developerWorks®

JVM concurrency: Building actor applications with Akka

Go beyond the basics to build applications that use actor interactions

Dennis Sosnoski

August 12, 2015

Actor applications require a different style of programming from the linear approach used for single-threaded applications. Go deeper into structuring systems in terms of actors and messages from Scala code, using the Akka toolkit and runtime.

View more content in this series

"*JVM concurrency*: Acting asynchronously with Akka" introduced you to the *actor model* and the Akka framework and runtime. Building actor applications is different from building traditional, linear applications. With a linear application, you consider control flow and the sequence of steps that are involved to accomplish a goal. To make effective use of the actor model, you decompose your application into independent bundles of state and behavior (the actors) and script out the interactions among these bundles (the messages). These two components — actors and messages — are the building blocks for your application.

If you compose your actors and messages correctly, you end up with a system in which most things happen asynchronously. Asynchronous operation is harder to understand than a linear approach, but it pays off in scalability. Highly asynchronous programs are better able to use increased system resources (for example, memory and processors) either to accomplish a particular task more quickly or to handle more instances of the task in parallel. With Akka, you can even extend this scalability across multiple systems, by using remoting to work with distributed actors.

About this series

Now that multicore systems are ubiquitous, concurrent programming must be applied more widely than ever before. But concurrency can be difficult to implement correctly, and you need new tools to help you use it. Many of the JVM-based languages are developing tools of this type, and Scala has been particularly active in this area. This series gives you a look at some of the newer approaches to concurrent programming for the Java and Scala languages.

In this article, you learn more about structuring systems in terms of actors and messages. The first of two example applications shows the basics of how actors and messages work in Akka. The second, more elaborate example demonstrates planning and visualizing the structure of an actor system. Both examples use Scala code, but they are easy for Java developers to understand (for help, see the previous article in this series for side-by-side examples of Scala and Java programming with Akka).

Download the sample code for this article.

Meet the `Stars`

The examples in the last article use:

- Actors created directly by the main application that starts the actor system
- Only a single type of actor
- A bare minimum of interactions among actors

For the first sample application, I use a slightly more complex structure, which I review piece-by-piece. Listing 1 shows the whole application.

Listing 1. Generations of `Stars`

```
import scala.concurrent.duration._
import scala.util.Random
import akka.actor._
import akka.util._
object Stars1 extends App {
  import Star._
  val starBaseLifetime = 5000 millis
  val starVariableLifetime = 2000 millis
  val starBaseSpawntime = 2000 millis
  val starVariableSpawntime = 1000 millis

  object Namer {
    case object GetName
    case class SetName(name: String)
    def props(names: Array[String]): Props = Props(new Namer(names))
  }
  class Namer(names: Array[String]) extends Actor {
    import context.dispatcher
    import Namer._

    context.setReceiveTimeout(starBaseSpawntime + starVariableSpawntime)

    def receive = {
      case GetName => {
        val name = ...
        sender ! SetName(name)
      }
      case ReceiveTimeout => {
        println("Namer receive timeout, shutting down system")
        system shutdown
      }
    }
  }

  object Star {
    case class Greet(peer: ActorRef)
    case object AskName
    case class TellName(name: String)
    case object Spawn
    case object IntroduceMe
    case object Die
    def props(greeting: String, gennum: Int, parent: String) = Props(new Star(greeting, gennum, parent))
  }
  class Star(greeting: String, gennum: Int, parent: String) extends Actor {
    import context.dispatcher
    var myName: String = ""
    var starsKnown = Map[String, ActorRef]()
    val random = Random
    val namer = context actorSelection namerPath
    namer ! Namer.GetName

    def scaledDuration(base: FiniteDuration, variable: FiniteDuration) =
      base + variable * random.nextInt(1000) / 1000

    val killtime = scaledDuration(starBaseLifetime, starVariableLifetime)
    val killer = scheduler.scheduleOnce(killtime, self, Die)
    val spawntime = scaledDuration(starBaseSpawntime, starVariableSpawntime)
    val spawner = scheduler.schedule(spawntime, 1 second, self, Spawn)
    if (gennum > 1) scheduler.scheduleOnce(1 second, context.parent, IntroduceMe)

    def receive = {
      case Namer.SetName(name) => {
        myName = name
        println(s"$name is the ${gennum}th generation child of $parent")
        context become named
      }
    }
    def named: Receive = {
      case Greet(peer) => peer ! AskName
      case AskName => sender ! TellName(myName)
      case TellName(name) => {
        println(s"$myName says: '$greeting, $name'")
        starsKnown += name -> sender
      }
      case Spawn => {
        println(s"$myName says: A star is born!")
        context.actorOf(props(greeting, gennum + 1, myName))
      }
      case IntroduceMe => starsKnown.foreach {
        case (name, ref) => ref ! Greet(sender)
      }
      case Die => {
        println(s"$myName says: 'I'd like to thank the Academy...'")
        context stop self
      }
    }
  }

  val namerPath = "/user/namer"
  val system = ActorSystem("actor-demo-scala")
  val scheduler = system.scheduler
  system.actorOf(Namer.props(Array("Bob", "Alice", "Rock", "Paper", "Scissors",
    "North", "South", "East", "West", "Up", "Down")), "namer")
  val star1 = system.actorOf(props("Howya doing", 1, "Nobody"))
  val star2 = system.actorOf(props("Happy to meet you", 1, "Nobody"))
  Thread sleep 500
  star1 ! Greet(star2)
  star2 ! Greet(star1)
}
```

This application creates an actor system with two actor types: `Namer` and `Star`. The `Namer` actor is a singleton, effectively a central directory of names. The `Star` actors get their (screen) names from the `Namer`, then print out greeting messages to other `Star`s, as in the last installment's examples. But they also spawn child `Star`s whom they then introduce to the `Star`s they know; and `Star` actors can eventually die.

