Reactor system services

In the earlier sections, we learned that reactors delimit concurrent executions, and that event streams allow routing events within each reactor. This is already a powerful set of abstractions, and we can use reactors and event streams to write all kinds of distributed programs. However, such a model is restricted to reactor computations only. We cannot, for example, start blocking I/O operations, read from a temperature sensor implemented in hardware, wait until a GPU computation completes, or react to temporal events. In some cases, we need to interact with the native capabilities of the OS, or tap into a rich ecosystem of existing libraries. For this purpose, every reactor system has a set of **services**: protocols that relate event streams to the outside world.

In this section, we will take a closer look at various services that are available by default, and also show how to implement custom services and plug them into reactor systems.

The logging service

We start with the simplest possible service called `Log`. This service is used to print logging messages to the standard output. In the following, we create an anonymous reactor that uses the `Log` service. We start by importing the `Log` service:

```
import io.reactors.services.Log
```

Next, we create a reactor system, and start a reactor instance. The reactor invokes the `service` method on the reactor system, which returns the service singleton with the specified type. The reactor then calls the `apply` method on the `log` object to print a message, and seals itself.

This is shown in the following snippet:

```
system.spawn(Reactor[String] { self =>
  val log = system.service[Log]
  log("Test reactor started!")
  self.main.seal()
})
```

Running the above snippet prints the timestamped message to the standard output. This example is very simple, but we use it to describe some important properties of services:

- Reactor system's method `service[S]` returns a service of type `S`.
- The service obtained this way is a lazily initialized singleton instance. There exists at most one instance of the service per reactor system, and it is created only after being requested by some reactor.
- Some standard services are eagerly initialized when the reactor system gets created. Such services are usually available as a standalone method on the `ReactorSystem` class. For example, `system.log` is an alternative way to obtain the `Log` service.

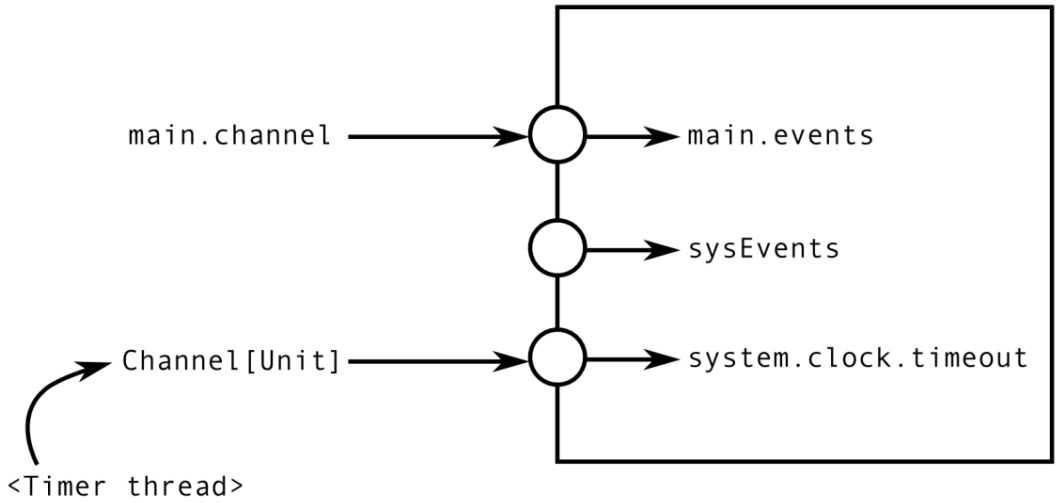
The clock service

Having seen a trivial service example, let's take a look at a more involved service that connects reactors with the outside world of events, namely, the `Clock` service. The `Clock` service is capable of producing time-driven events, for example, timeouts, countdowns, or periodic counting. This service is standard, so it is available by calling either `system.clock` or `system.service[Clock]`.

In the following, we create an anonymous reactor that uses the `Clock` service to create a timeout event after one second. The `timeout` method of the clock service returns an event stream of the `Unit` type that always produces at most one event. We install a callback to the `timeout` event stream, which seals the main channel of this reactor. This is shown in the following snippet:

```
import scala.concurrent.duration._
system.spawn(Reactor[String] { self =>
  system.clock.timeout(1.second) on {
    println("done")
    self.main.seal()
  }
})
```

The `Clock` service uses a separate timer thread under-the-hood, which sends events to the reactor when the timer thread decides it is time to do so. The events are sent on a special channel created by the `timeout` method, so they are seen only on the corresponding event stream combinator. This is summarized in the following figure:



When the main channel gets sealed, the reactor terminates. This is because the `timeout` event stream creates a daemon channel under-the-hood, which does not prevent our anonymous reactor from terminating after non-daemon channels are gone.

