SAMPLE CHAPTER

Akka IN ACTION

Raymond Roestenburg
Rob Bakker
Rob Williams

MANNING

Akka in Action

by Raymond Roestenburg
Rob Bakker
Rob Williams

Chapter 9

brief contents

9 Routing messages

In this chapter

- Using the enterprise integration router pattern
- Scaling with Akka routers
- Building a state-based router with become/unbecome

In the previous chapter we looked at enterprise integration patterns as a way to connect actors to solve a wide range of problems. Yet all those approaches involved processing incoming messages in the same way. But often, you need to handle messages differently.

Routers are essential when you want to scale up or out. For example, when you want to scale up, you need multiple instances of the same task, and routers will decide which instance will process the received message. We'll start this chapter by describing the enterprise router pattern and examine the three reasons for using routing to control message flow:

- Performance
- Message content
- State

We'll then show you how to create routing processes for each of these patterns.

If performance or scaling is why you need to turn to a routing solution, you should use Akka's built-in routers, because they're optimized. Concerns with a message's content or state, on the other hand, will point you to using normal actors.

9.1 The enterprise integration router pattern

First, we'll introduce you to the pattern generally—when it applies and how—before we get down to the matter of each specific router implementation. When we move on to implementation, we'll start with the commonly known pattern for routing different messages through a needed set of steps. Let's take a look at the speeding ticket example we introduced earlier. This time, we'll send the messages to the cleanup task or to the next step, depending on the speed of the vehicle in question. When the speed is lower than the maximum allowed speed, the message has to be sent to the cleaning step (instead of just discarding it). But when the speed is higher than the speed limit, it's a violation, and the message should be processed normally. To solve this problem, the router pattern is used. As figure 9.1 shows, the router is able to send messages to different flows.

There are many different reasons to construct logic that makes a decision about where to route a message. As we mentioned in the introduction, there are three reasons for controlling message flow in your applications:

- *Performance*—A task takes a lot of time to process, but the messages can be processed in parallel. So the messages should be divided among different instances. In the speeding ticket example, the evaluation of individual drivers can occur in parallel, because all processing logic resides solely within each captured case.
- *Content of the received message*—The message has an attribute (`License`, in our example) and depending on the value it has, the message should go to a different task.
- *State of the router*—For example, when the camera is in standby, all the messages have to go to the cleanup task; otherwise, they should be processed normally.

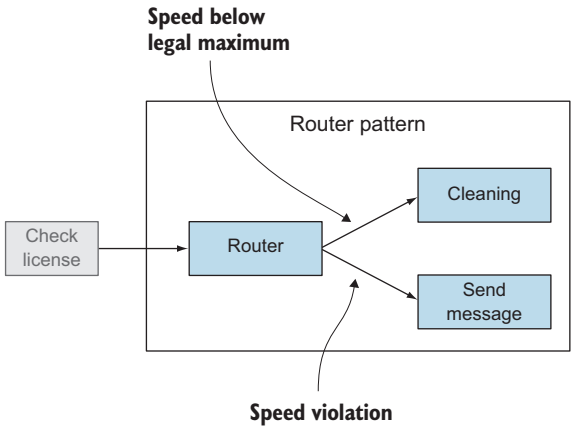

Figure 9.1 Routing logic sending different messages to different process flows

In all cases (no matter what the reason is or the specific logic used), the logic needs to decide which task it should send a message to. The possible tasks a router can choose from are called the *routees* in Akka.

In this chapter we'll show different approaches to routing messages. This will introduce several more Akka mechanisms that are helpful not only when implementing routers, but also in the implementation of your own processes, such as when you want to process messages differently depending on the state of an actor. In section 9.2 we'll start our overview of using routers with an example of a router that makes its decisions based on performance. Scaling is one of the main reasons to use the router functionality of Akka, which is a central component of the overall scaling strategy. In section 9.3 we'll explore routing using normal actors, when message content and state are the key concerns, and we'll show other approaches that use normal actors.

9.2 Balance load using Akka routers

One of the reasons to use a router is to balance the load over different actors, to improve the performance of the system when processing a lot of messages. This can be local actors (scale up) or even actors on remote servers (scale out). Part of the core Akka argument for using scaling is easy routing.

In our camera example, the recognize step takes a relatively long time to process. To be able to parallelize this task, we use a router.

In figure 9.2 you see that the router is able to send the message to one of the `GetLicense` instances. When a message is received by the router, the router picks one of the available processes and sends the message only to that process. When the next message is received, the router picks another process to handle it.

To implement this router, we'll use the built-in router functionality of Akka. In Akka a separation is made between the router, which contains the routing logic, and the actor that represents the router. The router logic decides which routee is selected and can be used within an actor to make a selection. The router actor is a self-contained actor that

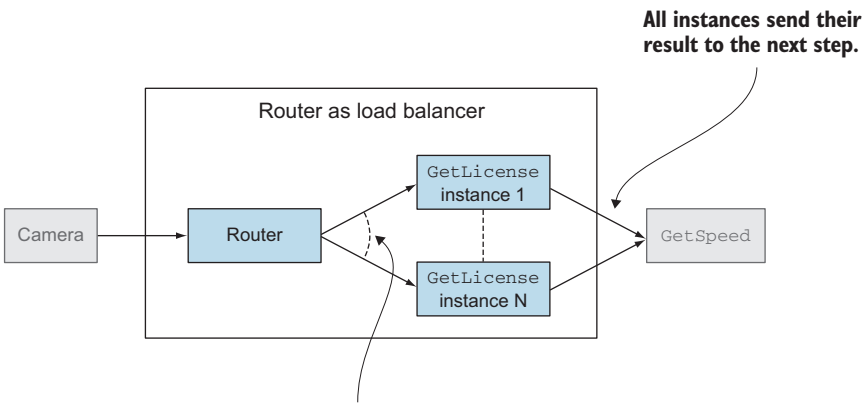

Figure 9.2 Router as load balancer

loads the routing logic and other settings from configuration and is able to manage the routees itself.

The built-in routers come in two varieties:

- *Pool*—These routers manage the routees. They're responsible for creating the routees and removing them from the list if they terminate. A pool can be used when all the routees are created and distributed the same way and there isn't a need for special recovery of the routees.
- *Group*—The group routers don't manage the routees. The routees have to be created by the system and the group router will use the actor selection to find the routees. Group routers also don't watch routees. All the routee management has to be implemented somewhere else within the system. A group can be used when you need to control the routees' lifecycles in a special way or want to have more control over where the routees are instantiated (on which instances).

The most pronounced difference in routers is that the pool router is simplest, as it provides management (throughout the routee lifecycle), but it comes at the cost of having no capacity to customize those behaviors by making routees contain the needed logic.

In figure 9.3 we show the actor hierarchy of the routees and see the difference between using a pool and the group router. The routees are children of the router, and when using the group, the routees can be a child of any other actor (in this example, the `RouteeCreator`). The routees don't need to have the same parent. They just need to be up and running.

