Scala and Akka: A Lightning Tour

Jeff Chase
Duke University
Draft of 9/7/15.1

The purpose of this note is to help students in my distributed systems class learn enough Akka and Scala to write simple programs with a minimum of hassle. There is a lot of material on the official sites and elsewhere on the Web, so I presume that the reader already knows what Akka and Scala are and why one might want to use them. There are also good books on Scala (e.g., Martin Odersky's book). But if you are already a solid programmer, the best way to come up to speed fast on a new language is to look at a program. This note is a tour of a simple program ("rings") that illustrates the key features of Akka and Scala. As a side bonus it illustrates some concepts for scalable applications using key-value stores.

The reader should recognize that we are just scratching the surface of Akka and Scala. Many features are not addressed here. In particular, we ignore Akka's failure models and the actor hierarchy. For Scala, this treatment ignores exception handling and is light on the technically correct dogma of asynchronous functional programming using *for comprehensions* and the like. But it will get you to a level of proficiency where you can begin to explore these topics.

Rings emulates a two-tier service, with a tier of application servers backed by a key/value store. The app servers and storage servers are Akka actors that interact using messages. A single `LoadMaster` actor at the front end generates a stream of commands to the application tier, and collects statistics. The rings package can act as a framework for implementing key-value applications.

The following sections point out some features in each of the source files, one by one. The best way to understand this document is to read the whole thing in order. Some concepts are introduced a bit vaguely, then explained in more detail later. Also, it will help to have the rings source code at hand. This document includes key code snippets, but some supporting detail is missing.

1 TestHarness

This file uses the `object` keyword to define an object. Like a class, an object definition defines member fields and methods for the object.

```
object TestHarness {
  val numNodes = 10
  val burstSize = 1000
  ...
 }
```

An object defined in this way is instantiated once when the program is launched: it is a static singleton. All of its member fields (variables local to the object) have an initial value. In this case, the `val` keyword indicates that the field never changes after initialization (the field is immutable). We could replace any of the `val` keywords with `var`, indicating that the field is mutable.

Although Scala is strongly typed, it is not always necessary to specify types in the source code. The Scala compiler is smart and it can often infer the types. For example, the types of these member fields are apparent from their initial values. Type inference helps make Scala concise, like a scripting language.

Scala does not require semicolons at the ends of statements. Actually, Scala statements *are* terminated by semicolons, but it is not generally necessary to put them in: the compiler can infer them—almost always.

The `TestHarness` object defines a `main` method:

```
def main(args: Array[String]): Unit = run()
```

The main method returns no value: the `Unit` type in scala is similar to void in C or Java.

If you type "run" to `sbt`, it will run the method named `main` in the current project. The main method may be in any source file. If there is more than one main method, then you must specify which to run, e.g., by saying `run TestHarness.main`. In this case there is only one main method in rings.

The `main` method receives command line arguments as an array. The arguments are strings: the `args` parameter is of type `Array[String]`. In Scala, the square brackets `[]` indicate a type parameter. Array is a *parameterized type*: arrays may be created with elements of any type, and the elements of this particular array (`args`) are of type `String`.

TestHarness illustrates how to print a line of output using an *interpolated string*:

```
println(s"Done in $runtime ms ($throughput Kops/sec)")
...
println(s"Final stats: $done")
```

An interpolated string is marked with an `s` prefix. Any $-marked fields in an interpolated string are expanded with the current values of the corresponding named variables or fields, represented as strings. The string representation of each value is determined by its type. For example, `done` is of type `Stats`, which is a class that defines a `toString` method to generate a string for a `Stats` object. This interpolated string construct splices the result of `Stats.toString` into the output.

Methods and fields are semi-interchangeable in Scala. For example, note that `runUntilDone` is a method, but it can be named without parentheses because it takes no arguments.