The `Clock` service shows a general pattern: when a native entity or an external event needs to communicate with a reactor, it creates a new channel, and then asynchronously sends events to it.

The channels service

Some services provide event streams that work with reactor system internals. The `Channels` service is one such example–it provides an event-driven view over all channels that exist in the current reactor system. This allows polling the channels that are currently available, or waiting until a channel with a specific name becomes available. Awaiting a channel is particularly useful, as it allows easier handling of asynchrony between reactors, which is inherent to distributed systems.

As a side-note, we actually saw and used the `Channels` service earlier, when we opened a second channel in a reactor. The expression `system.channels.open` actually calls the `open` method on the standard channel service. The channels service thus not only allows querying channels that exist in the reactor system, but also creating new channels within existing reactors.

To show basic usage of the `Channels` service, we construct two reactors. The first reactor will create a channel named `"hidden"` after some delay, and the second reactor will wait for that channel. When the channel appears, the second reactor will send an event to that channel. The first reactor prints the string `"event received"` after it receives the message, sealing its main channel. This is shown in the following snippet:

```
val first = Reactor[String] { self =>
  system.clock.timeout(1.second) on {
    val c = system.channels.daemon.named("hidden").open[Int]
    c.events on {
      println("event received")
      self.main.seal()
    }
  }
}
system.spawn(first.withName("first"))
system.spawn(Reactor[String] { self =>
  system.channels.await[Int]("first", "hidden") onEvent { ch =>
    ch ! 7
    self.main.seal()
  }
})
```

In the preceding program, we use the `Clock` service seen earlier to introduce a delay in the first reactor. In the second reactor, we use the `Channels` service to wait for the channel named `"hidden"` of the reactor named `"first"`. Both reactors start at approximately the same time.

After one second, the first reactor uses the `Channels` service to open a new daemon channel named `"hidden"`. The first reactor then installs a callback: when the first event arrives on the hidden channel, it prints a message to the standard output, and the main channel is sealed, to ensure that the reactor terminates. The second reactor gets an event from the `Channels` service, since a channel with the desired name now exists. This reactor sends a value `7` to the hidden channel, and terminates.

To conclude, waiting for channels to appear is important when establishing temporal order in an asynchronous system. In general, the creation of the hidden channel in the first reactor could have been delayed by an arbitrary amount by the reactor system, and the `Channels` service allows the computation to proceed only after specific channels in other reactors get created.

Custom services

Having seen a few existing services, we now show how to create a custom service. To do this, we must implement the `Protocol.Service` trait, which has a single member method called `shutdown`:

```
class CustomService(val system: ReactorSystem)
extends Protocol.Service {
  def shutdown(): Unit = ???
}
```

The `shutdown` method is called when the corresponding reactor system gets shut down, and is used to free any resources that the service potentially has. Any custom service must additionally have a single parameter constructor that takes a `ReactorSystem` object, which allows the service to interact with and use the reactor system during its existence.

As noted earlier, a service is a mechanism that gives access to events that a reactor normally cannot obtain from other reactors. Let's implement a service that notifies a reactor when the enclosing reactor system gets shut down. For this, we will need to keep a map of the channels that subscribed to the shutdown event and a lock to protect access to that state. Finally, we will expose a method `state`, which creates an event stream that emits an event when the reactor system is shut down.

The `state` method will return a special kind of event stream called a `Signal`. The `Signal` type extends the `Events` type, and a signal object emits events whenever its value changes. Additionally, a `Signal` caches the value of the previously emitted event, which can be accessed with the signal's `apply` method. Any event stream can be converted into a signal by calling the `toSignal` method.

The `state` method, called by a specific reactor, must create a new daemon channel called `shut`. This channel is added to the `subscribers` set of the shutdown service. The event stream associated with this channel is converted into a signal with the initial value `false`, and returned to the caller.

The implementation of the `Shutdown` service is shown in the following snippet:

```
class Shutdown(val system: ReactorSystem)
extends Protocol.Service {
  private val subscribers = mutable.Set[Channel[Boolean]]()
  private val lock = new AnyRef
  def state: Signal[Boolean] = {
    val shut = system.channels.daemon.open[Boolean]
    lock.synchronized {
      subscribers += shut.channel
    }
    shut.events.toSignal(false)
  }
  def shutdown() {
    lock.synchronized {
      for (ch <- subscribers) ch ! true
    }
  }
}
```

We can now use the `Shutdown` service in user programs. This is shown in the following snippet:

```
val system = ReactorSystem.default("test-shutdown-system")
system.spawn(Reactor[Unit] { self =>
  system.service[Shutdown].state on {
    println("Releasing important resource.")
    self.main.seal()
  }
})
```