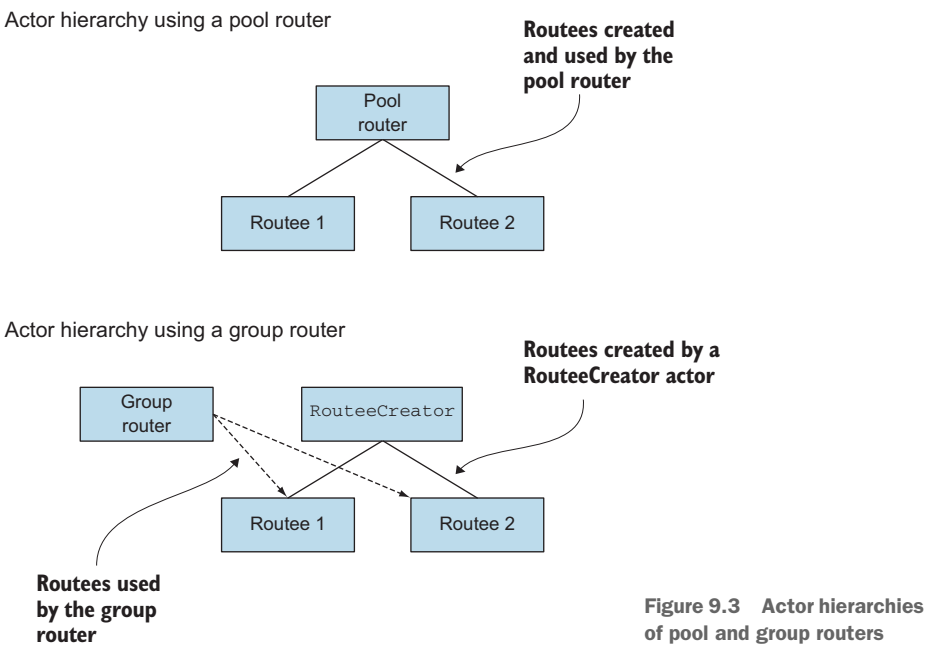

Figure 9.3 Actor hierarchies of pool and group routers

Akka has several built-in routers, summarized in table 9.1. This table shows the router logic and the associated pool and group that use the logic.

Table 9.1 List of available routers within Akka

Logic	Pool	Group	Description
`RoundRobinRoutingLogic`	`RoundRobinPool`	`RoundRobinGroup`	This logic sends the first received message to the first routee, the next message to the second routee, and so on. When all routees have gotten one message, the first routee gets the next, and so forth.
`RandomRoutingLogic`	`RandomPool`	`RandomGroup`	This logic sends every received message to a randomly chosen routee.
`SmallestMailboxRoutingLogic`	`SmallestMailboxPool`	Not available	This router checks the mailboxes of the routees and chooses the routee with the smallest mailbox. The group version isn't available because it uses the select actor functionality internally, and the mailbox size isn't available using these references.
Not available	`BalancingPool`	Not available	This router distributes the messages to the idle routees. It does this internally differently than the other routers. It uses one mailbox for all the routees. The router is able to do this by using a special dispatcher for the routees. This is also the reason that only the pool version is available.
`BroadcastRoutingLogic`	`BroadcastPool`	`BroadcastGroup`	Sends the received messages to all the routees. This is not a router as defined in the enterprise integration pattern, but it implements the recipient list.
`ScatterGatherFirstCompletedRoutingLogic`	`ScatterGatherFirstCompletedPool`	`ScatterGatherFirstCompletedGroup`	This router sends the message to all the routees and sends the first response to the original sender. Technically, this is a scatter-gather pattern using competing tasks that select the best result, in this case the fastest response.
`ConsistentHashingRoutingLogic`	`ConsistentHashingPool`	`ConsistentHashingGroup`	This router uses consistent hashing of the message to select a routee. This can be used when you need different messages to be routed to the same routee, but it doesn't matter which routee.

In section 9.2.1 we show a number of different ways to use these routers. The same logical requirements can be fulfilled using either type (the differences, as discussed earlier, are implementation-specific). We'll use the most common router logic—

round-robin, balancing, and consistent hashing—in the next sections. First, we'll start in section 9.2.1 with a pool router, specifically a `BalancingPool` router, because it has some special behavior. In section 9.2.2 you'll learn about the use of a group router; we'll use the `RoundRobinGroup` router for that. We'll end with an example that uses the `ConsistentHashingPool` in section 9.2.3 to explain when and how this router can be used.

9.2.1 Akka pool router

You've seen that several routers are available, and they come in three flavors: the logic to be used in your own actor, a group actor, or a pool actor. We'll start by showing how to use a pool router. When using the pool actor, you don't have to create or manage the routees; that's done by the pool router. A pool can be used when all the routees are created and distributed the same way and there isn't a need for special recovery of the routees. So for "simple" routees, a pool is a good choice.

CREATING A POOL ROUTER

Using a pool router is simple, and is the same for all the different pool routers. There are two different ways to use the pool: using the configuration, or configuring it within the code. We'll start with the configuration, because using it allows you to change the logic used by the router, which isn't possible when configuring the router in code. Let's use a `BalancingPool` for our `GetLicense` actor.

We have to create the router in the code. The router is also an `ActorRef`, which we can use to send our messages.

Listing 9.1 Creating a router using the configuration file

```
val router = system.actorOf(                    <-- Defines router using configuration
  FromConfig.props(Props(new GetLicense(endProbe.ref))),   <-- How router should create routees
  "poolRouter"                                  <-- Name of router
)
```

This is all you have to do in your code to create a configuration-specified router. But we're not completely done. We have to configure our router.

Listing 9.2 Configuring the router

```
akka.actor.deployment {
  /poolRouter {                          <-- Full name of router
    router = balancing-pool              <-- Logic used by router
    nr-of-instances = 5                  <-- Number of routees in pool
  }
}
```

Those three lines are enough to configure the router. The first line is the name of the router, and it has to be equal to the name used in the code. In our example we've created the router using `system.actorOf` and have created our router at the top level

of the actor path; therefore, the name is /poolRouter. If we create the router within another actor, for example, with the name `getLicenseBalancer`, the name of the router within the configuration would be /getLicenseBalancer/poolRouter. This is important; otherwise, the configuration wouldn't be found by the Akka framework.

The next line in the configuration defines the logic which that has to be used, in this case, the balancing pool actor. The last line defines how many routees (5) will be created within the pool.

This is all we have to do when we want to use a pool of `GetLicense` actors instead of only one `GetLicense` actor. The only difference in our code using a pool of actors is to insert `FromConfig.props()`. The rest is just the same. Sending messages to one of the `GetLicense` routees is accomplished simply by sending a message to the returned `ActorRef` of the created router:

```
router !  Photo("123xyz", 60)
```

The router decides which routee gets the message to process. We started this section by mentioning that there are two ways to define a router. The second way is less flexible, but we'll show it for completeness. It's also possible to define the same pool router within the code, as shown next.

Listing 9.3 Creating a `BalancingPool` in code

```
val router = system.actorOf(
  BalancingPool(5).props(Props(new GetLicense(endProbe.ref))),
  "poolRouter"
)
```

Creates a BalancingPool with 5 routees

The only difference is that we replaced `FromConfig` with `BalancingPool(5)` and have defined the pool and number of routees in the code directly. This is exactly the same as our prior defined configuration.

