JVM concurrency: Acting asynchronously with Akka

Build actor systems for concurrent applications

Dennis Sosnoski

April 08, 2015

The *actor model* has long been used both as a theoretical basis for analyzing concurrent programs and as a practical approach to implementing concurrent programs. Building on a structure of simple actor entities that communicate via messages, the actor model gives you an easy way to build applications for high concurrency and scalability. Learn about the actor model and start working in Scala or Java™ with the Akka implementation of the model.

View more content in this series

Earlier articles in this series looked at implementing concurrency by either:

- Doing the same operation on multiple sets of data in parallel (as with Java 8 streams)
- Explicitly structuring the computations to do certain operations asynchronously and then combining the results (as with futures)

Both are excellent ways of gaining concurrency, but you must design them explicitly into your application.

Now and in the next few installments, I'll focus on a different approach to concurrency, one based on a particular program structure rather than explicit coding. That program structure is the *actor model*. You'll see how to work with the Akka implementation of the actor model in particular. (Akka is a toolkit and runtime for building concurrent and distributed JVM applications.) See Related topics for a link to the full sample code for this article.

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.

Actor model basics

The actor model for concurrent computations builds up systems based on primitives called *actors*. Actors take actions in response to inputs called *messages*. Actions can include changing the actor's own internal state as well as sending off other messages and even creating other actors. All messages are delivered asynchronously, thereby decoupling message senders from receivers. Because of this decoupling, actor systems are inherently concurrent: Any actors that have input messages available can be executed in parallel, without restriction.

In Akka terms, actors come across as somewhat neurotic bundles of behaviors that interact through messages. Like real-world actors, Akka actors want a degree of privacy. You can't send messages directly to an Akka actor. Instead, you send messages to an actor *reference* that's the equivalent of a post office box. The incoming messages are routed via the reference into a mailbox for the actor, and the messages are delivered to the actor later. Akka actors even demand that all incoming messages be sterile (or in JVM terms, *immutable*) to avoid contamination from other actors.

Akka beyond the basics

Akka implements the actor model with some twists that add flexibility. One such twist is a fully distributed architecture that allows actor systems to spread across multiple nodes in a network. Akka also uses a hierarchical arrangement of actors, wherein each actor (except the system root supervisor) has a parent actor that takes responsibility for handling any failures of child actors. These are both important features of Akka but are somewhat advanced topics that I'll only touch on here.

Unlike some real-world actors' demands, these seemingly obsessive restrictions in Akka exist for a reason. Using references for actors prevents any interactions outside the message exchanges, which could break the decoupling at the core of the actor model. Actors are single-threaded in execution (no more than one thread ever executes a particular actor instance), so the mailbox acts as a buffer for holding onto messages until they can be processed. And immutability of messages (not currently enforced by Akka due to JVM limitations, but a stated requirement) means you never need to worry about synchronization issues affecting data shared between actors; if the only shared data is immutable, synchronization is never needed.

Nice to meet you

Now that you've had an overview of the actor model and Akka specifics, it's time to see some code. Using hello for a coding example is a cliche, but it does give a quick and easily understood snapshot of a language or system. Listing 1 shows an Akka version in Scala.

Listing 1. Simple Scala hello

```
import akka.actor._
import akka.util._

/** Simple hello from an actor in Scala. */
object Hello1 extends App {

  val system = ActorSystem("actor-demo-scala")
  val hello = system.actorOf(Props[Hello])
  hello ! "Bob"
  Thread sleep 1000
  system shutdown

  class Hello extends Actor {
    def receive = {
      case name: String => println(s"Hello $name")
    }
  }
}
```

The Listing 1 code is in two separate chunks, all contained within the `Hello1` application object. The first chunk of code is the Akka application infrastructure, which:

1. Creates an actor system (the `ActorSystem(...)` line).
2. Creates an actor within the system (the `system.actorOf(...)` line, which returns an actor reference for the created actor).
3. Uses the actor reference to send the actor a message (the `hello ! "Bob"` line).
4. Waits one second and then shuts down the actor system (the `system shutdown` line).