Later, when we shut down the system, we expect that the code in the callback runs and completes the promise:

```
system.shutdown()
```

Note that, when implementing a custom service, we are no longer in the same ballpark as when writing normal reactor code. A service may be invoked by multiple reactors concurrently, and this is why we had to synchronize access to the subscribers map in the `Shutdown` implementation. In general, when implementing a custom service, we have to take care to:

- Never block or acquire a lock in the service constructor
- Ensure that access to shared state of the service is properly synchronized

In conclusion, you should use custom services when you have a native event-driven API that must deliver events to reactors in your program, or wish to expose access to internals of the reactor system, the OS or the underlying hardware. Often the implementation of a reactor system service will employ some lower-level concurrency primitives, but will expose a high-level API that relies on event streams and channels.

Protocols

Reactors, event streams, and channels form the cornerstone of the reactor programming model. These basic primitives allow composing powerful communication abstractions. In this section, we go through some of the basic communication protocols that the Reactors framework implements in terms of its basic primitives. What these protocols have in common is that they are not artificial extensions of the basic model. Rather, they are composed from basic abstractions and other simpler protocols.

We start with one of the simplest protocols, namely the **server-client** protocol. First, we show how to implement a simple server-client protocol ourselves. After that, we show how to use the standard server-client implementation provided by the Reactors framework. In the later sections on protocols, we will not dive into the implementation, but instead immediately show how to use the protocol predefined in the framework.

This approach will serve several purposes. First, you should get an idea of how to implement a communication pattern using event streams and channels. Second, you will see that there is more than one way to implement a protocol and expose it to clients. Finally, you will see how protocols are structured and exposed in the Reactors framework.

Custom server-client protocol

In this subsection, we implement the server-client protocol ourselves. Before we start, we have to create a default reactor system:

```
val system = ReactorSystem.default("system")
```

Let's now consider the server-client protocol more closely. This protocol proceeds as follows: first, the client sends a request value to the server. Then, the server uses the request to compute a response value and send it to the client. But to do that, the server needs a response channel, which serves as the destination to send the response value to. This means that the client must not only send the request value to the server, but also send a channel used for the reply. The request sent by the client is thus a tuple with a value and the reply channel. The server channel used by the server must accept such tuples. We capture these relationships with the following two types:

```
type Req[T, S] = (T, Channel[S])
type Server[T, S] = Channel[Req[T, S]]
```

Here, `T` is the type of the request value, and `S` is the type of the response value. The `Req` type represents the request: a tuple of the request value `T` and the reply channel for responses of type `S`. The `Server` type is then just a channel that accepts request objects.

Next, we ask ourselves–how do we create a `Server` channel? There are several requirements that a factory method for the `Server` channel should satisfy. First, the server method should be generic in the request and the response type. Second, it should be generic in how the request type is mapped to the response type. Third, when a request is sent to the server, the mapped response should be sent back to the server. Putting these requirements together, we arrive at the following implementation of the `server` method, which instantiates a new server:

```
def server[T, S](f: T => S): Server[T, S] = {
  val c = system.channels.open[Req[T, S]]
  c.events onMatch {
    case (x, reply) => reply ! f(x)
  }
  c.channel
}
```

The `server` method starts by creating a connector for `Req[T, S]` type. It then adds a callback to the event stream of the newly created connector. The callback decomposes the request tuple into the request value `x` of type `T` and the `reply` channel, then maps the input value using the specified mapping function `f`, and finally sends the mapped value of type `S` back along the `reply` channel. The `server` method returns the channel associated with this connector. We can use this method to start a server that maps request strings to uppercase strings, as follows:

```
val proto = Reactor[Unit] { self =>
  val s = server[String, String](_.toUpperCase)
}
system.spawn(proto)
```

Next, we will implement the client protocol. We will define a new method called `?` on the `Channel` type, which sends the request to the server. This method cannot immediately return the server's response, because the response arrives asynchronously. Instead, method `?` must return an event stream with the server's reply. So, the `?` method must create a reply channel, send the `Req` object to the server, and then return the event stream associated with the reply channel. This is shown in the following snippet:

```
implicit class ServerOps[T, S: Arrayable](val s: Server[T, S]) {
  def ?(x: T): Events[S] = {
    val reply = system.channels.daemon.open[S]
    s ! (x, reply.channel)
    reply.events
  }
}
```

In the code above, we defined an extension method `?` for objects of the `Server` type by declaring an implicit class `ServerOps`. The `Arrayable` context bound on type `S` is required in the Reactors framework to enable the creation of arrays. The Reactors framework requires the `Arrayable` type class whenever we want to open a channel of a generic type, which is in this case the type `S`.