Listing 2 is a sample of the output that you might see when you run this application.

Listing 2. Application output

```
Bob is the 1th generation child of Nobody
Alice is the 1th generation child of Nobody
Bob says: 'Howya doing, Alice'
Alice says: 'Happy to meet you, Bob'
Bob says: A star is born!
Rock is the 2th generation child of Bob
Alice says: A star is born!
Paper is the 2th generation child of Alice
Bob says: A star is born!
Scissors is the 2th generation child of Bob
Alice says: 'Happy to meet you, Rock'
Alice says: A star is born!
North is the 2th generation child of Alice
Bob says: 'Howya doing, Paper'
Rock says: 'Howya doing, Paper'
Bob says: A star is born!
South is the 2th generation child of Bob
Alice says: 'Happy to meet you, Scissors'
Paper says: 'Happy to meet you, Scissors'
Alice says: A star is born!
East is the 2th generation child of Alice
Bob says: 'Howya doing, North'
Rock says: 'Howya doing, North'
Scissors says: 'Howya doing, North'
Paper says: A star is born!
West is the 3th generation child of Paper
Rock says: A star is born!
Up is the 3th generation child of Rock
Bob says: A star is born!
Down is the 2th generation child of Bob
Alice says: 'Happy to meet you, South'
North says: 'Happy to meet you, South'
Paper says: 'Happy to meet you, South'
Scissors says: A star is born!
Bob-Bob is the 3th generation child of Scissors
Alice says: A star is born!
Bob-Alice is the 2th generation child of Alice
Scissors says: 'Howya doing, East'
Rock says: 'Howya doing, East'
Bob says: 'Howya doing, East'
South says: 'Howya doing, East'
North says: A star is born!
Bob-Rock is the 3th generation child of North
Paper says: A star is born!
Bob-Paper is the 3th generation child of Paper
Bob says: 'I'd like to thank the Academy...'
Scissors says: 'Howya doing, West'
South says: 'Howya doing, West'
```

```
Alice says: A star is born!
Bob-Scissors is the 2th generation child of Alice
North says: A star is born!
Bob-North is the 3th generation child of North
Paper says: A star is born!
Bob-South is the 3th generation child of Paper
Alice says: 'I'd like to thank the Academy...'
Namer receive timeout, shutting down system
```

Generations of `Star`s

Unlike some real-world actors, `Star` actors don't produce offspring in dramatic and public fashion; instead, they quietly pop out a child each time they receive a `Spawn` message. Their only sign of excitement over this event is the simple birth announcement "`A star is born!`" Again, unlike real-world actors, the proud new parent `Star`s can't even announce the name of their new child, which is instead determined by the naming authority. After the fledgling `Star` is named, the `Namer` prints the child's name and details in a line of the form "`Ted is the 2th generation child of Bob`."

A `Star`'s death is triggered by the `Star` receiving a `Die` message, in response to which it prints a message "`I'd like to thank the Academy...`." The `Star` then executes the `context stop self` statement, telling the controlling Akka actor context that it is done and should be shut down. The context then takes care of all cleanup work and removes the actor from the system.

Changing roles

Real-world actors can play many different roles. Akka actors can also take on different roles, by changing message-handler methods. You can see this in the `Star` actor, where the default `receive` method handles only the `SetName` message, and all other messages are processed by the `named` method. The handover occurs in the processing of the `SetName` message, with the `context become named` statement. The intent with this role change is that the `Star` can't do anything until it's named, and after it's named, it can never be renamed.

You can always handle all your message processing in a single `receive` method, but this often makes for messy code with conditional statements based on the current actor state. Using a separate `receive` method for a different state keeps your code clean and direct. In general, any time you have an actor state for which a different message is appropriate, you should favor using a new `receive` method to represent that state.

You do need to be careful that you don't exclude handling of valid messages when you change actor roles. For instance, if `Star` actors were allowed to be renamed at any time, the `named` method in Listing 1 would need to handle the `SetName` message. Any messages that aren't handled by the actor's current `receive` method effectively get dropped (actually, sent to a dead-letter mailbox by default, but dropped as far as your user actors are concerned).

As an alternative to changing the message handler, you can also push the current message handler on a stack and set a new one using the two-argument form `become(named, false)`. You can then eventually restore the original handler with a `context unbecome` call. You can nest calls to `become`/`unbecome` in this way as deeply as you want, but you must be careful that the code eventually executes an `unbecome` matching every `become`. Any unmatched `become`s represent a memory leak.