1.1 Sending messages to Akka actors

The use of the `system` value in `TestHarness` shows how to initialize an Akka actor system and how to shut it down when the work is done, to allow the program to terminate. The argument is a string name that is used to label log messages and to generate fully-qualified names for the actors.

```
val system = ActorSystem("Rings")
...
system.shutdown()
```

The method `runUntilDone` shows how to send a message to an actor (the master) and also how to wait for a reply when one is expected.

```
def runUntilDone() = {
  val future = ask(master, Join()).mapTo[Stats]
  master ! Start(opsPerNode)
  val done = Await.result(future, 60 seconds)
  println(s"Final stats: $done")
}
```

An Akka program may send any object as a message. In this example we instantiate objects of the `Join` and `Start` classes and send them as messages. Ordinarily, we use Scala's `new` operator to instantiate new objects, but that is not always necessary, as discussed below.

A synchronous request/reply communication pattern is called the *ask* pattern, and is illustrated here by the `Join()` request. We also send a `Start()` message to the master using the *tell* pattern, which sends a message asynchronously and does not expect a reply ("fire and forget").

In this example we use the `!` operator to send the tell message. The tell (`!`) is really just a method call on the master object, passing the message as an argument: Scala allows a method name to be or to include symbols like `!`, and it is legal syntax to invoke a method using infix notation like this. That is how the Akka tell (!) operator works: Akka is an add-on to Scala using only general features of the language. This extensibility is a key design theme of the Scala language.

The `Stats` class discussed below shows a similar example of defining an operator to be invoked using infix notation.

Akka discourages the ask pattern for ideological reasons, but there is really nothing wrong with it except that the caller cannot make any progress or respond to any messages until the reply arrives or the request times out. In this case, that is the behavior we want.

Note that the ask call occurs in two separate steps: sending the request and awaiting the reply. The ask request returns a *future* object for the value of the reply (i.e., the reply message). Futures are the fundamental concurrency primitive in Scala: a future executes an action asynchronously (on another thread) and results in a value only after the action completes. Once a future has completed, we can retrieve its value with a method call on the future object. We can wait on a future object until it completes, as shown by the `Await.result` call here, or we can arrange for some other code to execute (also asynchronously) to operate on the resulting value when the future completes. Because the wait is a separate step, the code can start the request asynchronously and then perform other actions before waiting on the reply. In this example we send a `Join` request to the master to arrange for a reply when the master is finished, then tell the master to `Start`, then wait for it to finish.

Futures are tricky: you may be interested to read about them in the Scala docs, but in this course we will use them in limited ways following a cookbook approach, using this example as a template.

The trickiest part of our use of futures in the ask code is the use of `mapTo` to indicate that the expected reply value will be an instance of the `Stats` class, and to cast the reply value to the correct type. What is happening here is that the `ask` function returns a future of value `Any` (the most general type in Scala), and `mapTo` is a method on a future that returns a new future whose value is of the specified type. The new future completes when the object of the `mapTo` completes, and it succeeds only if the resulting value may be cast to the specified type. The type cast is necessary because Akka actors are dynamic: the type of a message cannot be known (checked) until the message arrives at runtime.

2 Stats

Let's look at `Stats` as a simple example of a class. A Scala class is similar to a class in Java or C++: it defines a template for instantiating multiple objects at runtime, each of which is an instance of the class, with its own copy of the member fields. A class may define methods, which are procedures or functions that operate on instances of the class. A method is invoked on a specific instance/object, and implicitly operates on that instance/object. The current instance may be referred to as `this`, but it is rarely necessary to say that explicitly: unqualified field names in a method implicitly refer to the fields of `this`.

We can instantiate a new `Stats` object using the builtin `new` operator, as in Java. A Scala class has no constructor method: the member fields of each new instance are initialized as stated in the class definition.

```
class Stats {
```

```
  var messages: Int = 0
  var allocated: Int = 0
  ...
}
```

The member fields of a `Stats` object are integer counters with names. The idea is that the application code will use these to tally up how often various events occur in the code, by incrementing a counter each time the corresponding event occurs. Because the values in these member fields will change after initialization, they are labeled as `var` (rather than `val`).