We now show the interaction between the server and the client by instantiating the two protocols within the same reactor. The server just returns an uppercase version of the input string, while the client sends the request with the content `"hello"`, and prints the response to the standard output. This is shown in the following snippet:

```
val serverClient = Reactor[Unit] { self =>
  val s = server[String, String](_.toUpperCase)

  (s ? "hello") onEvent { upper =>
    println(upper)
  }
}
system.spawn(serverClient)
```

Our implementation works, but it is not very useful to start the server-client protocol inside a single reactor. Normally, the server and the client are separated by the network, or are at least different reactors running inside the same reactor system.

It turns out that, with our toy implementation of the server-client protocol, it is not straightforward to instantiate the protocol in two different reactors. The main reason for this is that once the server channel is instantiated within one reactor, we have no way of *seeing* it in another reactor. The server channel is hidden inside the lexical scope of the server reactor. We will see how to easily overcome this problem with the standard server-client implementation that the Reactors framework provides.

Standard server-client protocol

We have just seen an example implementation of the server-client protocol, which relies only on the basic primitives provided by the Reactors framework. However, the implementation that was presented is very simplistic, and it ignores several important concerns. For example, how do we stop the server protocol? Also, we instantiated the server-client protocol in a single reactor, but is it possible to instantiate server-client in two different reactors?

In this section, we take a closer look at how the server-client protocol is exposed in the Reactors framework, and explain how some of the above concerns are addressed. Most predefined protocols can be instantiated in several ways:

- By installing the protocol on the existing connector inside an existing reactor, which has an appropriate type for that protocol. The main benefit of this is that you can install the protocol on, for example, the main channel of a reactor. This also makes the protocol accessible to other reactors that are aware of that respective channel.
- By creating a new connector for the protocol, and then installing the protocol to that connector. The main benefit of this is that you can fully customize the protocol's connector (for example, name it), but you will need to find some way of sharing the protocol's channel with other reactors, for example, by using it on the `Channels` service, or by sending the channel to specific reactors.
- By creating a new `Proto` object for a reactor that exclusively runs a specific protocol. The main benefit of this is being able to fully configure the reactor that you wish to start (for example, specify a scheduler, reactor name, or transport).
- By immediately spawning a reactor that runs a specific protocol. This is the most concise option.

These approaches are mostly equivalent, but they represent different trade-offs between convenience and customization. Let's take a look at the predefined server-client protocol to study these approaches in turn.

Using an existing connector

When using an existing connector, we need to ensure that the connector's type matches the type needed by the protocol. In the case of a server, the connector's event type must be `Server.Req`. In the following, we define a server prototype that multiplies the request integer by `2` to compute a response. To install the server-client protocol, we call the `serve` method on the connector:

```
val proto = Reactor[Server.Req[Int, Int]] { self =>
  self.main.serve(x => x * 2)
}
val server = system.spawn(proto)
```

The client can then query the `server` channel using the `?` operator. For convenience, we use the `spawnLocal` method, which simultaneously defines an anonymous reactor template and uses it to spawn a new client reactor. This is shown in the following snippet:

```
system.spawnLocal[Unit] { self =>
  (server ? 7) onEvent { response =>
    println(response)
  }
}
```

Creating a new connector

Let's say that the main channel is already used for something else. For example, the main channel could be accepting termination requests. Consequently, the main channel cannot be shared with the server protocol, as protocols usually need exclusive ownership of the respective channel. In such cases, we want to create a new connector for the protocol.

This approach is very similar to using an existing connector. The only difference is that we must first create the connector itself, giving us an opportunity to customize it. In particular, we will make the server a `daemon` channel, and we will assign it a specific name `"server"`, so that other reactors can find it. We will name the reactor itself `"Multiplier"`. To create a server connector, we use the convenience method called `server` on the channel builder object, to get a new connector of the appropriate type.

We can then call the `serve` method on the connector to start the protocol. This is shown in the following snippet:

```
val proto = Reactor[String] { self =>
  self.main.events onMatch {
    case "terminate" => self.main.seal()
  }
  self.system.channels.daemon.named("server")
    .server[Int, Int].serve(_ * 2)
}
system.spawn(proto.withName("Multiplier"))
```

The client must now query the name service to find the server channel, and from there on it proceeds as before, as shown in the following:

```
system.spawnLocal[Unit] { self =>
  self.system.channels.await[Server.Req[Int, Int]](
    "Multiplier", "server"
  ) onEvent { server =>
    (server ? 7) onEvent { response =>
      println(response)
    }
  }
}
```

Creating a protocol-specific reactor prototype

When we are sure that the reactor will exist only, or mainly, for the purposes of the server protocol, we can directly create a reactor server. To do this, we use the `server` method on the `Reactor` companion object. The `server` method returns the `Proto` object for the server, which can then be further customized before spawning the reactor. The `server` method takes a user function that is invoked each time a request arrives. This user function takes the state of the server and the request event, and returns the response event. This is shown in the following code snippet:

```
val proto = Reactor.server[Int, Int]((state, x) => x * 2)
val server = system.spawn(proto)

system.spawnLocal[Unit] { self =>
  (server ? 7) onEvent { response =>
    println(response)
  }
}
```

The `state` object for the server contains a `Subscription` object, which allows the users to stop the server if, for example, an unexpected event arrives.