When you send messages to the router, the message is normally sent to the routees. But there are some messages that are processed by the router itself. Throughout this section we'll cover most of these messages. But we'll start with the `Kill` and `PoisonPill` messages. These messages aren't sent to the routees, but will be processed by the router. The router will terminate, and when using a pool actor, all the routees will also terminate, due to the parent-child relation.

You've seen that when you send a message to the router, only one routee receives the message, at least for most routers. But it's possible to send one message to all the routees of the router. For this you can use another special message: the `Broadcast` message. When you send this message to the router, the router will send the content of the message to all the routees. `Broadcast` messages work on pool and group routers.

> **NOTE** The only router where the `Broadcast` message doesn't work is the `BalancingPool`. The problem is that all the routees have one and the same mailbox. Let's look at an example of a `BalancingPool` with five instances.

When the router wants to broadcast a message, it tries to send the message to all five routees. Due to the fact that there's only one mailbox, all five messages are placed in the same mailbox. Depending on the load of the different routees, the messages are distributed to the routees, which make the first five requests getting the next message. This will work when the load is equally distributed. But if one routee has a message that takes longer to process than the broadcast message, another routee will process multiple broadcast messages before the busy routee has finished. It's even possible that one routee could get all the broadcast messages and the other four routees none of them. So don't use `Broadcast` in combination with the `BalancingPool`.

REMOTE ROUTEES

In the previous section, the created routees were all local actors, but we mentioned before that it's possible to use routers between multiple servers. Instantiating routees on a remote server isn't hard. You have to wrap your router configuration with the `RemoteRouterConfig` and supply the remote addresses.

Listing 9.4 Wrap configuration in a `RemoteRouterConfig`

```
val addresses = Seq(
  Address("akka.tcp", "GetLicenseSystem", "192.1.1.20", 1234),
  AddressFromURIString("akka.tcp://GetLicenseSystem@192.1.1.21:1234"))

val routerRemote1 = system.actorOf(
  RemoteRouterConfig(FromConfig(), addresses).props(
    Props(new GetLicense(endProbe.ref))), "poolRouter-config")

val routerRemote2 = system.actorOf(
  RemoteRouterConfig(RoundRobinPool(5), addresses).props(
    Props(new GetLicense(endProbe.ref))), "poolRouter-code")
```

Here we show the two examples of constructing an address: using the `Address` class directly or constructing an `Address` from a URI. We also show the two versions of creating a `RouterConfig`. The created pool router will create its routees on the different remote servers. The routees will be deployed in round-robin fashion between the given remote addresses. This way the routees are evenly distributed over the remote servers.

As you can see, it's easy to scale out using routers. All you have to do is use the `RemoteRouterConfig`. There's a similar wrapper that's also able to create routees on several remote servers: `ClusterRouterPool`. This wrapper can be used when you have a cluster (and is described in chapter 14, which is devoted completely to the topic of clustering).

Until now, we've used routers with a static number of routees, but when the load of messages changes a lot, you need to change the number of routees, to get a balanced system. For this you can use a resizer on the pool.

DYNAMICALLY RESIZABLE POOL

When the load changes a lot, you'll want to change the number of routees; when you have too few routees, you'll suffer delays because messages have to wait until a routee is finished. But when you have too many routees, you can waste a lot of resources. In

these cases, it would be nice if you could change the pool size dynamically (depending on the load). This can be done with the resize functionality of the pool.

You can configure the resizer to your needs. You can set upper and lower bounds on the number of routees. When you need to increase or decrease the pool, Akka will do so. All this can be configured when defining the pool.

Listing 9.5 Resizer configuration

```
akka.actor.deployment {
  /poolRouter {
    router = round-robin-pool
    resizer {
      enabled = on

      lower-bound = 1
      upper-bound = 10

      pressure-threshold = 1

      rampup-rate = 0.25

      backoff-threshold = 0.3

      backoff-rate = 0.1

      messages-per-resize = 10
    }
  }
}
```

Turns resizer functionality on

Fewest number of routees the router should ever have

Maximum number of routees the router should ever have.

Defines when a routees is under pressure

How fast you're adding routees

When number of routees should decrease

How fast you're removing routees

How fast you're able to change the size again

The first step is to turn the functionality on. Next you can define the upper and lower bounds (on routees). This is done using the attributes `lower-bound` and `upper-bound`.

The next attributes are used to define when the pool should expand, and by what number of routees.

We'll start with the increasing part. When the router pool is under pressure (load), you need to increase the number of routees. But when is the pool under pressure? The answer is when all the current routees are under pressure, and when a routee is under pressure is defined using the `pressure-threshold` attribute. The value of the attribute defines how many messages should be in the mailbox of the routee before it's considered to be under pressure. For example, when the value is set to `1`, a routee is under pressure when it has at least one message in its mailbox, and when it's set to `3`, the routee needs to have at least three messages in its mailbox. A special case is the value `0`. This means that when the routee is processing a message, it's under pressure. Now that you know when a routee is under pressure, we'll look at how the mechanism of adding routees works.

Let's consider the example of a router pool with five instances and whose pressure threshold is set to `0`. When this router gets messages, it forwards them to the first four routees. At this moment, four routees are busy and one is idle. The first situation shown in figure 9.4 is that upon receiving the fifth message, nothing happens, because the check is done before assigning the message to a routee. And at this point one routee is still idle, which means the pool isn't under pressure yet.