Because these members are not marked `private`, the `Stats` object is really more like a C struct: any code outside of this class with a reference to a `Stats` object can access its counters directly, by their names.

The specific uses of these counters are up to the application (e.g., `RingService`). No other part of the code depends on the counters or their values. The exception is the `messages` field, which the `LoadMaster` (master) object presumes has a specific use, to count the commands received by an application actor. This is a quirk of the example and should probably be fixed.

The methods of `Stats` illustrate some useful features of Scala. The first method (`+=`) has a name that may be used in infix notation as an intuitive operator.

```
  def += (right: Stats): Stats = {
    messages += right.messages  /* number of messages/commands processed by an actor */
    ...
    errors += right.errors
    this
  }
```

Given two `Stats` objects, `s1` and `s2`, `s1 += s2` adds the values of `s2`'s counters into `s1`'s counters. That statement `s1 += s2` is an infix method invocation, invoking the method `+=` on the `Stats` object `s1`, passing a reference to `s2` as an argument. Therefore, inside the method we can refer to `s1` as `this`. A scala method returns the value of its last statement or expression. In this example, the last statement is just `this`: the `+=` method returns a reference to the modified `s1` as its result.

The `toString` method is similar to Java. All Scala objects have a `toString` method, which is invoked to generate a string representing the object. Any class may override the default `toString` method with its own definition that formats the object's contents into a message or string. Here the `toString` method is itself defined as an interpolated string naming the fields of the `Stats` object.

```
  override def toString(): String = {
    s"Stats msgs=$messages ... err=$errors"
  }
```

If a `println` statement includes a `Stats` object, e.g., in an interpolated string (as in the TestHarness example above), this `toString` method is called to generate a string to insert into the output.

3 KVAppService

Let's move on to `KVAppService`. Like TestHarness, `KVAppService` is defined as an `object`: it exists as a single named object at runtime, with no corresponding class.

The `KVAppService.apply` method instantiates some Akka actors and links them together. We instantiate an actor by calling the `actorOf` method on the `ActorSystem` object, which is passed in as an argument, e.g.:

```
val master = system.actorOf(LoadMaster.props(numNodes, servers, ackEach), "LoadMaster")
```

The `actorOf` method takes a `Props` object and a string name for the new actor. We discuss the `Props` objects later. The result of `actorOf` is an `ActorRef` that names the actor: the `ActorRef` is an object that can be passed around by value and used to send messages to the named actor, even if the actor resides on a different machine from the sender.

When the `KVAppService.apply` method completes, there exists a tier of application servers of the desired type, backed by a tier of storage servers of type `KVStore`. The app servers can talk to one another and to the storage servers. A `master` actor is also instantiated, which can send commands to the app servers. The method returns the master object.

3.1 Apply method

The `KVAppService` object defines a single method named `apply`. That is a special method name: the `apply` method of an `object` can be invoked simply by giving the object name and the arguments to its `apply` method. We can see an example in `TestHarness` at the line:

```
val master = KVAppService(system, numNodes, burstSize)
```

That line invokes the `apply` method of the `KVAppService` object with the listed arguments, and places the result (a reference to a freshly instantiated master actor) in the value named `master`.

3.2 Case classes

Now we can understand how Akka messages are defined in this example application. Consider the lines at the top of the file:

```
sealed trait AppServiceAPI
case class Prime() extends AppServiceAPI
case class Command() extends AppServiceAPI
case class View(endpoints: Seq[ActorRef]) extends AppServiceAPI
```

These lines define message types that any application service built using the rings package must accept. In Scala, a *trait* is (in essence) a base class: it cannot be instantiated, but other classes may inherit its methods and behavior. The `AppServiceAPI` trait is empty: it is a placeholder in the type hierarchy to represent the relationship among its subclasses. This trait is *sealed*, which means that all of its subclasses are defined within this file. It is not necessary to say this, but it may assist the compiler with certain kinds of error checking. The next three lines define three subclasses: `Prime`, `Command`, and `View`. Each of these classes represents a message type.