Spawning a protocol-specific reactor directly

Finally, we can immediately start a server reactor, without any customization. This is done by passing a server function to the `server` method on the `ReactorSystem`, as follows:

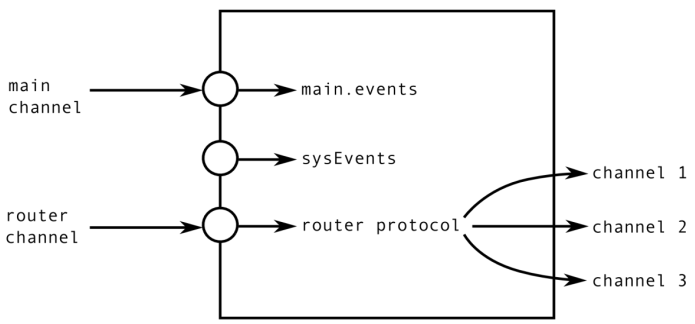
```
val server = system.server[Int, Int]((state, x) => x * 2)

system.spawnLocal[Unit] { self =>
  (server ? 7) onEvent { response => println(response)  }
}
```

In the subsequent sections, we will take a look at some other predefined protocols, which have similar API as the server-client protocol.

Router protocol

In this section, we take a look at a simple router protocol. Here, events coming to a specific channel are routed between a set of target channels, according to some user-specified policy. In practice, there are a number of applications of this protocol, ranging from data replication and sharding, to load-balancing and multicasting. The protocol is illustrated in the following figure:



To show the router protocol in action, we will instantiate a master reactor that will route the incoming requests between two workers. In a real system, requests typically represent workloads, and workers execute computations based on those requests. For simplicity, requests will be just strings, and the workers will just print those strings to the standard output.

As was the case with the server-client protocol, there are several ways to instantiate the router protocol. First, the protocol can be started within an existing reactor, in which case it is just one of the protocols running inside that reactor. Alternatively, the protocol can be started as a standalone reactor, in which case that reactor is dedicated to the router protocol. In our example, we create an instance of the router protocol in an existing reactor.

We first start two workers, called `worker1` and `worker2`. These two reactors will print incoming events to the standard output. We use a shorthand method `spawnLocal`, to concisely start the reactors without creating the `Proto` object:

```
val worker1 = system.spawnLocal[String] { self =>
  self.main.events.onEvent(x => println(s"1: ${x}"))
}
val worker2 = system.spawnLocal[String] { self =>
  self.main.events.onEvent(x => println(s"2: ${x}"))
}
```

Next, we declare a reactor whose main channel takes `Unit` events, since we will not be using the main channel for anything special. Inside that reactor, we first call the `router` method on the `channels` service to open a connector with the appropriate type for the router. By just calling the `router` method, the router protocol does not yet start. We need to call the `route` method on the newly created connector to actually start routing.

The `route` method expects a `Router.Policy` object as an argument. The policy object contains a function that returns a channel for an event that we want to route. This function of type `T => Channel[T]` represents the routing logic for the router protocol.

In our example, we will use the simple round-robin policy. This policy can be instantiated with the `Router.roundRobin` factory method, which expects a list of channels for the round-robin policy, so we will pass a list with `worker1` and `worker2` channels. We show this in the following snippet:

```
system.spawnLocal[Unit] { self =>
  val router = system.channels.daemon.router[String]
    .route(Router.roundRobin(Seq(worker1, worker2)))
  router.channel ! "one"
  router.channel ! "two"
}
```

