Straight on with a code example

Whenever I see a blog about a new software library or approach, I tend to jump straight to the examples so that I can quickly appreciate what the library does. So here to make things easy I am going to start with some code examples which I believe makes sense.

When we work with actors, most of the time they will interact with each other. If we have 2 actors, an AddActor and a MultiplyActor, code that performs (x+y)*z will have to send messages to both actors via – say – a third actor. Then somehow will have to make the calls as if they are done sequentially as (x+y) is required before calling the MultiplyActor. Implementing these 3 simple actors can be tricky. How can this be done with the active object approach?

```
trait AddService
{
    // R stands for Result. It allows us to execute our code within an actor
    // with all actor benefits, i.e. no need to sync for access to local variables
    def add(x: Int, y: Int): R[Int]
    def addAndMultiply(x: Int, y: Int, z: Int):R[Int]
}
// @remoteService is a macro annotation. The AddServiceRemote class will be re-written
// by the macro to i.e. have a method
// def add(x: Int, y: Int)(implicit val timeout:FiniteDuration): R[Int] with RemoteMethods[Int]
// This allows us to do remote calls as if we call local methods and also specify a timeout
// period. If the remote call throws an exception, then the call to it will result in an
// exception thrown.
// RemoteMethods[Int] adds methods like fireAndForget().
@remoteService
class AddServiceRemote extends AddService
// Due to the macro re-writing AddServiceRemote, the class is now :
// class AddServiceRemote(actor:ActorRef) ... extends Serializable {
//    def add(x: Int, y: Int)(implicit val timeout:FiniteDuration): R[Int] with RemoteMethods[Int] =
{ ... impl }
//    def addAndMultiply(x: Int, y: Int, z: Int)(implicit val timeout:FiniteDuration):R[Int] with
RemoteMethods[Int] = { ... impl }
// }
class AddServiceImpl(multiplyService: MultiplyServiceRemote) extends AddService
{
    override def add(x: Int, y: Int) = begin { // "begin" is the R dsl which converts the code to
an R[Int]
        // this code runs within an actor with actor-threading semantics
        x + y
    }
    override def addAndMultiply(x: Int, y: Int, z: Int) = add(x,y).andThen {
        sum =>
            // Note: our code will run pseudo-sequentially. This means that
            // add(x,y) will run first within an actor. Then the actor
            // will tell the multiplyService actor to do the multiplication.
            // (i.e. something similar to multiplyServiceActor ! Multiply(...) )
            // Then the multiplyActor will addActor ! Result(sum*z)
            // and then the addAndMultiply will return with the result.
            // So it all runs non-blocking and pseudo-sequentially.
            multiplyService.multiply(sum, z)(1 minute)
            // did you notice that the addActor will wait for 1 minute for a respond?
            // it will then throw a TimeoutException
    }
}
trait MultiplyService
```

```
{
    def multiply(x: Int, y: Int): R[Int]
}
@remoteService
class MultiplyServiceRemote extends MultiplyService
// Due to the macro re-writing MultiplyServiceRemote, the class is now :
// class MultiplyServiceRemote(actor:ActorRef) ..... {
//       def multiply(x: Int, y: Int)(implicit val timeout:FiniteDuration): R[Int] with
RemoteMethods[Int] = { ... impl }
// }
class MultiplyServiceImpl extends MultiplyService
{
    override def multiply(x: Int, y: Int) = begin {
        x * y
    }
}
// now how do we create the 2 actors and use them?
object MyApp extends App
{
    val system=ActorSystem("mySystem")
    val rActorService = new RActorService(system)
    // instead of system.actorOf(...) we need to create a local and a remote service
    val multiplyServiceRemote=rActorService.service(new MultiplyServiceImpl,name =
"multiplyService")(actor => new MultiplyServiceRemote(actor))
    val addServiceRemote=rActorService.service(new
AddServiceImpl(multiplyServiceRemote),name="addService")(actor => new
AddServiceRemote(actor))
    addServiceRemote.addAndMultiply(5,10,20)(1 minute).fireAndForget()
}
```

Testing the above code is quite easy, I.e to test the addAndMultiply(), we can isolate it by mocking the MultiplyServiceRemote and run the R on the thread that runs the unit test.

Reasons for this approach

The main driver for implementing active objects is my work on creating a distributed sql database. For a lot of IT projects there is a need to ingest and query large volumes of data. Traditional databases fail to distribute these data into multiple servers and to respond quickly enough to queries. Big data technologies do effectively full table scans on the data even if a small subset of the data are required and also the developers need to spend a lot of effort thinking of how to structure the data to optimize sets of queries.