These message types are defined as `case` classes. In Scala, marking a class as `case` is a shorthand for some useful behaviors. Briefly:

- Instances of a case class are endowed with public immutable fields whose names and values derive from the parameters in the class definition.
- Each case class implicitly defines an object (a companion object) with an `apply` method that instantiates a new instance of the class. That makes it possible to create a new instance of the case class (e.g., a new message) just by naming the class, without the `new` keyword. `TestHarness` instantiates the `Join` and `Start` messages in this way in the code snippet shown and discussed above.

- Case classes facilitate use of Scala pattern matching. We see this used later to decode incoming messages in the message receive functions of the actors.

I am not suggesting that Akka messages must be instances of case classes: any object may be sent as a message. Nor am I suggesting that case classes are only useful for messages. This is just an example of a useful Scala feature and one recommended way to define message protocols.

3.3 For loops

`KVAppService` illustrates three forms of Scala's for loop:

```
val stores = for (i <- 0 until numNodes)
  yield system.actorOf(KVStore.props(), "RingStore" + i)

val servers = for (i <- 0 to numNodes-1)
  yield system.actorOf(RingServer.props(i, numNodes, stores, ackEach), "RingServer" + i)

for (server <- servers)
  server ! View(servers)
```

The `for...until` and `for...to` variants here have the same meaning. These examples illustrate that a for loop may return a value. The `for...yield` construct builds the value to return. These `for...yield` loops return values that are instances of immutable collection types (e.g., sequences or lists). Each iteration of the loop yields a new element for the collection. The final value is a collection of all of the yielded values.

The third loop shows how easy it is to iterate through a collection in Scala. This example also illustrates how it is possible to send messages containing complex collection objects. That loop uses the tell (`!`) operator to send to each server in the application tier a sequenced list of `ActorRefs` for all of its peers. After an actor receives the `View` message, it knows these `ActorRefs` and so can send messages to its peers.

Note that the string names of actors in these service tiers are made using the `+` operator: as in Java, it means "concatenate" when applied to strings. Take care when using this operator: as in Java, string concatenation is extremely slow.

4 KVStore

Now let's look at the service tiers from the bottom up. The `KVStore` class is the first class we've looked at that defines an actor: each instance of `KVStore` is an actor. As we know, an actor is an object that can receive and react to messages. The actor handles messages in sequence, completing the processing of each message before accepting the next. In a pure actor-based program the actor code executes only in response to messages: each actor executes sequentially. An actor-based program is concurrent because different actors may run in parallel (e.g., on different threads).

An actor has a private internal state, which is/should be accessed only by sending messages to the actor. Because each actor is (in essence) single-threaded, the actor state does not require any locks or other synchronization to manage concurrency—*presuming it does not share any mutable data with other actors or futures.* Note this caution well, since it is possible to pass mutable object references in messages among actors. This pattern can be very useful, but any shared objects must be thread-safe and their interactions with the actor messaging patterns must be free of deadlock.

4.1 Defining actors

An actor class extends (is a subclass of) the Akka `Actor` trait. Actors have an event-driven structure: each actor includes a `receive` method that defines a function to apply to all incoming messages of this actor. An Akka actor receives and processes messages one at a time through this function. To illustrate, here is the class definition for `KVStore`:

```
class KVStore extends Actor {
  private val store = new scala.collection.mutable.HashMap[BigInt, Any]

  override def receive = {
    case Put(key, cell) =>
      sender ! store.put(key,cell)
    case Get(key) =>
      sender ! store.get(key)
  }
}
```

An `Actor` subclass must provide its own definition of the `receive` method. This actor `receive` method above illustrates the use of Scala pattern matching to decode the incoming messages. `KVStore` accepts only two message types: `Get` and `Put`. Upon receiving a `Put` message, the actor places the value in its store, mapped to the specified key. Upon receiving a `Get`, it attempts to retrieve a value from its store by looking up the key. The results are returned to the sender, whoever that is. The `sender` identifier actually refers to a method of `Actor`, which returns the `ActorRef` of the actor that sent the most recently received message.