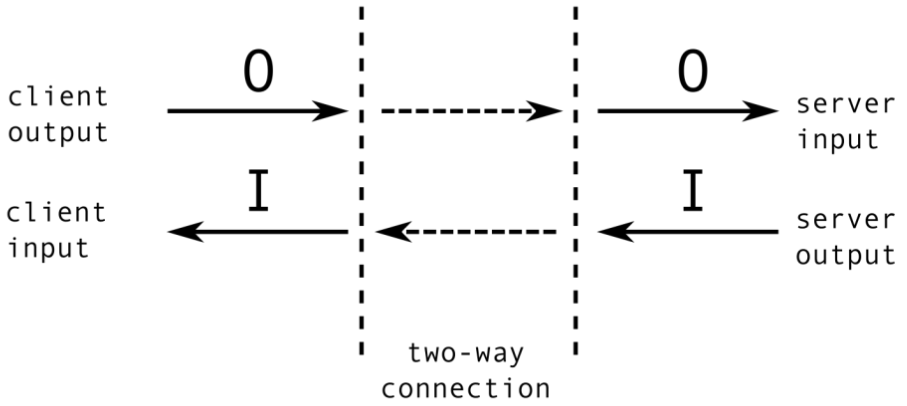
After starting the router protocol and sending the events `"one"` and `"two"` to the router channel, the two strings are delivered to the two different workers. The `roundRobin` policy does not specify which of the target channels is chosen first, so the output can either contain `"1: one"` and `"2: two"`, or `"1: two"` and `"2: one"`.

The round-robin routing policy does not have any knowledge about the two target channels, so it just picks one after another in succession, and then the first one again when it reaches the end of the target list. Effectively, this policy constitutes a very simple form of load-balancing.

There are other predefined policies that can be used with the router protocol. For example, the `Router.random` policy uses a random number generator to route events to different channels, which is more robust in scenarios when a high-load event gets sent periodically. Another policy is `Router.hash`, which computes the hash code of the event, and uses it to find the target channel. If either of these are not satisfactory, `deficitRoundRobin` strategy tracks the expected cost of each event, and biases its routing decisions to balance the total cost sent to each target. Users can also create custom routing policies for other use-cases.

Two-way protocol

In this section, we show a two-way communication protocol. In two-way communication, two parties obtain a connection handle of type `TwoWay`, which allows them to simultaneously send and receive an unlimited number of events until they decide to close this connection. One party initiates the connection, so we call that party the client, and the other party the server. The `TwoWay` type has two type parameters `I` and `O`, which describe the types of input and output events, respectively, from the client's point of view. This is illustrated in the following figure:



Note that these types are reversed depending on whether you are looking at the connection from the server-side or from the client-side. The type of the client-side two-way connection is:

```
val clientTwoWay: TwoWay[In, Out]
```

Whereas the type of the server sees the two-way connection as:

```
val serverTwoWay: TwoWay[Out, In]
```

Accordingly, the `TwoWay` object contains an output channel `output`, and an input event stream `input`. To close the connection, the `TwoWay` object contains a subscription object called `subscription`, which is used to close the connection and free the associated resources.

Let's create an instance of the two-way protocol. This protocol works in two phases. First, a client asks a two-way connection server to establish a two-way connection. After that, the client and the server use the two-way channel to communicate.

In what follows, we declare a reactor, and instantiate a two-way connection server within that reactor. For each established two-way connection, the two-way server will receive strings, and send back the length of those strings.

```
val seeker = Reactor[Unit] { self =>
  val lengthServer = self.system.channels
    .twoWayServer[Int, String].serveTwoWay()
```

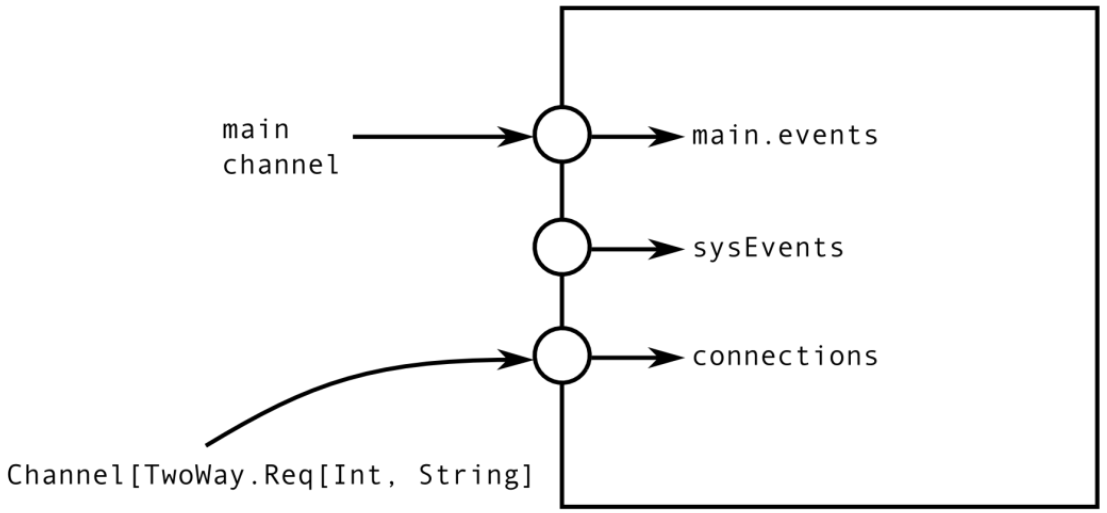
The two lines above declare a reactor `Proto` object, which instantiates a two-way server called `lengthServer`. We first called the `twoWayServer` method on the `Channels` service, and specified the input and the output type (from the point of view of the client). Then, we called the `serverTwoWay` method to start the protocol. In our case, we set the input type `I` to `Int`, meaning that the client will receive integers from the server, and the output type `O` to `String`, meaning that the client will be sending strings to the server.

