here by doing the checks in parallel.

Q12: Based on your observations, what are the main similarities and differences between Akka and Erlang? You can refer to syntactic, semantic, or pragmatic aspects of the languages. Do the languages differ in terms of the concepts associated with the programming paradigms discussed in this course?

From a programming paradigm perspective the main difference is that Erlang is a mostly functional language and does not support classes and a 'object orientated' approach of programming. Akka supports both object orientated programming and functional programming both in Scala and Java, although it is good to note that generally speaking the object orientated approach is more natural in Java and functional programming is slightly better supported in Scala.

One of the pragmatic differences is that Akka integrates with the Java ecosystem where Erlang needs OTP installed. Both have some form of hot code swapping but Erlang can do this much better than Akka because of the lack of flexibility in the JVM class reloading. Both are semantically strongly typed. They differ a lot syntactically. Erlang has a more compact way of programming where Java is very verbose due to it's object orientated approach.

Q13: Write a concurrency abstraction using Akka actors that implements remote procedure calls. The goal of this abstraction is to make it possible to just execute a function/method that does the communication for us and waits for a response which it returns as the result of the call. You can use either Scala or Java or pseudo-code resembling Scala or Java.

The Communicator and the Listener are actors that responds to each other. If the exchange has reached 100 times the Communicator stops with the messaging. The Initiator will first start the Actors and sends the first message to start the communication.

```
public class Communicator extends AbstractBehavior<Communicator.Hello> {

    public static class Hello {
        public final int times;
        public Hello(int times) {
            this.times = times;
        }
    }


    public static Behavior<Hello> create() {
        return Behaviors.setup(Communicator::new);
    }

    private final ActorRef<Listener.Request> listener;

    private Communicator(ActorContext<Hello> context) {
        super(context);
        listener = context.spawn(Listener.create(), "listener");
    }

    @Override
    public Receive<Hello> createReceive() {
        return newReceiveBuilder().onMessage(Hello.class,
this::onSayHello).build();
    }

    private Behavior<Hello> onSayHello(Hello command) {

        System.out.println("I have reveived [" + command.times + "]
hello's" );
        if (command.times > 100){
            System.out.println("Ok ok, I will stop!" );
        } else {
            listener.tell(new Listener.Request(command.times + 1,
getContext().getSelf()));
        }


        return this;
    }
}
```

```
import akka.actor.typed.ActorSystem;

public class Initiator {
  public static void main(String[] args) {

    final ActorSystem<Communicator.Hello> communicator =
ActorSystem.create(Communicator.create(), "somestring");
    communicator.tell(new Communicator.Hello(0));
  }
}
```

```
public class Listener extends AbstractBehavior<Listener.Request> {

    public static class Request {
        public final int times;
        public final ActorRef<Communicator.Hello> replyTo;

        public Request(int times, ActorRef<Communicator.Hello> replyTo) {
          this.times = times;
          this.replyTo = replyTo;
        }
      }

    public static Behavior<Request> create() {
        return Behaviors.setup(Listener::new);
    }

    private Listener(ActorContext<Request> context) {
        super(context);
    }

    @Override
    public Receive<Request> createReceive() {
        return newReceiveBuilder().onMessage(Request.class,
this::onRequest).build();
    }

    private Behavior<Request> onRequest(Request request) {
        System.out.println("I am listening [" + request.times + "]
times");
        request.replyTo.tell(new Communicator.Hello(request.times));
        return this;
    }
}
```

References

1. https://doc.akka.io/docs/akka/current/general/jmm.html
2. https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html
3. https://doc.akka.io/docs/akka/current/typed/guide/tutorial_1.html
4. https://doc.akka.io/docs/akka/current/general/supervision.html
5. https://docs.oracle.com/javase/specs/jvms/se7/html/