4.2 The Scala Collections API

In this example, the state of each `KVStore` actor is an unsynchronized hash table. The hash table is a memory-based data structure. Of course, a real storage server would store data on disk.

It is easy to use a hash table because Scala does all of the work. To implement a data collection like this, we select a suitable collection type from the rich, powerful, and well-documented Scala collections API in package `scala.collection`, and use the `new` operator to make a collection object of the selected type:

```
  private val store = new scala.collection.mutable.HashMap[BigInt, Any]
```

The collections API makes it easy to handle mundane data structures (lists, queues, arrays, hash tables) that are so common in systems code. The `HashMap` maintains a hashed mapping from keys to values: lookup (`get`) of a key returns a value in constant time, if a mapping for that key is available. A `put` installs a mapping for a key, also in constant time. Because we will make changes to the `HashMap` as the application executes, it must be `mutable`.

Scala collection types are *parameterized types*. In the case of `HashMap`, either the keys or the values could be of any type. We can have a `HashMap` that maps `String` to `Int`, or `Int` to `String`, or...whatever else we choose. However, we must choose. All mappings in any given `HashMap` object must conform to a single specified key type and a single specified value type. The code that instantiates a parameterized type via `new` must specify the type parameters (in square brackets `[]`). Once the types are specified, they cannot change.

In this case, we use the Scala `BigInt` type for the keys: BigInts are 128-bit integers, which might be typical for keys in real key-value stores. But what about the values? We prefer to avoid constraining applications: an app could store any kind of data in a key-value store. With a real key-value store, we might encode complex data structures using some format like JSON or XML, store them as strings, and then unpack a string back into a complex data structure after reading it from the store. This concept has been called *pickling,*

marshalling, flattening, or object serialization in the past, particularly in conjunction with RPC systems. You should be sure to understand it.

In this case, the code using the `KVStore` is all native in Scala, and the Scala type system has *inclusion polymorphism*: types are arranged in class/trait hierarchies in which more specific types extend and conform to the types of their predecessors in the hierarchy, which are less specific. Every Scala object is an instance of type `Any`. Knowing that an object is of type `Any` does not tell you much about it: you cannot use an `Any`-typed reference to access the object's fields or invoke its methods. But in the case of `KVStore`, all that is needed is to store references to the values in the store, retrieve those references, and pass them on.

Because `KVStore` stores values of type `Any`, its clients must do some type casting to "unpack" their values when they get them back, converting them from the vague type `Any` back to a more specific application type. We will see that later.

4.3 Companion objects and Actor Props

KVStore also illustrates the useful Scala concept of companion objects. A *companion object* is an `object` definition that has the same name as a class definition in the same file.

```
class KVStore extends Actor {...}
object KVStore {...}
```

Like the other object definitions we have looked at, the `KVStore` object is instantiated as a singleton when the program launches. The class and object are companions. In particular, members of the class have access to the fields of the object. In Scala, the use of a companion object with an `apply` method is a common design pattern that fills the role of the "factory" objects and classes that often infest Java programs.

In this case, the `KVStore` object does nothing more than to provide a single method that returns an Akka `Props` object for use in making actors of the `KVStore` actor class.

```
  def props(): Props = {
     Props(classOf[KVStore])
  }
```

As is apparent from this example, `Props` are also constructed by applying a companion object for the `Props` class. You can read about `Props` in the Akka documentation (if you are curious). They exist primarily to enable deployment of actors on remote machines across a network. Basically, `Props` is an object that can be passed by value across the network and that encapsulates everything an Akka system needs to know to instantiate a given type of actor.

5 KVClient

`KVClient` implements a client's interface to a `KVStore`, with an optional writeback cache. The idea is that an array of `KVStore` actors/servers acts as the data storage tier for a set of actors/servers for some specific application. Each of the app actors/servers creates a local `KVClient` object that provides that actor's interface to the storage tier. Like the `KVStore` itself, the `KVClient` API stores values of type `Any` indexed by keys of type `BigInt`.