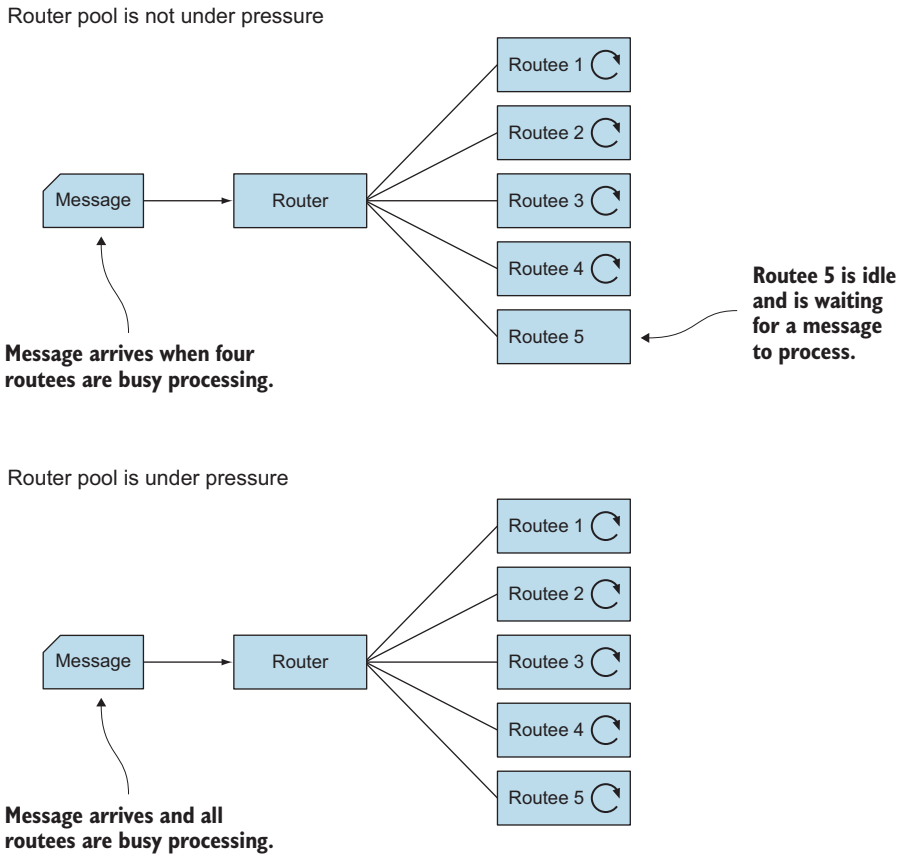

Figure 9.4 A router pool coming under pressure

But when the router receives another message, all the routees are busy processing messages (the second situation in figure 9.4). This means the pool is under pressure at this point, and new routees are added. This resizing isn't done synchronously, because creating a new routee isn't always faster than just waiting for the routee to finish the previous message. In a balanced system, the previous message is probably almost finished. This means that the sixth message isn't routed to the newly created routees, but to one of the already existing routees. But probably the next message can use the newly added routees.

When the pool is under pressure, it adds new routees. The `rampup-rate` defines how many routees are added. The value is a percentage of the total. For example, when you have five routees and the `rampup-rate` is 0.25, the pool will be increased by 25% (rounded up to whole numbers) so the pool will be increased by two routees ($5 \times 0.25 = 1.25$, which is rounded up to 2), resulting in seven routees.

Now that you know how you can increase the pool size, you can also decrease the size. The `backoff-threshold` is the attribute that defines when the router pool will decrease. The back-off won't be triggered until the percentage of routees that are

busy is below the `backoff-threshold`. When you have 10 routees, the back-off is triggered when the percentage of busy routees is below 30%. This means that when only two routees (or fewer) are busy, the number of routees in the pool will decrease.

The number of routees removed is defined by the `backoff-rate` and works just like the `rampup-rate`. In this example with 10 routees and a `backoff-rate` of 0.1, it will decrease by one routee ($10 \times 0.1 = 1$).

The last attribute, `messages-per-resize`, will define the number of messages a router has to receive before it's able to do another resize. This is to prevent a situation where the router is continually increasing or decreasing with every message. This can happen when the load is just between two sizes: the load is too big for one pool size, but when it increases the load is too small, which causes the pool to be adjusted every time. Or when the messages come in batches, this attribute can be used to delay the resize action until the next batch of messages arrives.

SUPERVISION

Another function of the router that should be addressed is supervision. Because the router is creating the routees, it's also the supervisor of the actors. When using the default router, it will always escalate to its own supervisor. This can lead to unexpected results. When one routee fails, the router will escalate to its supervisor. This supervisor probably wants the actor to be restarted, but instead of restarting the routee, the supervisor restarts the router. And the behavior of restarting a router will cause all the routees to be restarted, not only the failing one. The effect is that it looks like the router uses an `AllForOneStrategy`. To solve this issue, we can give the router its own strategy when creating the router.

To set this up, all we need to do is create the strategy and associate it with the router:

```
val myStrategy = SupervisorStrategy.defaultStrategy
val router = system.actorOf(
  RoundRobinPool(5,supervisorStrategy = myStrategy).props(Props[TestSuper]),
  "roundrobinRouter"
)
```

Creates supervisor strategy

Uses supervisor strategy

When one routee is failing, only the failing routee is restarted, and the other routees will proceed without any problem. You can use the default supervisor like in our example, but it's also possible to create a new strategy for the router, or use the strategy of the parent actor of the router. This way all the routees of the given router will behave the same as if they are children of the router's parent.

It's possible to stop the child when there's a failure, but a pool won't spawn a new routee when a routee terminates; it only removes the routee from the pool. When all the routees are terminated, the router will terminate too. Only when you use a resizer will the router not terminate and keep the specified minimum number of routers.

In this section you've seen that router pools are flexible, especially when instantiating the router using the configuration. You're able to change the number of routees and even the router logic. And when you have multiple servers, you can also instantiate routees on different servers, without any complex or difficult constructions.

But sometime the pools are too restricted, and you want more flexibility and control over the creation and management of the routees. These needs can be met by implementing groups.

9.2.2 Akka group router

The pools in the previous section were managing the routees for you. Using the group routers, you have to instantiate the routees yourself. This can be necessary when you want to take control of when and where the routees are created. We'll start with creating our group router. Next, we'll show how we can dynamically change the routees using another set of router messages.

CREATING GROUPS

Creating a group is almost the same as creating a pool. The only difference is that a pool needs the number of routee instances, and a group needs a list of routee paths. Let's start with creating the routees. You don't need to do anything special, but in our example we want one parent `Actor` for all our `GetLicense` actors. We'll introduce a `GetLicenseCreator`, which is responsible for creating the `GetLicense` actors. This actor will be used later to create new routees when one terminates.

Listing 9.6 `GetLicenseCreator` creating our routees

```
class GetLicenseCreator(nrActors: Int) extends Actor {

  override def preStart() {
    super.preStart()
    (0 until nrActors).map { nr =>
      context.actorOf(Props[GetLicense], "GetLicense"+nr)
      system.actorOf(Props( new GetLicenseCreator(2)),"Creator")
    }
  }
  ...
}

system.actorOf(Props( new GetLicenseCreator(2)),"Creator")
```

Creates routees

Creates routee creator

Just as with a pool, there are two ways to create a router group using the configuration and within the code. We'll start with the configuration example.

Listing 9.7 Configuration of the router using a group

```
akka.actor.deployment {
  /groupRouter {
    router = round-robin-group
    routees.paths = [
      "/user/Creator/GetLicense0",
      "/user/Creator/GetLicense1"]
  }
}

val router = system.actorOf(FromConfig.props(), "groupRouter")
```

Full name of router

Group used by router

Actor paths of used routees

Creates a group router

As you can see, the configuration barely differs from the pool configuration; `nr-of-instances` is replaced by `routees.paths`. Creating a group is even easier than creating a pool, because you don't need to specify how the routees are to be created. And because a group uses actor paths, adding remote actors doesn't need any changes. Just add the full path of the remote actor:

```
akka.actor.deployment {
  /groupRouter {
    router = round-robin-group

    routees.paths = [
      "akka.tcp://AkkaSystemName@10.0.0.1:2552/user/Creator/GetLicense0",
      "akka.tcp://AkkaSystemName@10.0.0.2:2552/user/Creator/GetLicense0"]
  }
}
```

Configuring a group with the code is again easy; you only have to supply a list of the routee paths.

Listing 9.8 Creating a group router with code

```
val paths = List("/user/Creator/GetLicense0",
  "/user/Creator/GetLicense1")

val router = system.actorOf(
  RoundRobinGroup(paths).props(), "groupRouter")
```

At this point we can use our router just as we used a router pool. There's one difference: when a routee terminates. When a routee terminates within a pool, the router detects this and removes the routee from the pool. A group router doesn't support this. When a routee terminates, the group router will still send messages to the routee. This is done because the router doesn't manage the routees and it's possible that the actor will be available at some point.