The `Namer` actor

The `Namer` actor is passed an array of name strings in its constructor. Each time it receives a `GetName` message, it returns the next name in the array in a `SetName` message, going to hyphenated names when it runs out of simple names. The point of the `Namer` actor is to assign names (ideally, unique names) to `Star` actors, so there's no reason to have more than one `Namer` instance in this system. The application code that starts the actor system creates this singleton instance directly so it's available for use by every `Star`.

Because the application creates the `Namer` singleton, it could pass an `ActorRef` for this actor to each `Star`, and the `Star` actors could pass it on to their children. But Akka gives you a cleaner way of handling this type of well-known actor. The `val namer = context actorSelection namerPath` line in the `Star` actor initialization looks up the `Namer` actor by its path in the actor system — in this case, `/user/namer`. (The `/user` prefix applies to all user-created actors, and `namer` is the name that's set when the `Namer` actor is created using `system.actorOf`.) The `namer` value is visible to all actors included in the application, so it can be used directly when needed.

Scheduled messages

The Listing 1 example uses several scheduled messages to prompt the various actors. The `Star` actors create either two or three scheduled messages during initialization. The `val killer = scheduler.scheduleOnce(killtime, self, Die)` statement creates a one-time message scheduler to trigger the death of the `Star` by sending a `Die` message when its time on the stage is over. The `val spawner = scheduler.schedule(spawntime, 1 second, self, Spawn)` statement creates a repeating scheduler that sends `Spawn` messages at 1-second intervals after an initial delay to populate a new generation of `Star`s.

The third type of scheduled message for a `Star` is used only when the `Star` is the offspring of another `Star` (rather than created by the application code outside the actor system). The `if (gennum > 1) scheduler.scheduleOnce(1 second, context.parent, IntroduceMe)` statement creates a scheduled message to be sent to the `Star`'s parent one second after the `Star` is initialized, if the new `Star` is second-generation or later. When the parent `Star` receives this message, it sends a `Greet` message to each other `Star` it's been introduced to, asking these known `Stars` to introduce themselves to the child.

The `Namer` actor also uses a scheduled message, this one in the form of a receive timeout. The `context.setReceiveTimeout(starBaseSpawntime + starVariableSpawntime)` statement sets a timeout to the maximum spawn time for stars. This timeout is reset by the context each time the actor receives a message, so that it fires only if the specified time passes without any messages being received. `Star`s continually create new child `Star`s that send messages to the `Namer`, so the timeout occurs only if all the `Star` actors are gone. If the timeout does occur, `Namer` handles the resulting `ReceiveTimeout` message (defined in the `akka.actor` package) by shutting down the entire actor system.

Sharp-eyed readers might wonder how the `Namer` timeout ever occurs. The lifetime of a `Star` is always going to be at least 5 seconds, and each `Star` starts spawning child `Star`s by the time it's a maximum of 3 seconds old — so it *seems* like there should be an ever-growing glut of `Star`s (kind of like in reality TV). So how does this work? The answer lies in the Akka *actor supervision* model and parent-child relationship.

Families of actors

Akka enforces a supervision hierarchy for actors, based on parentage. When one actor creates another actor, the created actor becomes a subordinate of the original actor. This means the parent actor is responsible for its child actors (a principle we'd often like to see applied to real-world actors). This responsibility is mostly concerned with failure handling, but it does have some implications for how actors work.

The supervision hierarchy is the reason the Listing 1 actor system shuts down. Because the hierarchy requires the parent actor to be available, terminating a parent actor automatically terminates all its child actors. In Listing 1, only two `Star` actors are initially created by the application (which always receive the names `Bob` and `Alice`). All other `Star`s are created by one of these two initial `Star`s, or by one of their child or grandchild `Star`s. So when each of these root `Star`s terminates, it takes all its descendants with it. After both of them terminate, no `Star`s remain. Without any `Star`s to spawn child `Star`s, no requests for names go to `Namer`, so the `Namer` timeout is eventually fired and the system shuts down.

More-complex actor systems

You saw in Listing 1 an example of how a simple actor system can work. But real application systems typically have more types of actors (often into the tens or hundreds) and more-complex interactions among the actors. One of the best ways of designing and organizing complex actor systems is by specifying the message flows between actors.

For a more-complex example, I extend the Listing 1 application to implement a simple model of movie production. This model uses four main actor types and two specialized secondary actor types:

- `Star`: An actor to participate in movies
- `Scout`: A talent scout to find new `Star`s
- `Academy`: A singleton registry that tracks all active `Star`s
- `Director`: A maker of movies
    - `CastingAssistant`: Assistant to a `Director` to cast a movie
    - `ProductionAssistant`: Assistant to a `Director` to make a movie

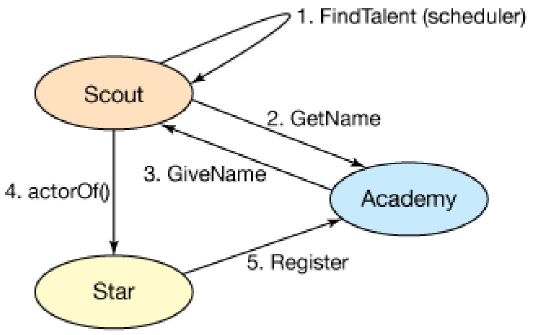
Like the `Star`s in Listing 1, the `Star` actors in this application have limited lifetimes. When a `Director` starts to make a movie, it gets a list of currently active `Star`s to be cast in the movie. First, the `Director` needs to get the `Star`s committed to the movie, and then make the movie after all `Star`s are committed. If any of the `Star`s in the movie quits the business (or dies, in actor terms) before the movie is completed, the movie fails.