At AKT IT I am developing a next generation big data nosql + sql database that will distribute the data and queries to multiple servers and is meant for tables containing terrabytes of data. This will be a sql distributed database, which means developers can normalize the data and index them as they do on relational databases.

The software is implemented in Scala using akka. Initially an implementation was done using typed actors but because those were deprecated and due to various problems with that model of distributed computing, the implementation was switched to akka actors.

Coding such a complex piece of software with actors and message passing proved cumbersome. There are several sections of the code that were hard to work with and especially hard to unit test. Many times communication between 3 types of actors was required and in a sequential fashion. Other times scatter/gather patterns have to be applied and even schedules to be run in order to modify the state of an actor. Flows of data had to move across different actors, making sure in case of exceptions or timeouts the process fails gracefully. Although we implemented utilities that could flow the data over all required actors (similar to akka streams), unit testing actors using those flows were quite hard due to the volume of participants and that there is no type safety. Refactoring was hard as it is hard to control who sends and who receives certain messages.

The new active-objects approach

During development we realized that a new approach to coding distributed systems is required to ease the development of this software. Standard object oriented approaches proved more appropriate and testing via mocking traits and classes easier. I ended up implementing a simple active-object-like library on top of akka and refactoring actors to active objects.

Those familiarized with spark will feel right at home. Those who know typed actors will also easily understand the library as it has a lot of similarities - but some significant differences.
Lets go straight into an example. Assuming we have 2 actors, an addition and a multiplication one, we want to impl them using this new library.

```
trait AddService
{
    // add x to y and use the remote multiply service to multiply by z
    def addAndMultiply(x: Int, y: Int, z: Int): R[Int]
}

@remoteService
class AddServiceRemote extends AddService

trait MultiplyService
{
    def multiply(x: Int, y: Int): R[Int]
}

@remoteService
class MultiplyServiceRemote extends MultiplyService
```

The above definitions are straight forward apart from the @remoteService part. This is a marker for a macro that rewrites AddServiceRemote to be a remote proxy for AddService. We will go into the details later on. Now the implementations of the traits:

```
class MultiplyServiceImpl extends MultiplyService
{
     // returns an R { x * y }
     override def multiply(x: Int, y: Int) = begin {
         x * y
     }
}
class AddServiceImpl(multiplyService: MultiplyServiceRemote) extends AddService
{
      def addAndMultiply(x: Int, y: Int, z: Int) = {
          val sum = x + y
          // since the multiply service is remote, we need a timeout for every call
```

```
            implicit val timeout = 1 minute
            multiplyService.multiply(sum, z) // executes remotely, returns an R { sum * z }
        }
}
```

Now the above are straight forward. There are some utilities like the begin { } which returns an R[T] but other than that it feels as if we write object oriented code that will execute sequentially. All code runs with actor threading semantics which means we could modify local variables without synchronization. We can chain calls and the code will execute in a pseudo-sequential way, i.e.

```
def addAndMultiplyAndAddOne(x: Int, y: Int, z: Int) = addAndMultiply(x,y,z).map(_ + 1)
```

The above method will return (x + y)*z + 1 . With the benefit that it all seems to run sequentially (and non-blocking) despite the call to the remote multiplication service.

More akka goodies are included. I.e. we can schedule things and resume processing as if it is all executed sequentially:

```
def scheduledMultiplication(x: Int, y: Int, z: Int) = scheduleOnce(500 millis) {
      // this will run after 500 ms of the call, it runs using actor semantics,
      // which means we could modify local variables without synchronization
      x+y
}.andThen { sum =>
      // sum=x+y, we will now call the remote multiply service
      // since the multiply service is remote, we need a timeout for every call
      implicit val timeout = 1 minute
      multiplyService.multiply(sum,z)
}
```

And we can also use Future's, all seemingly executing sequentially. This way if we need to do say a slow I/O operation, we won't block the actor.

```
def futureMultiplication(x: Int, y: Int, z: Int) = future {
      // this will run in a separate thread, not blocking the actor,
      // but access to state needs to be sync'd
      x+y
}.andThen { sum =>
      // sum=x+y, we will now call the remote multiply service.
      // Note that this runs with actor semantics again, so we could modify state
      // without sync
      implicit val timeout = 1 minute
      multiplyService.multiply(sum,z)
}
```

Testing the above active objects is easy and can be done with the usual mocking and testing libs. We can mock the multiplication remote service to isolate the AddService and test it. And we can run the code outside an actor for unit testing purposes. In fact, tests of actors refactored to active objects are far simpler than tests of the actors themselves. And unit testing of some very complicated actor logic became possible.

```
when(multiplyService.multiply(3, 6)).thenReturn(100)
addService.addAndMultiply(1,2,6).value should be (100)
```