The `system.actorOf(Props[Hello])` call is the recommended way of creating an actor instance, using configuration properties specialized for the `Hello` actor type. For this simple actor (playing a bit part, with a single line of dialogue) there's no configuration information, so the `Props` object has no parameters. If you want to set a configuration on your actor, you can define a `Props` class specifically for that actor that includes all the necessary information. (A later example shows how to do this.)

The `hello ! "Bob"` statement sends a message (in this case, just the string `Bob`) to the created actor. The `!` operator is a convenient way to represent sending a message to an actor in Akka, with a fire-and-forget pattern. If you don't like the specialized operator style, you can instead use the `tell()` method to do the same thing.

The second chunk of code is the `Hello` actor definition, starting with `class Hello extends Actor`. This particular actor definition is about as simple as possible. It defines the required (for all actors) partial function `receive`, which implements the handling of incoming messages. (`receive` is a *partial function* because it is only defined for some inputs — in this case, only `String` message inputs.) The handling implemented for this actor is to print a greeting using the message value any time a `String` message is received.

Hello in Java

Listing 2 shows the Listing 1 Akka Hello in ordinary Java.

Listing 2. Hello in Java

```
import akka.actor.*;

public class Hello1 {

    public static void main(String[] args) {
        ActorSystem system = ActorSystem.create("actor-demo-java");
        ActorRef hello = system.actorOf(Props.create(Hello.class));
        hello.tell("Bob", ActorRef.noSender());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) { /* ignore */ }
        system.shutdown();
    }

    private static class Hello extends UntypedActor {

        public void onReceive(Object message) throws Exception {
            if (message instanceof String) {
                System.out.println("Hello " + message);
            }
        }
    }
}
```

Listing 3 shows the actor definition from the Java 8 version with lambdas, along with the required import for the lambda-supporting `ReceiveBuilder` class. Listing 3 is perhaps a little more compact but otherwise much the same as Listing 2.

Listing 3. Java 8 version of the Akka Hello

```
import akka.japi.pf.ReceiveBuilder;
...
    private static class Hello extends AbstractActor {

        public Hello() {
            receive(ReceiveBuilder.
                match(String.class, s -> { System.out.println("Hello " + s); }).
                build());
        }
    }
```

Compared to Listing 2, the Listing 3 Java 8 code uses a different base class —`AbstractActor` instead of `UntypedActor`— and also uses a different way of defining the message-processing alternatives. The `ReceiveBuilder` class enables you to use lambda expressions to define the processing of messages, with a somewhat Scala-like match syntax. If you mostly develop in Scala, this technique might help your Java Akka code look a little cleaner, but otherwise the benefits of using the Java 8-specific version seem pretty minor.

Why wait?

The main application code includes a wait in the form of `Thread sleep 1000` after sending a message to the actor, before shutting down the system. You might wonder why this is necessary. After all, the message is trivial to process; won't it go through to the actor immediately and be processed by the time the `hello ! "Bob"` statement is completed?

The short answer to this question is no. Akka actors run asynchronously, so even if the target actor is located within the same JVM as the sender, execution of the target actor never takes place immediately. Instead, the thread executing the message send adds the message to the target actor's mailbox. Adding a message to the mailbox in turn triggers the execution of a thread to take the message out of the mailbox and process it by invoking the actor's `receive` method. But the thread that takes the message out of the mailbox is generally not the same one that added the message to the mailbox.

Message-delivery timing and guarantees

Beyond the short answer to the "Why wait?" question is a deeper principle. Akka supports actor remoting with location transparency, meaning that your code doesn't have any direct way of knowing whether a particular actor is located within the same JVM or running in a system somewhere out in the cloud. But these two cases will obviously have very different characteristics in actual operation.

> *“ Akka doesn't guarantee that messages will be delivered. The philosophical rationale underlying this delivery nonguarantee goes to one of Akka's core principles. ”*

One difference concerns message loss. Akka doesn't guarantee that messages will be delivered, which might come as a surprise to developers accustomed to messaging systems that are used to connect applications. The philosophical rationale underlying this delivery nonguarantee goes to one of Akka's core principles: designing for failure. As a deliberate oversimplification, consider that delivery guarantees add substantial complexity to message-transfer systems, but nevertheless these more-complex systems sometimes don't work as expected, and the application code must be involved in recovery. It makes sense, then, for the application code to handle the delivery-failure case itself all the time, enabling the message-transfer system to be kept simple.