Diagramming the messages

The Listing 1 application was simple enough that I could explain the actor interactions in prose. This much more complex new application calls for a better way of presenting the interactions.

Message-passing diagrams are a great way to show these interactions. Figure 1 shows the sequence of interactions that are involved in a `Scout` finding a new `Star` (or in actor terms, creating a `Star`) and that new `Star` registering with the `Academy`.

Figure 1. `Star` creation and initialization



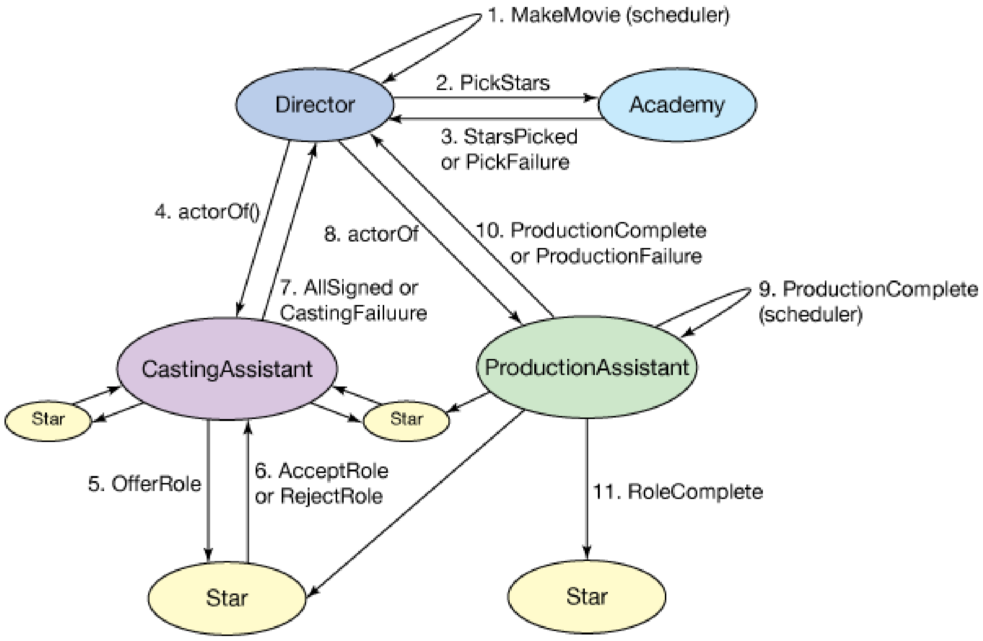
Here's the sequence of messages (and creation steps) involved in adding a `Star`:

1. `FindTalent` (from `Scheduler` to `Scout`): Trigger to add a `Star`.
2. `GetName` (from `Scout` to `Academy`): Allocate a name for `Star`.
3. `GiveName` (response from `Academy`): Supply allocated name.
4. `actorOf()`: `Scout` creates the new `Star` actor with supplied name.
5. `Register` (from `Star` to `Academy`): Register `Star` with `Academy`.

This message sequence is designed to be scalable and flexible. Each message can be processed in isolation, so actors don't need to change their internal states to handle the message exchanges. (The `Academy` singleton changes state, but that's part of the whole purpose of the exchange.) Because no internal state changes, you don't need to enforce a strict sequence of messages. For instance, you can have the `FindTalent` message create multiple `Star`s by sending more than one `GetName` message to the `Academy`. You can even process multiple `FindTalent` messages in succession before you complete the last `Star` creation. You can also add any number of `Scout` actors to your system and have them run independently, without conflicts.

Making a movie is a much more complex process than creating a new `Star`, involving more state changes and potential failure conditions. Figure 2 shows the main application messages that are involved in making a movie:

Figure 2. Making a movie


1. MakeMovie (scheduler)
Director
2. PickStars
Academy
3. StarsPicked or PickFailure
4. actorOf()
8. actorOf
10. ProductionComplete or ProductionFailure
7. AllSigned or CastingFailuure
9. ProductionComplete (scheduler)
CastingAssistant
ProductionAssistant
Star
Star
Star
5. OfferRole
6. AcceptRole or RejectRole
11. RoleComplete
Star
Star

Here's the sequence of messages involved in making a movie, looking mostly at the happy path where everything works without any glitches:

