Exercise 5: Actors

TECHNISCHE UNIVERSITÄT DARMSTADT

Concepts and Technologies for Distributed Systems and Big Data Processing – SS 2017

Introduction

The project for this exercise is provided as an SBT project (`http://www.scala-sbt.org/`). SBT can be used to easily run your Scala projects. Install SBT on your platform and make sure you have the Java 8 SDK installed. You can run the project, by running the command `sbt run` inside of the project source directory. If you like to develop in Eclipse, you first need to install the ScalaIDE for Eclipse (`http://scala-ide.org/`). Next, run `sbt eclipse` and use "Import existing project" in Eclipse on your project directory. IntelliJ users can directly open the SBT project in the IDE.

In this exercise, we use the latest version 2.5.2 of Akka[1]. You can find the documentation here: `http://doc.akka.io/docs/akka/current/scala/index.html`. The following snippet shows, how actors with and without parameters can be instantiated. Further information can be found in the documentation (`http://doc.akka.io/docs/akka/current/scala/actors.html`).

```scala
class MyActor extends Actor {
  def receive: Receive = ???
}

class MyActorWithParams(a: String, b: Int) extends Actor {
  def receive: Receive = ???
}

val system = ActorSystem("MyActorSystem")
val actor1 = system.actorOf(Props[MyActor])
val actor2 = system.actorOf(Props(classOf[MyActorWithParams], "foo", 5))
// same as val actor2 = system.actorOf(Props(new MyActorWithParams("foo", 5)))
```

Task 1 Getting Started

Write a simple actor that is able to receive messages that contain a number or a string. Whenever the actor receives such a message, it simply doubles the value and prints the result. The result of doubling a number n is simply $2*n$. The result of doubling a string str results in $strstr$.

Task 2 Implementing a Simple Chat

In this task, you should implement a very simple chat application, consisting of a chat server and multiple clients. A client needs to log in before the server accepts messages of this client. After logging in, the chat server sends the chat history of all previously sent messages to the client. If a client sends a message to the chat server before logging in, the message is simply ignored. Otherwise, the message is appended to the chat history and forwarded to all other clients. For simplicity reasons, we only use local actors with a single actor system and leave out any remote communication.

Task 2.1 Chat Server

First, we need to define the messages that the chat server accepts. Create a `LoginMessage` and a `LogoutMessage` that both carry the user name of the client, a `HistoryMessage` that carries the chat history and finally a `ChatMessage` that carries the user name of the sender and the current message sent. You can use strings for all parameters. Next, implement the actor for the chat server. When the chat server receives a `LoginMessage`, it should store the user name together with a reference to the sending actor (`ActorRef`). The reference to the sender can be obtained by calling the `sender` method.

[1] `http://akka.io`

In the following snippet, when the `EchoActor` receives a message `MyMessage` from another actor, it sends a new message of the same type back to this actor.

```
case class MyMessage(p: String)

class EchoActor extends Actor {
  def receive: Receive = {
    case MyMessage(p) => sender ! MyMessage(p)
  }
}
```

If the history is not empty, the chat server should send a `HistoryMessage` containing the chat history back to the newly connected client. When the chat server receives a `LogoutMessage`, it should remove the respective actor reference from the list of currently active clients. When the chat server receives a `ChatMessage` from a client that is already logged in, it should append the message with the user name to the chat history and forward the message to all currently connected clients.

Task 2.2 Chat Client

As shown in the following snippet, a chat client should be controlled by sending different kinds of messages. Provide implementations for the messages `Login`, `Logout` and `Say`.

```
object Chat extends App {
  ...
  val client1 = system.actorOf(...)
  client1 ! Login
  client1 ! Say("Hello")
  client1 ! Logout
}
```

Next, implement the actor for the chat client. When the chat client receives a `Login` message, it should send a `LoginMessage` to the server with its username and similarly for `Logout` messages. When the chat client receives a `Say` message, it should send a `ChatMessage` with the message text and its user name to the chat server. When receiving a `HistoryMessage` or a `ChatMessage`, the chat client should just print the history and the sender and message respectively on the console.