Akka *does* guarantee that messages will be delivered at most once, and that messages sent from one actor instance to another actor instance will never be received out of order. The second part of the guarantee applies only to particular pairs of actors, and it's not associative. If actor A sends messages to actor B, those messages will never be out of order. The same is true if actor A sends messages to actor C. But if actor B also sends messages to actor C (for example, by forwarding the messages from A on to C), B's messages might be out of order with respect to the messages from A.

In the Listing 1 code, the possibility of message loss is pretty remote, because the code runs in a single JVM and doesn't generate a heavy message load. (A heavy message load can lead to message loss. If Akka runs out of space to store messages, for instance, it has no alternative but to discard them.) But the Listing 1 code is still structured to not make any assumptions about message-delivery timing and to allow for the asynchronous operation of the actor system.

Actors and state

Akka's actor model is flexible and allows for all types of actors. You can have actors with no state information (as in the `Hello1` example), but these actors tend to be the equivalent of method calls. Adding state information allows for much more flexible actor functions.

Listing 1 gives a complete (albeit trivial) example of an actor system — but with an actor constrained to always doing the same exact thing, over and over again. Even actors get bored just repeating the same line, so Listing 4 makes things a bit more interesting by adding some state information to the actor.

Listing 4. Polyglot Scala hello

```
object Hello2 extends App {

  case class Greeting(greet: String)
  case class Greet(name: String)

  val system = ActorSystem("actor-demo-scala")
  val hello = system.actorOf(Props[Hello], "hello")
  hello ! Greeting("Hello")
  hello ! Greet("Bob")
  hello ! Greet("Alice")
  hello ! Greeting("Hola")
  hello ! Greet("Alice")
  hello ! Greet("Bob")
  Thread sleep 1000
  system shutdown

  class Hello extends Actor {
    var greeting = ""
    def receive = {
      case Greeting(greet) => greeting = greet
      case Greet(name) => println(s"$greeting $name")
    }
  }
}
```

The Listing 4 actor knows how to handle two different types of messages, defined near the start of the listing: the `Greeting` message and the `Greet` message, each of which wraps a string value. When the modified `Hello` actor receives a `Greeting` message, it saves the wrapped string as the `greeting` value. When it receives a `Greet` message, it combines that saved greeting with the `Greet` string to form the complete message. Here's what you see (though not necessarily in this order, because the actor execution order is indeterminate) printed to the console when you run this application:

```
Hello Bob
Hello Alice
Hola Alice
Hola Bob
```

Not much is new in the Listing 4 code, so I'm not including the Java versions here. You'll find them in the code download (see Related topics) as `com.sosnoski.concur.article5java.Hello2` and `com.sosnoski.concur.article5java8.Hello2`.

Properties and interactions

Real actor systems get things done by using multiple actors, which interact by sending messages to one another. These actors also often need to be supplied configuration information to prepare for their specific roles. Listing 5 builds on the techniques used in the Hello examples to show a simple version of actor configuration and interactions.

Listing 5. Actor properties and interactions

```
object Hello3 extends App {

  import Greeter._
  val system = ActorSystem("actor-demo-scala")
  val bob = system.actorOf(props("Bob", "Howya doing"))
  val alice = system.actorOf(props("Alice", "Happy to meet you"))
  bob ! Greet(alice)
  alice ! Greet(bob)
  Thread sleep 1000
  system shutdown

  object Greeter {
    case class Greet(peer: ActorRef)
    case object AskName
    case class TellName(name: String)
    def props(name: String, greeting: String) = Props(new Greeter(name, greeting))
  }

  class Greeter(myName: String, greeting: String) extends Actor {
    import Greeter._
    def receive = {
      case Greet(peer) => peer ! AskName
      case AskName => sender ! TellName(myName)
      case TellName(name) => println(s"$greeting, $name")
    }
  }
}
```