1. `MakeMovie` (from `Scheduler` to `Director`): Trigger to start a movie.
2. `PickStars` (from `Director` to `Academy`): Select the `Star`s to act in the movie.
3. `StarsPicked` or `PickFailure` (response from `Academy`): If enough `Star`s are available to make the movie, the `Academy` selects the required number and sends back the list in a `StarsPicked` message; otherwise, the `Academy` sends a `PickFailure` response.
4. `actorOf()`: `Director` creates a `CastingAssistant` actor to handle casting the movie.
5. OfferRole (`CastingAssistant` to each `Star` in movie): `CastingAssistant` offers the role to the `Star`.
6. `AcceptRole` or `RejectRole` (response from each `Star`): A `Star` rejects an offered role if it's already committed to another role, but otherwise accepts.
7. `AllSigned` or `CastingFailure` (`CastingAssistant` to parent): When all `Star`s have accepted their roles, the `CastingAssistant`'s work is done so it passes the success on to the parent `Director` with an `AllSigned` message; if it's not possible to cast the `Star`s (in particular, if one dies), the `CastingAssistant` passes the failure on to the parent. Either way, the `CastingAssistant` is done and can terminate.
8. `actorOf()`: `Director` creates a `ProductionAssistant` actor to handle filming the movie.
9. `ProductionComplete` (from `Scheduler` to `ProductionAssistant`): Triggers completion of the movie after required time has elapsed.
10. `ProductionComplete` or `ProductionFailure` (`ProductionAssistant` to parent): When the timer fires for the completion of the movie the `ProductionAssistant` reports to its parent that the movie is complete.
11. `RoleComplete` (`ProductionAssistant` to each `Star` in movie): The `ProductionAssistant` also needs to notify each `Star` that the movie is complete, so that they become available for other movies.

This message sequence uses state changes in some of the actors as part of the processing. `Star`s need to change state between being available and being committed to a movie. `CastingAssistant` actors need to track which `Star`s have accepted roles in the movie to be made, so they know which ones they still need to recruit. But `Director` actors don't need to change state, because they respond only to messages that they receive (including messages from their child actors). `ProductionAssistant` actors also don't need to change state, because they only need to notify other actors when the movie terminates.

You could avoid using separate `CastingAssistant` and `ProductionAssistant` actors if you merge their functionality into the `Director` actor. Eliminating the other actors makes `Director` much more complex, though, and in this case it makes more sense to separate the functionality into other actors. This is especially true when you consider handling failures.

Handling failures

One important aspect of the application is left out of the message flows in Figure 1 and Figure 2. `Star`s have limited lifetimes, so all the actors that deal with `Star`s need to be aware of when one dies. In particular, if a `Star` that's been chosen for a movie dies before the movie is complete, the movie necessarily fails.

Failure handling in Akka actor systems uses parental supervision, whereby failure conditions are passed up the hierarchy of actors. Failures are normally represented in the JVM as exceptions, so Akka uses the natural handling of exceptions to detect when a failure occurs. If an actor doesn't handle an exception in its own code, Akka handles the uncaught exception by terminating the actor and passing the failure up to the parent actor. The parent can then handle the failure, or itself fail to its parent.

Akka's built-in failure handling works well for conditions such as I/O-related failures, but for the movie-making system exceptions would be an unnecessary complication. What's needed in this case is monitoring of other actors, and fortunately Akka provides an easy way of doing this. By using the `DeathWatch` component of the actor system, an actor can register itself as watching any other actor. Once registered, the watching actor receives a system `Terminated` message if the watched actor dies. (To avoid any race conditions, if the watched actor is already dead before the watch starts, the `Terminated` message immediately appears in the watching actor's mailbox.)

`DeathWatch` is activated by calling the `context.watch()` method, which takes the `ActorRef` of the actor to be watched. The resulting `Terminated` message when an actor of interest dies is all the failure handling that's needed for the movie-making example.

`Star` creation code

Listing 3 shows the code involved in starting the application and creating new `Star`s, matching the message flow that is shown in Figure 1.