`KVClient` is the first non-case class we have seen that takes a parameter. Each `KVClient` object is passed a reference to an indexed sequence (indexable like an array) of `ActorRef` references for the `KVStore` actors. Since there are no constructor methods in Scala, we indicate this by placing the parameters directly in the class definition:

```
class KVClient (stores: Seq[ActorRef]) {
...
}
```

The `stores` sequence is then addressable as a member field within each `KVClient` instance. There is no need to define a local member field and assign the parameter value to it, as in Java.

5.1 Routing requests to the storage tier

Looking at the `route` method, we see that the `KVStore` actor for a given key is selected by a simple modulo operator (`%`) over the length of the stores sequence, i.e., the number of storage servers. The output of the mod is the server number: once we have the server's number, we can use it to index into the `stores` sequence to obtain the `ActorRef` for that server, which is returned.

```
  /**
    * @param key A key
    * @return An ActorRef for a store server that stores the key's value.
    */
  private def route(key: BigInt): ActorRef = {
      stores((key % stores.length).toInt)
  }
```

In this way the key space is partitioned evenly across the `KVStore` actors. This simple approach is good enough for our "key-value store in a bottle". But consider what happens if we add a `KVStore` actor, changing the value of `stores.length`. Every single key would route to a different actor than it did before! We have the same problem if an actor/server fails, as well as the additional problem of losing any data stored by that server! Real key-value stores must be robust to changes in the number of servers to handle elasticity and failures. To this end, they incorporate replication and use more sophisticated routing approaches such as *consistent hashing* to route keys among the servers. Typically the routing function is implemented within the storage service itself, rather than on the clients, as it is in this example.

5.2 Managing the key space

Another interesting method of `KVClient` is `hashForKey`. It is intended as an illustration of how to index application data evenly in a uniform key/value key space.

```
  import java.security.MessageDigest

  /** Generates a hash key for an object to be written to the store.  Each object is created
    * by a given client, which gives it a sequence number distinct from other objects created
    * by that client.
    * @param nodeID Which client
    * @param seqID Unique sequence number within client
    */
  def hashForKey(nodeID: Int, seqID: Int): BigInt = {
    val label = "Node" ++ nodeID.toString ++ "+Cell" ++ seqID.toString
    val md: MessageDigest = MessageDigest.getInstance("MD5")
    val digest: Array[Byte] = md.digest(label.getBytes)
    BigInt(1, digest)
  }
```

The first thing to notice is how `hashForKey` maps strings into the key space. We use the MD5 hashing/digest function to take a hash of the string, which we then whittle down to a `BigInt` that may be used directly as a key. This shows how to map arbitrary strings into the key space. Question to ponder: what is the probability that two different strings might hash down to the same key, generating a collision in the store? Is it a risk that the application must consider, detect, and handle?

Note that the MD5 digest function is obtained from the `MessageDigest` class in the Java standard library. Scala programs run on a Java JVM: this enables Scala to be interoperable with Java. A Scala program may instantiate and operate on Java classes and objects in this way.

In real applications of key-value stores it is common to obtain keys by hashing strings or other values using MD5 or other hash functions. For example, *deduplication* systems may store e-mail messages or documents by the hash of their contents, so that the store retains only one copy of each message or document: all copies of a given document hash to the same key, naturally coalescing them into a single copy.

The second thing to notice about `hashForKey` is how it builds the string to hash. Again, this is an illustration: this approach is general, but it is not necessarily the best or fastest way to get the job done. The premise here is that each value in the `KVStore` represents some object created by a specific app server, and that each app server assigns a unique number to every object it creates. Given both numbers, we can obtain a key for the object. In essence, this approach partitions the key space among the app servers, so that two app servers do not accidentally choose the same key. It makes it easy for app servers to guess which keys are used by their peers, and to index one another's data. This approach is useful in synthetic benchmarking scenarios, as in this example. It has the nice property that if app servers create objects at different rates, the load is still spread evenly across the storage tier.