The resulting object `lengthServer` represents the state of the connection. It contains an event stream called `connections`, which emits an event every time a client requests a connection. If we do nothing with this event stream, the server will remain silent – it will start new connections, but ignore events coming from the clients. How exactly the client and server communicate over the two-way connection (and when to terminate this connection) is up to the user to specify. To customize the two-way communication protocol with our own logic, we need to react to the `TwoWay` events emitted by the `connections` event stream, and install callbacks to the `TwoWay` objects.

In our case, for each incoming two-way connection, we want to react to `input` strings by computing the length of the string, and then sending that length back along the `output` channel. We can do this as follows:

```
lengthServer.connections.onEvent { serverTwoWay =>
  serverTwoWay.input.onEvent { s =>
    serverTwoWay.output ! s.length
  }
}
```

We now have a working instance of the two-way connection server. The current state of the reactor can be illustrated with the following figure, where our new channel appears alongside standard reactor channels:



Next, let's start the client-side part of the protocol. The client must use the two-way server channel to request a connection. The `lengthServer` object that we saw earlier has a field called `channel` that must be used for this purpose. The client must know about this channel to start the connection. Note that only the `channel` must be shared, not the complete `lengthServer` object. To make things simple, we will instantiate the client-side part of the protocol inside the same reactor as the server-side part.

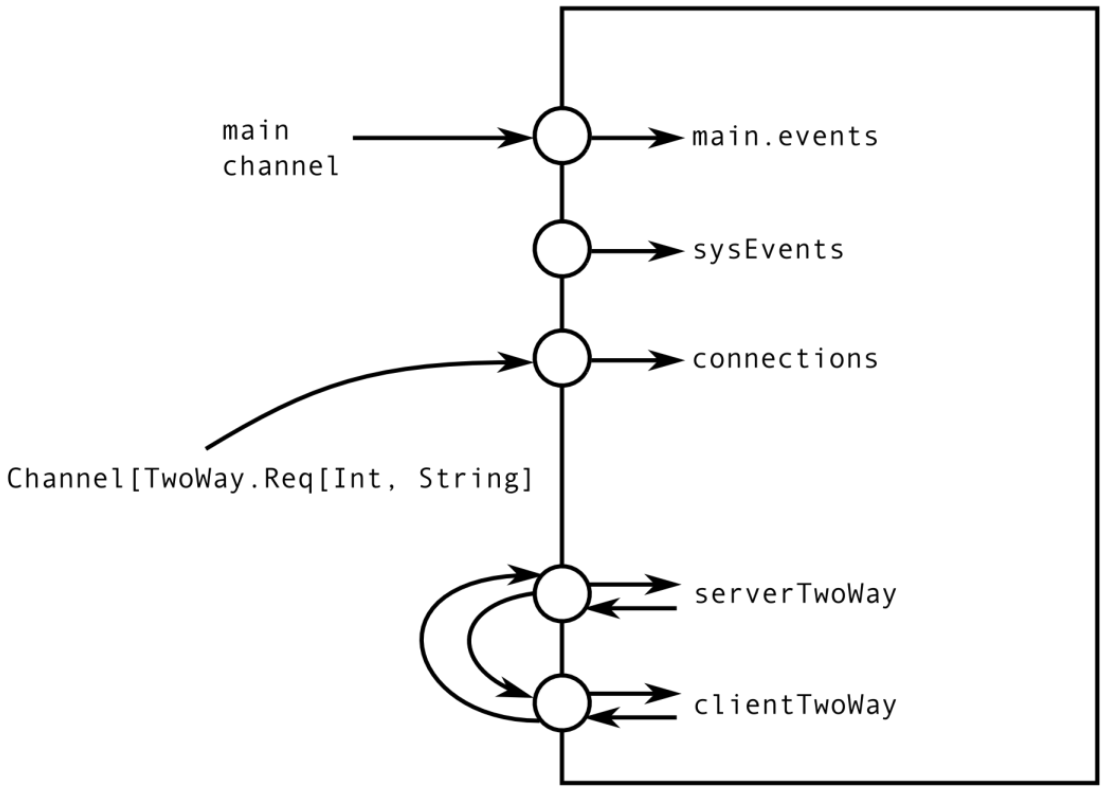
To connect to the server, the client must invoke the `connectTwoWay` extension method on the `channel`. This method is only available when the package `io.reactors.protocol` is imported, and works on two-way server channels. The `connect` method returns an event stream that emits a `TwoWay` object once the connection gets established.