Listing 3. Code for `Star` creation

```
object Stars2 extends App {  object Scout {
    case object FindTalent
    val starBaseLifetime = 7 seconds
    val starVariableLifetime = 3 seconds
    val findBaseTime = 1 seconds
    val findVariableTime = 3 seconds
    def props(): Props = Props(new Scout())
}
class Scout extends Actor {
    import Scout._
    import Academy._
    import context.dispatcher

    val random = Random
    scheduleFind

    def scheduleFind = {
      val nextTime = scaledDuration(findBaseTime, findVariableTime)
      scheduler.scheduleOnce(nextTime, self, FindTalent)
    }

    def scaledDuration(base: FiniteDuration, variable: FiniteDuration) =
      base + variable * random.nextInt(1000) / 1000

    def receive = {
      case FindTalent => academy ! GetName
      case GiveName(name) => {
        system.actorOf(Star.props(name, scaledDuration(starBaseLifetime, starVariableLifetime)), name)
        println(s"$name has been discovered")
        scheduleFind
      }
    }
  }

  object Academy {
    case object GetName
    case class GiveName(name: String)
    case class Register(name: String)
    ...
    def props(names: Array[String]): Props = Props(new Academy(names))
  }
class Academy(names: Array[String]) extends Actor {
    import Academy._

    var nextNameIndex = 0
    val nameIndexLimit = names.length * (names.length + 1)
    val liveStars = Buffer[(ActorRef, String)]()
    ...
    def receive = {
      case GetName => {
        val name =
          if (nextNameIndex < names.length) names(nextNameIndex)
          else {
            val first = nextNameIndex / names.length - 1
            val second = nextNameIndex % names.length
            names(first) + "-" + names(second)
          }
        sender ! GiveName(name)
        nextNameIndex = (nextNameIndex + 1) % nameIndexLimit
      }
      case Register(name) => {
        liveStars += ((sender, name))
        context.watch(sender)
        println(s"Academy now tracking ${liveStars.size} stars")
      }
     case Terminated(ref) => {
        val star = (liveStars.find(_._1 == ref)).get
        liveStars -= star
        println(s"${star._2} has left the business\nAcademy now tracking ${liveStars.size} Stars")
      }
      ...
      }
    }
  }

  object Star {
    ...
    def props(name: String, lifespan: FiniteDuration) = Props(new Star(name, lifespan))
  }
class Star(name: String, lifespan: FiniteDuration) extends Actor {
    import Star._
    import context.dispatcher

    academy ! Academy.Register(name)

    scheduler.scheduleOnce(lifespan, self, PoisonPill)
  }
  ...
  val system = ActorSystem("actor-demo-scala")
  val scheduler = system.scheduler
  val academy = system.actorOf(Academy.props(Array("Bob", "Alice", "Rock",
    "Paper", "Scissors", "North", "South", "East",  "West", "Up", "Down")), "Academy")
  system.actorOf(Scout.props(), "Sam")
  system.actorOf(Scout.props(), "Dean")
  system.actorOf(Director.props("Astro"), "Astro")
  system.actorOf(Director.props("Cosmo"), "Cosmo")
  Thread sleep 15000
  system.shutdown
}
```

The Listing 3 code mostly uses the same Akka functionality as the Listing 1 `Star`s example, with the addition of the `DeathWatch`-activating `context.watch()` call made by the `Academy` actor in handling a `Register` message from a new `Star`. The `Academy` actor records both the `ActorRef` and the name of each `Star`, and when a `Terminated` message is processed, it uses the `ActorRef` to find and remove the `Star` that died. That way the `Buffer` (essentially an `ArrayList`) of live `Star`s stays up to date.

The main application code first creates the singleton `Academy` actor, then a pair of `Scout`s, and finally a pair of `Director`s. The application allows the actor system to run for 15 seconds and then shuts the system down and exits.

Starting a movie

Listing 4 shows the first part of the code involved in making a movie: the casting of `Star`s to participate in the movie. This code matches the top part of the Figure 2 message flow, including the `Scheduler` and the interaction between a `Director` and the `Academy` actor.

Listing 4. Movie making code

```
object Stars2 extends App {
  ...
  object Director {
    case object MakeMovie

    val starCountBase = 2
    val starCountVariable = 4
    val productionTime = 3 seconds
    val recoveryTime = 3 seconds

    def props(name: String) = Props(new Director(name))
  }
```

```
class Director(name: String) extends Actor {
  import Academy._
  import Director._
  import ProductionAssistant._
  import context.dispatcher

  val random = Random

  def makeMovie = {
    val numstars = random.nextInt(starCountVariable) + starCountBase
    academy ! PickStars(numstars)
  }
  def retryMovie = scheduler.scheduleOnce(recoveryTime, self, MakeMovie)
  makeMovie

  def receive = {
    case MakeMovie => makeMovie
    case PickFailure => retryMovie
    case StarsPicked(stars) => {
      println(s"$name wants to make a movie with ${stars.length} actors")
      context.actorOf(CastingAssistant.props(name, stars.map(_._1)), name + ":Casting")
      context become casting
    }
  }
  ...
}
...
object Academy {
  ...
  case class PickStars(count: Int)
  case object PickFailure
  case class StarsPicked(ref: List[(ActorRef, String)])

  def props(names: Array[String]): Props = Props(new Academy(names))
}

class Academy(names: Array[String]) extends Actor {
  ...
  def pickStars(n: Int): Seq[(ActorRef, String)] = ...

  def receive = {
    ...
    case PickStars(n) => {
      if (liveStars.size < n) sender ! PickFailure
      else sender ! StarsPicked(pickStars(n).toList)
    }
  }
}
```

The start of the Listing 4 code gives the `Director` object and part of the actor definition, showing the start of movie production triggered by the `Scheduler` sending a `MakeMovie` message to the `Director`. The `Director` starts the movie-making process when this `MakeMovie` message is received, requesting the `Academy` to assign `Star`s to the movie with a `PickStars` message. The `Academy` code for handling the `PickStars` message, shown at the end of Listing 4, sends back either a `PickFailure` (if not enough `Star`s are available) or a `StarsPicked` message. If the `Director` receives a `PickFailure` message, it schedules another attempt for later. If the `Director` receives a `StarsPicked` message, it starts a `CastingAssistant` actor with the list of `Stars` selected by the `Academy` for roles in the movie, then changes state to handle the response from the `CastingAssistant`. Listing 5 continues from this point, starting with the `Director` actor's casting `Receive` method.