Let's enhance our `GetLicenseCreator` to create a new actor when one child terminates. We'll use the watch functionality described in chapter 4.

Listing 9.9 Creating new actors when routee terminates

```
class GetLicenseCreator(nrActors: Int) extends Actor {

  override def preStart() {
    super.preStart()
    (0 until  nrActors).map(nr => {
      val child = context.actorOf(
        Props(new GetLicense(nextStep)), "GetLicense"+nr)
      context.watch(child)
    })
  }

  def receive = {
    case Terminated(child) => {
      val newChild = context.actorOf(
```

Uses a watch on created routees

Re-creates a routee when one terminates

```
        Props(new GetLicense(nextStep)), child.path.name)
      context.watch(newChild)
    }
  }
}
```

When we use this new `GetLicenseCreator`, the router group can always use the references to the actor without any modification or actions. Let's see this in action. We'll start by creating the routees and then the group, but before we do anything we'll send a `PoisonPill` to all the routees.

Listing 9.10 Testing the `GetLicenseCreator` that manages the routees

```
val endProbe = TestProbe()

val creator = system.actorOf(
  Props(new GetLicenseCreator2(2, endProbe.ref)),"Creator")
val paths = List(
  "/user/Creator/GetLicense0",
  "/user/Creator/GetLicense1")
val router = system.actorOf(
  RoundRobinGroup(paths).props(), "groupRouter")

router ! Broadcast(PoisonPill)
Thread.sleep(100)

val msg = PerformanceRoutingMessage(
        ImageProcessing.createPhotoString(new Date(), 60, "123xyz"),
        None,
        None)

//test if the routees respond
router ! msg
endProbe.expectMsgType[PerformanceRoutingMessage](1 second)
```

Creates routees

Creates router

Kills all routees

If message is received by router before routees are recreated, it will be lost

Tests if new routees will process requests

As you can see, after the routees are killed, the newly created routees will take over and process the incoming messages. `Thread.sleep` is the laziest way to make sure that the `GetLicenseCreator` has re-created the routees. It would be better to publish an event on the event stream once all routees are re-created and subscribe to this event in the test; or add some messages to the `GetLicenseCreator` to inspect the number of re-created routees; or use the `GetRoutees` message described in the next section. This is left as an exercise for the reader.

In this example we created new actors with the same path, but it's also possible to remove or add routees to the group using router messages.

Dynamically resize the router group

We talked already about messages that are processed by the router. Now we'll talk about three other messages for managing the group routees, which enables you to get the routees of a given router and add or remove them:

- *`GetRoutees`*—To get all the current routees, you can send this message to a router, which will reply with a `Routees` message containing the routees.

- *AddRoutee(routee: Routee)*—Sending this message will add the routee to the router. This message takes a `RouteeTrait` containing the new routee.
- *RemoveRoutee(routee: Routee)*—Sending this message will remove the routee from the router.

But using these messages has some pitfalls. These messages and the replies use the `Routee` trait, which contains only one `send` method. This method enables you to send a message directly to a routee. Other functionality isn't supported without converting the `Routee` to an implementation class.

Using the `GetRoutees` message gives you less information than expected, without casting the `Routee` to the actual implementation. The only actual use is to get the number of routees or when you want to bypass the router. This can be handy when you want to send specific messages to specific routees. The last use for this message is to be sure that a router management message is processed, by sending a `GetRoutees` message right after a router message. Subsequently receiving a `Routees` response means that the router message sent before the `GetRoutees` message has been processed. When you receive the reply (`Routees` message), you know that the previous message was also processed.

The add and remove messages need a `Routee`. When you want to add an actor to the router, you need to convert an `ActorRef` or path to a `Routee`.

There are three implementations of the `Routee` trait available within Akka:

- `ActorRefRoutee(ref: ActorRef)`
- `ActorSelectionRoutee(selection: ActorSelection)`
- `SeveralRoutees(routees: immutable.IndexedSeq[Routee])`

Choosing between the three options, we dispense with the last one, `SeveralRoutees`, because it creates a `Routee` from a list of `Routees`. If we add a routee with the first option, `ActorRefRoutee`, the router will create a watch on the new routee. This sounds like it shouldn't be a problem, but when a router receives a `Terminated` message and it isn't the supervisor of the `Routee`, it will throw an `akka.actor.DeathPactException`, which will terminate the router. This is probably not something you want; you should use the second option, the `ActorSelectionRoutee` implementation, to be able to recover from a termination of a routee.

When removing a routee, you have to use the same `Routee` instance type as you used to add the `Routee`. Otherwise, the routee won't be removed. This is why you also need to use the `ActorSelectionRoutee` when removing a routee.

Let's assume we still need the functionality of resizing a group; we'll probably end up with a solution close to listing 9.11. We'll create a `DynamicRouteeSizer`, which will manage the routees and the number used within the group router. We can change the size by sending a `PreferredSize` message.

Listing 9.11 Example of a routee sizer for a group

```
class DynamicRouteeSizer(nrActors: Int,
                         props: Props,
                         router: ActorRef) extends Actor {
  var nrChildren = nrActors
  var childInstanceNr = 0

  //restart children
  override def preStart() {
    super.preStart()
    (0 until  nrChildren).map(nr => createRoutee())
  }

  def createRoutee() {
    childInstanceNr += 1
    val child = context.actorOf(props, "routee" + childInstanceNr)
    val selection = context.actorSelection(child.path)
    router ! AddRoutee(ActorSelectionRoutee(selection))
    context.watch(child)
  }

  def receive = {
    case PreferredSize(size) => {
      if (size < nrChildren) {
        //remove
        context.children.take(nrChildren - size).foreach(ref => {
          val selection = context.actorSelection(ref.path)
          router ! RemoveRoutee(ActorSelectionRoutee(selection))
        })
        router ! GetRoutees
      } else {
        (nrChildren until size).map(nr => createRoutee())
      }
      nrChildren = size
    }
    case routees: Routees => {
      //translate Routees into a actorPath
      import collection.JavaConversions._
      var active = routees.getRoutees.map{
        case x: ActorRefRoutee => x.ref.path.toString
        case x: ActorSelectionRoutee => x.selection.pathString
      }
      //process the routee list
      for(routee <- context.children) {
        val index = active.indexOf(routee.path.toStringWithoutAddress)
        if (index >= 0) {
          active.remove(index)
        } else {
          //Child isn't used anymore by router
          routee ! PoisonPill
        }
      }
      //active contains the terminated routees
      for (terminated <- active) {
        val name = terminated.substring(terminated.lastIndexOf("/")+1)
        val child = context.actorOf(props, name)
```

When starting, creates number of initial requested routees

After creating a new child and adding it to the router using the ActorSelectionRoutee

Changes number of routees

Removes too many routees from the router

Creates new routees

Checks if we can terminate children or need to re-create them after a termination

Translates routee list into a list of actor paths

Child is removed from router; now we can terminate the child

Restarting accidentally terminates children

```
        context.watch(child)
      }
    }
    case Terminated(child) => router ! GetRoutees
  }
}
```

Child has terminated; checks if it was a planned termination by requesting the routees of the router

There's a lot going on here. We start with receiving the `PreferredSize`. There are two options when receiving this message: we have too few or too many routees. When there are too few, we can easily correct this by creating more child actors and adding them to the router. When we have too many, we need to remove them from the router and terminate them. We need to do this in order to prevent the router from sending messages to a killed child actor. This means that we're losing messages. Therefore, we send the `RemoveRoutee` message followed by the `GetRoutees` message. When we get the reply routees, we know that the router won't send any messages to the removed routees, and we can terminate the child actors. We use the `PoisonPill` because we want all previous messages sent to the routee to be processed before stopping it.