Listing 5 features a new actor in a leading role, the `Greeter` actor. `Greeter` goes a step beyond the `Hello2` example, with:

- Properties passed to configure the `Greeter` instances
- A Scala companion object that defines the configuration properties and messages (if you're coming from a Java background, think of the companion object as a static helper class with the same name as the actor class)
- Messages sent between instances of the `Greeter` actor

This code produces simple output:

```
Howya doing, Alice
Happy to meet you, Bob
```

If you try running the code a few times, you'll probably see the lines in reverse order. This ordering is another example of the dynamic nature of the Akka actor system, where the order in which messages are processed is not determinant (with the few important exceptions that I discussed in "Message-delivery timing and guarantees").

`Greeter` in Java

Listing 6 shows the Listing 5 Akka `Greeter` code in ordinary Java.

Listing 6. `Greeter` in Java

```
public class Hello3 {

    public static void main(String[] args) {
        ActorSystem system = ActorSystem.create("actor-demo-java");
        ActorRef bob = system.actorOf(Greeter.props("Bob", "Howya doing"));
        ActorRef alice = system.actorOf(Greeter.props("Alice", "Happy to meet you"));
        bob.tell(new Greet(alice), ActorRef.noSender());
        alice.tell(new Greet(bob), ActorRef.noSender());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) { /* ignore */ }
        system.shutdown();
    }

    // messages
    private static class Greet {
        public final ActorRef target;

        public Greet(ActorRef actor) {
            target = actor;
        }
    }

    private static Object AskName = new Object();

    private static class TellName {
        public final String name;

        public TellName(String name) {
            this.name = name;
        }
    }

    // actor implementation
    private static class Greeter extends UntypedActor {
        private final String myName;
        private final String greeting;

        Greeter(String name, String greeting) {
            myName = name;
            this.greeting = greeting;
        }

        public static Props props(String name, String greeting) {
            return Props.create(Greeter.class, name, greeting);
          }

        public void onReceive(Object message) throws Exception {
            if (message instanceof Greet) {
                ((Greet)message).target.tell(AskName, self());
            } else if (message == AskName) {
                sender().tell(new TellName(myName), self());
            } else if (message instanceof TellName) {
                System.out.println(greeting + ", " + ((TellName)message).name);
            }
        }
    }
}
```

Listing 7 shows the Java 8 version with lambdas. Again, this version is a little more compact in the message-handling implementation, but otherwise the same.

Listing 7. Java 8 version

```
import akka.japi.pf.ReceiveBuilder;
...
    private static class Greeter extends AbstractActor {
        private final String myName;
        private final String greeting;

        Greeter(String name, String greeting) {
            myName = name;
            this.greeting = greeting;
            receive(ReceiveBuilder.
                match(Greet.class, g -> { g.target.tell(AskName, self()); }).
                matchEquals(AskName, a -> { sender().tell(new TellName(myName), self()); }).
                match(TellName.class, t -> { System.out.println(greeting + ", " + t.name); }).
                build());
        }

        public static Props props(String name, String greeting) {
            return Props.create(Greeter.class, name, greeting);
          }
    }
```

Passing properties

Akka uses `Props` objects to pass configuration properties to an actor. Each `Props` instance wraps a copy of the constructor arguments needed by the actor class, along with a reference to the class. This information can be passed to the `Props` constructor in two ways. The Listing 5 example passes the constructor for the actor as a pass-by-name parameter to the `Props` constructor. Note that this way doesn't invoke the constructor immediately and pass the result; it passes the constructor call (which might seem strange if you come from a Java background).

The other way to pass actor configuration to the `Props` constructor is by giving the class of the actor as the first parameter and the constructor arguments for the actor as the remaining parameters. For the Listing 5 example, this form of call would be `Props(classOf[Greeter], name, greeting)`.

Whichever form of `Props` constructor you use, the values being passed to the newborn actor need to be serializable, so that the `Props` can be sent over the network, if necessary, to wherever the actor instance will run. In the case of the pass-by-name constructor call, as used in Listing 5, the closure of the call is serialized when it needs to be sent out of the JVM.