Listing 5. `CastingAssistant` operation

```
  class Director(name: String) extends Actor {
    ...
    def casting: Receive = {
      case CastingAssistant.AllSigned(stars) => {
        println(s"$name cast ${stars.length} actors for movie, starting production")
        context.actorOf(ProductionAssistant.props(productionTime, stars), name + ":Production")
        context become making
      }
      case CastingAssistant.CastingFailure => {
        println(s"$name failed casting a movie")
        retryMovie
        context become receive
      }
    }
    ...
  }

object CastingAssistant {
    case class AllSigned(stars: List[ActorRef])
    case object CastingFailure

    val retryTime = 1 second

    def props(dirname: String, stars: List[ActorRef]) = Props(new CastingAssistant(dirname, stars))
  }

  class CastingAssistant(dirname: String, stars: List[ActorRef]) extends Actor {
    import CastingAssistant._
    import Star._
    import context.dispatcher

    var signed = Set[ActorRef]()
    stars.foreach { star =>
      {
        star ! OfferRole
        context.watch(star)
      }
    }

    def receive = {
      case AcceptRole => {
        signed += sender
        println(s"Signed star ${signed.size} of ${stars.size} for director $dirname")
        if (signed.size == stars.size) {
          context.parent ! AllSigned(stars)
          context.stop(self)
        }
      }
      case RejectRole => scheduler.scheduleOnce(retryTime, sender, OfferRole)
      case Terminated(ref) => {
        context.parent ! CastingFailure
        stars.foreach { _ ! Star.CancelOffer }
        context.stop(self)
      }
    }
  }

  object Star {
    case object OfferRole
    case object AcceptRole
    case object RejectRole
    case object CancelOffer
    case object RoleComplete
    ...
  }
```

```
class Star(name: String, lifespan: FiniteDuration) extends Actor {
  ...
  var acceptedOffer: ActorRef = null

  scheduler.scheduleOnce(lifespan, self, PoisonPill)

  def receive = {
    case OfferRole => {
      sender ! AcceptRole
      acceptedOffer = sender
      context become booked
    }
  }

  def booked: Receive = {
    case OfferRole => sender ! RejectRole
    case CancelOffer => if (sender == acceptedOffer) context become receive
    case RoleComplete => context become receive
  }
}
```

The `Director` creates the `CastingAssistant` with a list of `ActorRef`s for `Star`s to be cast in the movie. `CastingAssistant` first sends an `OfferRole` to each of these `Star`s, and also registers itself as a watcher on each `Star`. `CastingAssistant` then waits for either an `AcceptRole` or a `RejectRole` message back from each `Star`, or a `Terminated` message from the actor system reporting the demise of one of the `Star`s.

If `CastingAssistant` receives `AcceptRole` from every `Star` in the cast, it sends an `AllSigned` message back to its `Director` parent. This message includes the list of `Star actorRef`s as a convenience, because this needs to be passed on for the next processing step.

If `CastingAssistant` receives a `RejectRole` message from any `Star`, it schedules a resend of the `OfferRole` to the same actor after a delay. (Stars are often inaccessible, so if you want them to be in your movie, you need to keep asking until they accept.)

If `CastingAssistant` gets a `Terminated` message, it means one of the `Star`s selected for the movie has died. In this regrettable case, the `CastingAssistant` reports back a `CastingFailure` to its parent `Director` and ends itself. Before it ends, though, it sends a `CancelOffer` message to each `Star` in its list, so that any `Star`s that have committed to roles in the movie are freed to take on other roles.

You might wonder why `CastingAssistant` sends the `CancelOffer` message to *every*`Star`— even the ones from which an `AcceptRole` message has not been processed. The reason is that it's possible a `Star` on the list has sent an `AcceptRole`, but it's still in the mailbox when the `Terminated` message is handled. In the general case of a distributed actor system, it would also be possible that the `Star` has accepted, but the `AcceptRole` message was still in transit or had been lost. Sending the `CancelOffer` message to each `Star` makes the failure handling cleaner in either case, and it's easy for the `Star` to ignore `CancelOffer` messages if it hasn't accepted a role in the movie being made.

Listing 6 shows the last part of the movie-making process: the operation of the `ProductionAssistant` actor (matching the lower right of Figure 2). This part is simple, because the `ProductionAssistant` just needs to handle either the `SchedulerProductionComplete` message or a `Terminated` message.