Next, we describe the action when a child is terminated. Again there are two possible situations when we get a terminated message. The first one is that we're busy with the downsizing; in this situation we don't have to do anything. In the second situation, an active routee is terminated accidentally. In this case we need to re-create the routee. We want to re-create the child using the same name instead of removing the child from the router and creating a new one, because it's possible that removing a terminated child will cause the router to terminate when that child was the last active routee. To decide what needs to be done, we send a `GetRoutees` message and choose which action needs to be taken when we get the reply.

The last part we need to discuss is what happens if we get the `Routees` reply. We use this message to determine if we can safely terminate a child and if we need to restart a child. To be able to do this, we need the actor paths of the routees, which aren't available in the `Routee` interface. To solve this problem, we use the implementation classes `ActorSelectionRoutee` and `ActorRefRoutee`. The latter class is probably not used within the router, but is added just to be sure. Now that we have a list of actor paths, we can check if we need to stop children or restart them.

To use this sizer, we simply create the router and the sizer actor:

```
val router = system.actorOf(RoundRobinGroup(List()).props(), "router")
val props = Props(new GetLicense(endProbe.ref))
val creator = system.actorOf(
  Props( new DynamicRouteeSizer(2, props, router)),
  "DynamicRouteeSizer"
)
```

As we described in this section, you're able to dynamically change the routees of a group, but it would be preferable to avoid doing so due to the number of pitfalls.

You've learned how to use router pools and groups, but there's one type of router logic that works a little differently than the others, and that is the `ConsistentHashing` router.

9.2.3 ConsistentHashing router

The previous section showed that routers are a great and easy way to be able to scale up and even scale out. But there can be a problem with sending messages to different routees. What happens when you've implemented state in your actor, which relies on the received message? Take, for example, the `Aggregator` of the scatter-gather pattern from section 8.2.4. When you have a router with 10 `Aggregator` routees, each of which joins two related messages into one, there's a good chance that the first message will be sent to routee 1 and the second message to routee 2. When this happens, both aggregators will decide that the message can't be joined. To solve this problem, the `ConsistentHashing` router was introduced.

This router will send similar messages to the same routee. When the second message is received, the router will route it to the same routee as the first one. This enables the `Aggregator` to join the two messages. To make this work, the router must identify when two messages are similar. The `ConsistentHashing` router makes a hash code of the message and maps this to one of its routees. There are several steps to map a message to a routee, which are shown in figure 9.5.

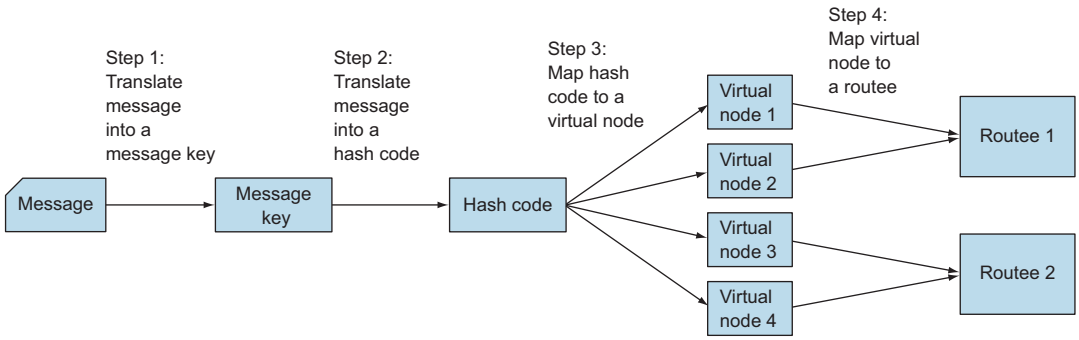

Figure 9.5 Steps the `ConsistentHashing` router follows to decide which routee to select

Step 1 translates a message to a message key object. Similar messages have the same key, for example, the ID of the message. It doesn't matter what type the key is; the only restriction is that this object is always the same for similar messages. This message key is different for every type of message and should be implemented somehow. The router supports three ways to translate the message into a message key:

- *A partial function is specified in the router.*
  This makes the decision specific to that router.
- *The message implements* `akka.routing.ConsistentHashingRouter.ConsistentHashable`.
  This makes the decision specific for the messages used.
- *The messages can be wrapped in a* `akka.routing.ConsistentHashingRouter.ConsistentHashableEnvelope`.
  This makes the decision specific for the sender. The sender knows which key to use.

The last option is less preferable, because this makes the sender closely coupled to the routees. The sender needs to know that the next `ActorRef` is using a `ConsistentHashingRouter` and how to distribute the messages. The other two solutions are much more loosely coupled. Later, we'll show how to use these three different methods.

Step 2 creates a hash code from this message key. This hash code is used to select a virtual node (step 3), and the last step (4) is to select the routee that handles all the messages for that virtual node. The first thing you'll notice is the use of a virtual node. Can't we just map a hash code directly to a routee? Using virtual nodes is done to get a bigger change to equally spread all the messages over the routees. The number of virtual nodes serviced by a routee has to be configured when using a `ConsistentHashingRouter`. In our example we have two virtual nodes for each routee.

Let's take a look at an example of using a routee that will join two messages based on their IDs. We've stripped all the error recovery from this gather example.

Listing 9.12 Joining two message into one

```
trait GatherMessage {
  val id:String
  val values:Seq[String]
}

case class GatherMessageNormalImpl(id:String, values:Seq[String])
  extends GatherMessage

class SimpleGather(nextStep: ActorRef) extends Actor {
  var messages = Map[String, GatherMessage]()
  def receive = {
    case msg: GatherMessage => {
      messages.get(msg.id) match {
        case Some(previous) => {
          //join
          nextStep ! GatherMessageNormalImpl(
                       msg.id,
                       previous.values ++ msg.values)
          messages -= msg.id
        }
        case None => messages += msg.id -> msg
      }
    }
  }
}
```

The `SimpleGather` actor will join two messages with the same ID together into one message. We use a trait as a message type, to be able to use different implementations of the message, which is needed in one of the hashing examples. Let's look at the three ways to specify the message key.