The Akka-recommended practice for creating `Props` objects in Scala code is to define a factory method in a companion object, as is done in Listing 5. This technique prevents any possible issues with accidentally closing over the `this` reference to an actor object when you use the pass-by-name constructor call approach to `Props`. The companion object is also a great place to define the messages that the actor will receive, so that all the associated information is in one place. For Java actors, a static constructor method inside the actor class works well, as used in Listing 6.

Actors sending messages

Each of the Listing 5 `Greeter` actors is configured with a name and a greeting, but when it's told to greet another actor it first needs to find out that other actor's name. The `Greeter` actors accomplish this task by sending a separate message to the other actor: an `AskName` message. The `AskName` message doesn't carry any information per se, but the `Greeter` instance receiving it knows to respond with a `TellName` message that carries the name of the `TellName` sender. When the first `Greeter` receives back a `TellName` message, it prints out its greeting.

Each message sent to an actor comes with some additional information provided by Akka, most notably an `ActorRef` for the sender of the message. You can access this sender information at any time during the processing of a message by calling the `sender()` method defined on the actor's base class. The `Greeter` actors use the sender reference in their handling of an `AskName` message, so that they send the `TellName` response to the correct actor.

Akka allows you to send a message on behalf of another actor (a benign form of identity theft), so that the actor receiving the message will see the other actor as the sender. This is often a helpful feature to use in your actor systems, especially for request-response type message exchanges in which you want the response to go somewhere other than to the actor making the request. Messages sent by application code outside of an actor will by default use a special Akka actor called the *deadletter* actor as the sender. The deadletter actor is also used any time a message cannot be delivered to an actor, giving a convenient way to track undeliverable messages in your actor system by turning on the appropriate logging (something I'll cover in the next installment).

Typing actors

You might notice that nowhere in the examples' sequence of messages is there any type information that explicitly shows that the target of the message is a `Greeter` instance. That's generally the case with Akka actors and the messages they exchange. Even the `ActorRef` used to identity the target actor for a message is untyped.

Programming an untyped actor system has practical advantages. You *could* define actor types — say, by the set of the messages they can process — but doing so would be misleading. In Akka, actors can change their behavior (more on this in the next installment), so different sets of messages can be appropriate for different actor states. Types also tend to get in the way of the elegant simplicity of the actor model, which treats all actors as having at least the potential to handle any message.

Still, Akka does offer typed actor support for when you really want to use it. This support is mostly useful for when you're interfacing between actors and nonactor code. You can define an interface that can be used by the nonactor code to work with the actor, making the actor look more like a normal program component. For most purposes, this is probably more trouble than it's worth, considering how easy it is to send messages directly to an actor even from outside the actor system (as you can see from any of the sample applications so far, in which the nonactor code has been sending messages), but it's great to have the option available.

Messages and mutability

Akka wants you to be certain that you don't accidentally share mutable data between actors. If you did so, the results could be very bad — not quite as bad as crossing the streams of your proton packs when fighting ghosts (*Ghostbusters* reference, if you're among the uninitiated), but still very bad. The problem with sharing mutable data is that actors run in separate threads. If you share mutable data between actors, there's no coordination between the threads running the actors, so they won't see what the other threads are doing and might step on one another in many different ways. The problems get even worse if you're running a distributed system, where each actor will have its own copy of the mutable data.

So messages must be immutable, and not just at the surface level. If any objects are part of the message data, those objects must also be immutable, and so on all the way down across the closure of everything referenced from the message. Akka can't enforce this requirement at present, but the Akka developers want to put restrictions in place at some point in the future. If you want your code to stay usable with future versions of Akka, you must heed this requirement now.

Asking versus telling

The Listing 5 code uses the standard `tell` operation for sending messages. With Akka, you can also use an `ask` message pattern, as an auxiliary operation. The `ask` operation (shown by the `?` operator, or by using the `ask` function) sends a message with a `Future` for the response. Listing 8 shows the Listing 5 code restructured to use `ask` instead of `tell`.