Listing 6. `ProductionAssistant` operation

```
class Director(name: String) extends Actor {
  ...
  def making: Receive = {
    case m: ProductionAssistant.ProductionEnd => {
      m match {
        case ProductionComplete => println(s"$name made a movie!")
        case ProductionFailed => println(s"$name failed making a movie")
      }
      makeMovie
      context become receive
    }
  }
}

object ProductionAssistant {
  sealed trait ProductionEnd
  case object ProductionComplete extends ProductionEnd
  case object ProductionFailed extends ProductionEnd

  def props(time: FiniteDuration, stars: List[ActorRef]) = Props(new ProductionAssistant(time, stars))
}

class ProductionAssistant(time: FiniteDuration, stars: List[ActorRef]) extends Actor {
  import ProductionAssistant._
  import context.dispatcher

  stars.foreach { star => context.watch(star) }
  scheduler.scheduleOnce(time, self, ProductionComplete)

  def endProduction(end: ProductionEnd) = {
    context.parent ! end
    stars.foreach { star => star ! Star.RoleComplete }
    context.stop(self)
  }

  def receive = {
    case ProductionComplete => endProduction(ProductionComplete)
    case Terminated(ref) => endProduction(ProductionFailed)
  }
}
```

If the `ProductionAssistant` receives the `ProductionComplete` message from the `Scheduler`, it can report back a success to the parent `Director`. If it receives a `Terminated` message first, it has to report a failure. Either way, it also cleans up by telling all the `Star`s involved in the movie that their jobs are done.

Listing 7 is a sample of the output you would see when you run this program, with the movie-making results shown in bold.

Listing 7. Sample output

```
Bob has been discovered
Academy now tracking 1 stars
Alice has been discovered
```

```
Academy now tracking 2 stars
Rock has been discovered
Academy now tracking 3 stars
Paper has been discovered
Academy now tracking 4 stars
Cosmo wants to make a movie with 4 actors
Astro wants to make a movie with 3 actors
Signed star 1 of 4 for director Cosmo
Signed star 2 of 4 for director Cosmo
Signed star 3 of 4 for director Cosmo
Signed star 4 of 4 for director Cosmo
Cosmo cast 4 actors for movie, starting production
Scissors has been discovered
Academy now tracking 5 stars
Cosmo made a movie!
Cosmo wants to make a movie with 4 actors
Signed star 1 of 4 for director Cosmo
Signed star 2 of 4 for director Cosmo
Signed star 3 of 4 for director Cosmo
Signed star 4 of 4 for director Cosmo
Cosmo cast 4 actors for movie, starting production
North has been discovered
Academy now tracking 6 stars
South has been discovered
Academy now tracking 7 stars
Cosmo failed making a movieAstro failed casting a movie
Bob has left the business
Academy now tracking 6 Stars
Cosmo wants to make a movie with 3 actors
Signed star 1 of 3 for director Cosmo
Signed star 2 of 3 for director Cosmo
Signed star 3 of 3 for director Cosmo
Cosmo cast 3 actors for movie, starting production
East has been discovered
Academy now tracking 7 stars
West has been discovered
Academy now tracking 8 stars
Alice has left the business
Academy now tracking 7 Stars
Rock has left the business
Academy now tracking 6 Stars
Up has been discovered
Academy now tracking 7 stars
Astro wants to make a movie with 2 actors
Signed star 1 of 2 for director Astro
Signed star 2 of 2 for director Astro
Astro cast 2 actors for movie, starting production
Cosmo made a movie!
Cosmo wants to make a movie with 3 actors
Signed star 1 of 3 for director Cosmo
Signed star 2 of 3 for director Cosmo
Signed star 3 of 3 for director Cosmo
Cosmo cast 3 actors for movie, starting production
Down has been discovered
Academy now tracking 8 stars
```

The double failure near the midpoint of the listing shows an interesting sequence of outputs. First comes the `Cosmo failed making a movie` line, then `Astro failed casting a movie`, followed by `Bob has left the business`. These lines show the interactions resulting from the termination of one `Star`: `Bob`. In this case, `Bob` had accepted a role in the movie being made by `Cosmo` and production had already started, so `Cosmo`'s `ProductionAssistant` received the `Terminated` message and failed the making of the movie. `Bob` had also been selected for a role in a movie being made by `Astro` but hadn't yet accepted that role (because `Bob` was already committed to `Cosmo`'s movie), so `Astro`'s `CastingAssistant` received the `Terminated` message and failed the casting of the movie. The third message was generated by the `Academy` when it received the `Terminated` message.

Conclusion

Real actor-system applications involve multiple — usually *many*— actors and messages among those actors. This article shows how you can structure an actor system and diagram the actor interactions as an aid to understanding the operation of the system. Working with actors and messages is a different approach to programming than writing sequential code. After you gain some experience, you'll find that the actor approach makes it easy to create highly scalable programs with asynchronous execution.

Structuring the actors and message exchanges goes only so far toward getting your actor system working. At some point, you'll need to track down where your actors are misbehaving. The asynchronous nature of actor systems makes it harder to pinpoint problematic interactions. How to trace and debug actor interactions is a topic worthy in itself of an entire article.

Related topics

- Scalable Scala: Series author Dennis Sosnoski shares insights and behind-the-scenes information on the content in this series and Scala development in general.
- Sample code for this article: Get this article's full sample code from the author's repository on GitHub.
- Scala: Scala is a modern, functional language on the JVM.
- Akka.io is the source for all things Akka, including complete documentation for both Scala and Java applications.
- "Introduction to Actors Systems" (Josh Suereth, DevNexus conference): Josh Suereth designs a distributed search service with Akka using actors, delivering along the way a great overview of many of the great features offered by Akka.