SUPPLY HASHMAPPING PARTIAL FUNCTION TO ROUTER

The first way we'll examine is specifying the hash mapping of the router. When creating the router, you supply a partial function that selects the message key:

```
def hashMapping: ConsistentHashMapping = {
  case msg: GatherMessage => msg.id
}

val router = system.actorOf(
  ConsistentHashingPool(10,
    virtualNodesFactor = 10,
    hashMapping = hashMapping
  ).props(Props(new SimpleGather(endProbe.ref))),
  name = "routerMapping"
)
```

Defines partial hash-mapping function

Sets mapping function

Sets number of virtual hashing nodes per routee

This is all you need to do to use a `ConsistentHashingRouter`. You create a partial function to select a message key from the received message. When you send two messages with the same ID, the router makes sure that both messages are sent to the same routee. Let's try this:

```
router ! GatherMessageNormalImpl("1", Seq("msg1"))
router ! GatherMessageNormalImpl("1", Seq("msg2"))
endProbe.expectMsg(GatherMessageNormalImpl("1",Seq("msg1","msg2")))
```

This method can be used when the router has some specific needs to distribute this message, for example, when you have several routers in the system that are getting the same message type, but you want to use another message key. Suppose one router joins the message based on the ID, and another router counts the message with the same first value and needs the first value as the message key. When the message key is always the same for a given message, it makes more sense to implement the translation within the message.

MESSAGE HAS A HASHMAPPING

It's also possible to translate the message to a key within the message itself by extending the `ConsistentHashable` trait:

```
case class GatherMessageWithHash(id:String, values:Seq[String])
  extends GatherMessage with ConsistentHashable {

  override def consistentHashKey: Any = id
}
```

When using this message, you don't have to supply a mapping function, because the mapping function of the message is used:

```
val router = system.actorOf(
  ConsistentHashingPool(10, virtualNodesFactor = 10)
    .props(Props(new SimpleGather(endProbe.ref))),
  name = "routerMessage"
)

router ! GatherMessageWithHash("1", Seq("msg1"))
router ! GatherMessageWithHash("1", Seq("msg2"))
endProbe.expectMsg(GatherMessageNormalImpl("1",Seq("msg1","msg2")))
```

When the message key is always the same for a given message, this solution is preferable. But we mentioned that we have three ways to get the message key from a message. So let's take a look at the last version: using the `ConsistentHashableEnvelope`.

SENDER HAS A HASHMAPPING

The last method is to supply the message key using a `ConsistentHashableEnvelope` message.

```
val router = system.actorOf(
  ConsistentHashingPool(10, virtualNodesFactor = 10)
    .props(Props(new SimpleGather(endProbe.ref))),
  name = "routerMessage"
)

router ! ConsistentHashableEnvelope(
  message = GatherMessageNormalImpl("1", Seq("msg1")), hashKey = "1")
router ! ConsistentHashableEnvelope(
  message = GatherMessageNormalImpl("1", Seq("msg2")), hashKey = "1")
endProbe.expectMsg(GatherMessageNormalImpl("1",Seq("msg1","msg2")))
```

Instead of sending our message to the router, we send the `ConsistentHashableEnvelope`, which contains our actual message and the `hashKey` to use as the message key. But as we mentioned before, this solution requires that all the senders know that a `ConsistentHashingRouter` is used and what the message key should be. One example of when this method applies is when you need all the messages from one sender to be processed by one routee: then you can use this approach and use a `senderId` as the `hashKey`. But this doesn't mean that each routee processes messages from one sender. It is possible that multiple senders are processed by one routee.

We've shown three different ways to translate a message into a message key, but it is possible to use the three solutions in one router.

In this section, we learned how to use Akka routers, which are used for performance reasons, but remember that routers are also used based on the content of a message or state. In the next section, we describe content and state-based routing approaches.

9.3 Implementing the router pattern using actors

Implementing the router pattern doesn't always require Akka routers. When the decision of the routee is based on the message or some state, it's easier to implement it in a normal actor, because you can leverage all the benefits of actors. You do need to address possible concurrency issues when creating a custom Akka router.

In this section we'll look at some implementations of the router pattern using normal actors. We'll start with a message-based router. In the next section, we'll use the become/unbecome functionality to implement a state-based router. After this, we'll discuss why it's not required for a router pattern to be implemented in a separate actor, but that it's also possible to integrate it into the message-processing actor.

9.3.1 Content-based routing

The most common routing pattern in a system is based on the messages themselves. At the start of section 9.1, we showed an example of a message-based router. When the speed is lower than the speed limit, the driver isn't in violation, and the message need not be processed anymore, but the cleanup step has to be done. When the speed is higher than the speed limit, then it's a violation, and processing should continue to the next step, the send-message task.

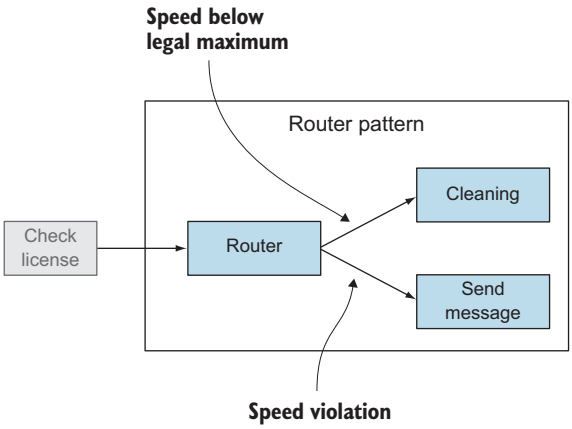

Figure 9.6 Routing based on the value of the speed

Figure 9.6 shows that, based on the content of the message, a flow is chosen. In this example we need to route based on the speed, but it can also be based on the type of the message or any other test done on the message itself. We're not showing an example, because the implementation is very functionality related, and you should be able to implement it with your current knowledge of the Akka framework. In the next section, we'll take a look at state-based routing.

9.3.2 State-based routing

This approach involves changing the routing behavior based on the state of the router. The simplest case would be a switch router that has two states: on and off. When it's in the on state, all messages are sent to the normal flow, and when it's in the off state, the messages are sent to the cleanup flow. To implement this example, we won't use the Akka router, because we want to have state in our router and the Akka router's state is not thread-safe by default. Instead, we'll use a normal actor. It's possible to implement the state as a class attribute, but because it's possible to change the behavior of an actor during its lifecycle, using the become/unbecome functionality, we can employ this as our state representation mechanism.

We'll use the become capability to change the behavior of the actor depending on its state. In our example we have two states, `on` and `off`. When the actor is in the `on` state, the messages should be routed to the normal flow, and when it's in the `off` state, the messages should go to the cleanup process. To do this we'll create two functions to handle the messages. When we want to switch to another state, we'll simply replace the receive function using the become method of the actor's context. In our example we use two messages, `RouteStateOn` and `RouteStateOff`, to change our state and therefore our behavior.