In the following, we connect to the server. Once the server responds, we use the `TwoWay[Int, String]` object to send a string event, and then print the length event that we get back:

```
    lengthServer.channel.connect() onEvent { clientTwoWay =>
      clientTwoWay.output ! "What's my length?"
      clientTwoWay.input onEvent { len =>
        if (len == 17) println("received correct reply")
        else println("reply incorrect: " + len)
      }
    }
  }

  system.spawn(seeker)
```

After the connection is established, the state of the reactor and its connectors is as shown in the following diagram:



Note that, in this case, the two-way channel has both endpoints in the same reactor. This is because we called `twoWayServe` and `connect` in the same reactor, for the purposes of demonstration. In real scenarios, we would typically invoke these two operations on separate reactors.

Summary

In this chapter, we learned about the reactor model, and its implementation in the Reactors framework. We saw how to define and instantiate reactors, compose event streams, customize reactor names and assign schedulers, use reactor system services, and define custom ones. Importantly, we saw how to use a few basic low-level protocols such as the server-client, router, and the two-way connection protocol.

To learn more about reactors, you can find a lot of information on the website of the Reactors framework, at `http://reactors.io`. The Reactors framework is relatively new, but it is under constant development. As the framework matures and gains more features, you will find more and more information on the website. To learn more about the reactor programming model itself, the paper *Reactors, Channels, and Event Streams for Composable Distributed Programming* is worth taking a look at.

Exercises

In the following exercises, you are expected to define several reactor protocols. In some cases, the task is to first investigate a specific algorithm online on your own, and then implement it using the Reactors framework. The exercises are ordered by their difficulty, and range from simple tasks to more complex ones.

1. Define a method called `twice`, which takes a target channel, and returns a channel that forwards every event twice to the target.

   ```
   def twice[T](target: Channel[T]): Channel[T]
   ```

2. Define a method called `throttle`, which throttles the rate at which events are forwarded to the target channel.

   ```
   def throttle[T](target: Channel[T]): Channel[T]
   ```

   Hint: you will have to use the `Clock` service and the functional event stream composition.

3. The `Shutdown` service shown in this chapter can run out of memory if there are a lot of reactors subscribing to it. This is because the current implementation never removes entries from the service's `subscribers` map. Modify the custom `Shutdown` service so that the clients of the `state` signals can unsubscribe from listening to shutdown events. Additionally, ensure that when a reactor terminates, it unsubscribes from the `Shutdown` service if it was subscribed to it. Use the `sysEvents` event stream for this purpose.
4. Assume that normal `Channel` objects can occasionally lose some events or reorder them, but never duplicate or corrupt events. Implement a reliable channel protocol, which ensures that every event sent through a channel is delivered to its destination in the order it was sent. Define two methods `reliableServer` and `openReliable`, which are used to start the reliable connection server and open the reliable connection on the client, respectively. The methods must have the following signatures, where it is up to you to determine the types:

```
def reliableServer[T](): Channel[Reliable.Req[T]]
def openReliable[T]
(s: Channel[Reliable.Req[T]]): Events[Channel[T]]
```

5. Implement the *best-effort broadcast protocol*, which delivers events to multiple targets. The broadcast method must implement the following interface, where events sent to the resulting channel must be forwarded to all the targets:

```
def broadcast(targets: Seq[Channel[T]]): Channel[T]
```

6. Investigate and learn about how the CRDT counter algorithm works. Then, use the best-effort broadcast protocol from an earlier exercise to implement the CRDT counter algorithm. Define a method called `crdt` to allow users to create the CRDT counter.
7. Implement a `failureDetector` method, which takes a heartbeat server of `Unit` request and response types, and returns a `Signal` object that denotes whether the server is suspected to have failed:

```
def failureDetector(s: Server[Unit, Unit]): Signal[Boolean]
```

   The protocol started by this method must regularly send heartbeat signals to the server, and expect replies within a certain time period. The server is suspected to have failed when its response does not arrive before that time period elapses. Implement a unit test to validate that the resulting signal correctly detects server failure.

8. Implement the *reliable broadcast algorithm*, which has the same interface as the best-effort broadcast from an earlier exercise, but guarantees delivery to either all or none of the targets even if the sender dies halfway during the send operation. Implement unit tests to validate the correctness of your implementation.

Index

D

E

F

Q

R

S

T

U

V

W