Listing 8. Using `ask`

```
import scala.concurrent.duration._
import akka.actor._
import akka.util._
import akka.pattern.ask

object Hello4 extends App {

  import Greeter._
  val system = ActorSystem("actor-demo-scala")
  val bob = system.actorOf(props("Bob", "Howya doing"))
  val alice = system.actorOf(props("Alice", "Happy to meet you"))
  bob ! Greet(alice)
  alice ! Greet(bob)
  Thread sleep 1000
  system shutdown

  object Greeter {
    case class Greet(peer: ActorRef)
    case object AskName
    def props(name: String, greeting: String) = Props(new Greeter(name, greeting))
  }

  class Greeter(myName: String, greeting: String) extends Actor {
    import Greeter._
    import system.dispatcher
    implicit val timeout = Timeout(5 seconds)
    def receive = {
      case Greet(peer) => {
        val futureName = peer ? AskName
        futureName.foreach { name => println(s"$greeting, $name") }
```

```
      }
      case AskName => sender ! myName
    }
  }
}
```

In the Listing 8 code, the `TellName` message has been replaced by an `ask`. The future returned by the `ask` operation is of type `Future[Any]`, because the compiler doesn't know anything about the result to be returned. When the future completes, the `foreach` uses the implicit dispatcher defined by the `import system.dispatcher` statement to execute the `println`. If the future doesn't complete with a response message within the allowed timeout (another implicit value, defined as five seconds in this case), it instead completes with a timeout exception.

Behind the scenes, the `ask` pattern creates a specialized one-shot actor that acts as an intermediary in the message exchange. The intermediary gets passed a `Promise` and the message to be sent, along with the destination actor reference. It sends the message, then waits for the expected response message. When the response is received, it fulfills the promise and completes the future used by the original actor.

Using the `ask` approach has some limitations. In particular, to avoid exposing actor state (and potentially causing threading issues) you must make sure that you don't use any mutable state from the actor in code executed when the future completes. In practical terms, it's usually easier to use the `tell` pattern for messages sent between actors. The one case in which the ask pattern is more useful occurs when you have application code (such as the main program that launches the actor system and creates the initial actors) that needs to get a response back from an actor (whether typed or untyped).

Bit-part actors

> *“ Don't hesitate to introduce a new actor to your design whenever it helps you handle asynchronous operations cleanly. ”*

The one-shot actor created by the ask pattern is an example of a good design principle to keep in mind when using Akka. It's often desirable to structure your actor system so that intermediate processing steps are performed by special actors crafted for that specific purpose. One common example is the need to merge different asynchronous results before moving on to the next stage of processing. If you use messages for the different results, you can have an actor collect the results until all are ready, and then fire off the next stage. This is basically a generalization of the one-shot actor used by the ask pattern.

Akka actors are lightweight (roughly 300 to 400 bytes per actor instance, plus whatever storage the actor class uses), so you can safely structure your design to use many actors when appropriate. Using specialized actors helps keep your code simple and easy to understand, which is even more of an advantage in writing concurrent programs than in writing sequential programs. Don't hesitate to introduce a new actor to your design whenever it helps you handle asynchronous operations cleanly.

Intermission

Akka is a powerful system, but Akka and the actor model in general require a different style of programming from straight procedural code. With procedural code, you have a program structure in which all the calls are deterministic and you can view the program's entire call tree. In the actor model, messages are optimistically fired off with no guarantee that they'll ever be delivered, and the sequence in which things happen is often hard to establish. The benefit of the actor model is an easy way of structuring applications for high concurrency and scalability, a point that I'll come back to in later installments.

I hope that this article has given you enough of a taste of Akka to whet your appetite for more. Next time I'll take you deeper into actor systems and actor interactions, including a look at how you can easily track the interactions among actors in your system.

Related topics

- See IBM Bluemix in action: In this demo, David Barnes shows you how to develop, create, and deploy an application in the cloud.
- 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.
- Akka.io: Visit the source for all things Akka, including complete documentation for both Scala and Java applications.
- Scala: Scala is a modern, functional language on the JVM.
- "A Java actor library for parallel execution" (Barry Feigenbaum, developerWorks, May 2012): Read an introduction to the µJavaActors library — a lightweight, Java-based actor package for highly parallel execution in traditional Java applications. You can also view Feigenbaum's accompanying Modernize common concurrency patterns with actors video presentation.