4. Exercises: Actor model

DCC-FCUP, University of Porto
José Proença

Concurrent Programming – Part 2

These exercises are taken mainly from the book "*Learning Concurrent Programming in Scala*", to test the understanding over the actor programming model in Akka. Try to solve these exercises by first assuming that no machines fail, and then consider what happens if some of the machines fail during the execution of the program:

Exercise 1. Implement the timer actor with the `TimerActor` class. After receiving a `Register` message containing the `t` timeout in milliseconds, the timer actor sends a `Timeout` message back after `t` milliseconds. The timer must accept multiple `Register` messages. Test your code with another actor `TestActor` that interacts with the `TimerActor`.

Exercise 2. Recall the bank account example from the slides on Java's memory model in the theoretical lessons. Implement different bank accounts as separate actors, represented by the `AccountActor` class. When an `AccountActor` class receives a `Send` message, it must transfer the specified amount of money to the target actor. What will happen if either of the actors receives a `Kill` message at any point during the money transaction?

Exercise 3. Implement the `SessionActor` class for actors that control access to other actors:

```
class SessionActor(password: String, r: ActorRef)
extends Actor {
  def receive = ???
}
```

After the `SessionActor` instance receives the `StartSession` message with the correct password, it forwards all the messages to the actor reference `r`, until it receives the `EndSession` message. Use behaviours to model this actor.

Exercise 4. Use actors to implement the `ExecutionContext` interface, described in the slides over traditional concurrency blocks.

Exercise 5. Implement the `FailureDetector` actor, which sends `Identify` messages to the specified actors every `interval` seconds. If an actor does not reply with any `ActorIdentity` messages within `threshold` seconds, the `FailureDetector` actor sends a `Failed` message to its parent actor, which contains the actor reference of the failed actor.

Exercise 6. Implement a `FlowRateActor` class for an actor that forwards incoming messages to a target actor. This actor must ensure that the number of messages forwarded per second does not exceed a rate specified in its constructor.

Exercise 7. Implement a `Sequencer` actor, which forwards messages to the target actor. If the message is a two-element tuple where the first element is a `Long` value, then the `Long` value is interpreted as a sequence number. All such messages must be forwarded in the proper sequence number order, starting from number 0.