Listing 9.13 State-based router

```
case class RouteStateOn()
case class RouteStateOff()

class SwitchRouter(normalFlow: ActorRef, cleanUp: ActorRef)
  extends Actor with ActorLogging {

  def on: Receive = {
    case RouteStateOn =>
      log.warning("Received on while already in on state")
    case RouteStateOff => context.become(off)
    case msg: AnyRef => {
      normalFlow ! msg
    }
  }
  def off: Receive = {
    case RouteStateOn => context.become(on)
    case RouteStateOff =>
      log.warning("Received off while already in off state")
    case msg: AnyRef => {
      cleanUp ! msg
    }
  }
  def receive = off
}
```

Receives method when state is on

Switches to state off

When state is on, sends message to normal flow

Receives method when state is off

Switches to state on

When state is off, sends message to cleanup

Actor starts with state off

We start with the state `off` because this is the initial function our actor uses. When the actor receives messages, it is routed to the cleanup actor. When we send a `RouteStateOn` to our router, the `become` method is called and replaces the receive function with the `on` implementation of the receive function. All the subsequent messages are then routed to the normal flow actor.

Listing 9.14 Testing state `routerRedirect` actor

```
val normalFlowProbe = TestProbe()
val cleanupProbe = TestProbe()
val router = system.actorOf(
  Props(new SwitchRouter(
    normalFlow = normalFlowProbe.ref,
    cleanUp = cleanupProbe.ref)))

val msg = "message"
router ! msg

cleanupProbe.expectMsg(msg)
normalFlowProbe.expectNoMsg(1 second)

router ! RouteStateOn

router ! msg

cleanupProbe.expectNoMsg(1 second)
normalFlowProbe.expectMsg(msg)

router ! RouteStateOff
router ! msg
```

Switches state to on

Switches state to off

```
    cleanupProbe.expectMsg(msg)
    normalFlowProbe.expectNoMsg(1 second)
```

In our example, we used only the `become` method, but there's also an `unbecome` method. Calling this method causes the new receive function to be removed and the original function to be used. Let's rewrite our router using the `unbecome` method. (It's a semantic difference, but also a matter of following the convention provided.)

Listing 9.15 State-based router using `unbecome`

```
class SwitchRouter2(normalFlow: ActorRef, cleanUp: ActorRef)
  extends Actor with ActorLogging {

  def on: Receive = {
    case RouteStateOn =>
      log.warning("Received on while already in on state")
    case RouteStateOff => context.unbecome()
    case msg: AnyRef => normalFlow ! msg
  }
  def off: Receive = {
    case RouteStateOn => context.become(on)
    case RouteStateOff =>
      log.warning("Received off while already in off state")
    case msg: AnyRef => cleanUp ! msg
  }
  def receive = {
    case msg: AnyRef => off(msg)
  }
}
```

Using the unbecome method instead of become off

There's one warning when using the become functionality: after a restart, the behavior of the actor is also returned to its initial state. The become and unbecome functionality can be handy and powerful when you need to change behavior during the processing of messages.

9.3.3 Router implementations

So far we've shown different routers and how you can implement each. But all these examples are implementing the clean router pattern; no processing is done in the router, it merely directs messages to the appropriate recipients. This is the correct preliminary approach when designing with these patterns, but when you're implementing the processing task and the router component, it can make sense for routing to be subsumed with processing into a single actor, as shown in figure 9.7. This is most likely to make sense when the results of processing will influence what the next step should be.

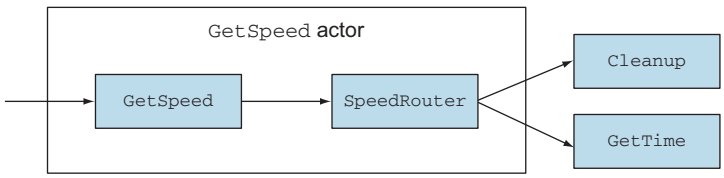

Figure 9.7 Multiple pattern implementation

In our camera example, we have a `GetSpeed` process that finds the speed. When this fails or when the speed is too low, the message has to be sent to the cleanup task; otherwise the message should be sent to the normal flow, in our case the `GetTime` task. To design this we need two patterns:

- A process task
- A router pattern

But when implementing these patterns, it's possible to implement the two components `GetSpeed` and `SpeedRouter` into one actor. The actor starts the processing task first, and depending on the result, it sends the message to either the `GetTime` or the `Cleanup` task. The decision to implement these components into one actor or two depends on the required degree of reusability. When we need the `GetSpeed` to be separate, we can't integrate both steps in one actor. But when the processing actor also has the obligation to render a decision about how the message should be further processed, it would be easier to integrate the two components. Another factor would be that the separation of normal flow and error flow is preferred for the `GetSpeed` component.

9.4 Summary

This chapter was all about how to route messages through different tasks. The Akka routers are an important mechanism for scaling your application, and they're very flexible, especially when using configuration. You've seen that there's different built-in logic that can be used. In this chapter you've also learned the following:

- Akka routers come in two varieties: groups and pools. Pools manage the creation and termination of the routees, and when using the group you have to manage routees yourself.
- Akka routers can easily use remote actors as routees.
- The state-based router we implemented with the become/unbecome mechanism enables us to change the behavior of the actor during its lifecycle by replacing the `receive` method. When using this approach, we have to be careful with restarts, because when the actor `restarts` the receive message is returned to the initial implementation.
- Deciding where to send the message can be based on performance, the content of the received message, or the state of the router.

We've focused on the structure of steps within the application, and how program flow can be modeled using core Akka services. In the next chapter we'll focus on how you send messages between actors, and you'll see that there are more ways to send messages than simply using the actor reference.

SCALA/JAVA

Akka IN ACTION

Roestenburg • Bakker • Williams

Akka makes it relatively easy to build applications in the cloud or on devices with many cores that efficiently use the full capacity of the computing power available. It's a toolkit that provides an actor programming model, a runtime, and required support tools for building scalable applications.

Akka in Action shows you how to build message-oriented systems with Akka. This comprehensive, hands-on tutorial introduces each concept with a working example. You'll start with the big picture of how Akka works, and then quickly build and deploy a fully functional REST service out of actors. You'll explore test-driven development and deploying and scaling fault-tolerant systems. After mastering the basics, you'll discover how to model immutable messages, implement domain models, and apply techniques like event sourcing and CQRS. You'll also find a tutorial on building streaming applications using akka-stream and akka-http. Finally, you'll get practical advice on how to customize and extend your Akka system.

What's Inside

- Getting concurrency right
- Testing and performance tuning
- Clustered and cloud-based applications
- Covers Akka version 2.4

This book assumes that you're comfortable with Java and Scala. No prior experience with Akka required.

A software craftsman and architect, **Raymond Roestenburg** is an Akka committer. **Rob Bakker** specializes in concurrent back-end systems and systems integration. **Rob Williams** has more than 20 years of product development experience.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/akka-in-action

Free eBook
SEE INSERT

"The most readable and up-to-date treatment of Akka I have seen."
—Kevin Esler
TimeTrade Systems

"A great way to get started and go beyond the basics with Akka."
—Andy Hicks
London Scala Users Group

"A user's guide to Akka in the real world!"
—William E. Wheeler
TEKsystems

"A really useful book. Every chapter has working, real-world code that illustrates how to get things done using Akka."
—Iain Starks
Game Account Network

ISBN-13: 978-1-61729-101-2
ISBN-10: 1-61729-101-3
54999
9 781617 291012

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]