5.3 The KVClient cache

The `read, write, push`, and `purge` primitives of `KVClient` show how to use the optional writeback cache. The client maintains a local `HashMap` of key-value mappings as a cache. A read is satisfied from the local cache if the requested key and value are present (hit). If the cache access is a miss, then the read request is routed as a `Get` message to the `KVStore` server that is responsible for the requested key. The caller may use `purge` to invalidate the entire cache.

The cache is writeback: a `write` is placed in the local cache and not written to the `KVStore` immediately. To keep track of dirty values, each write is recorded in a `dirtyset` passed by the caller. The caller can `push` the dirtyset at a later time to force the writes out to the `KVStore`.

`KVClient` provides `directRead` and `directWrite` primitives that bypass the cache and route directly to a `KVStore` server. For the GroupServer assignment, I recommend using these direct primitives. We expect to use the writeback cache in a later assignment.

6 RingService

`RingService` is an example KV application service. It does not do anything useful: it just reads and writes at various keys and makes sure that things are working as expected. The comment says it all:

```
/**
 * RingService is an example app service for the actor-based KVStore/KVClient.
 * This one stores RingCell objects in the KVStore.  Each app server allocates new
 * RingCells (allocCell), writes them, and reads them randomly with consistency
 * checking (touchCell).  The allocCell and touchCell commands use direct reads
 * and writes to bypass the client cache.  Keeps a running set of Stats for each burst.
 *
```

```
 * @param myNodeID sequence number of this actor/server in the app tier
 * @param numNodes total number of servers in the app tier
 * @param storeServers the ActorRefs of the KVStore servers
 * @param burstSize number of commands per burst
 */
}
```

`RingService` includes some methods that give an example of how to use the `KVClient` writeback cache. These methods are not currently used: all reads and writes are direct.

At a later time we expect to see a more complete version of `RingService` that makes it clear why it has the name. Briefly, the idea is to store linked data structures within the `KVStore`; the `RingCell.prev` and `RingCell.next` pointers store keys of other values in the store, acting (in essence) as pointers. However, managing linked data structures in a key-value store will require strong consistency or even transactional consistency.

Here are some new Akka/Scala points to notice in `RingService`:

- **Random numbers**. RingService uses a `scala.util.Random` to generate pseudo-random integers to decide what to do next. It is a useful technique for synthetic benchmarks and testing code like this. The integer generated by `generator.nextInt` is between zero and the value passed as an argument.

```
val generator = new scala.util.Random
...
private def chooseActiveCell(): BigInt = {
   val chosenNodeID =
     if (generator.nextInt(100) <= localWeight)
       myNodeID
     else
       generator.nextInt(numNodes - 1)
   ...
}
```

- **If-else**. The method `chooseActiveCell` (above) illustrates that a Scala `if` statement can form an expression that returns a value. The method `rwcheck` shows that an `if` statement may have any number of `else` branches.

- **Type casts**. The `read` primitives cast values returned by the `KVClient/KVStore` into instances of the desired/expected type. The sections for `KVClient` and `KVStore` explain why this casting is necessary. Actually, the `write` methods use casting too, because a write into the store returns a previous value associated with the specified key, if any. Type casting is performed by `asInstanceOf`, which is a method of any Scala type. It takes a type argument (square brackets), and attempts to cast the invoked object reference into a reference of the specified type. It returns the retyped reference. Obviously, you should cast sparingly. Note that casting does not circumvent the type system as it does in C: if the invoked type is incompatible with the requested type, an exception is thrown at runtime. Scala is strongly typed, and there is no escaping the type system.

- **Props with arguments**. The companion `RingServer` object shows how to create `Props` for an Akka actor that is constructed with arguments. It is simple: the `Props` apply method lets us pass a variable number of arguments of any types, which are passed through to the actor when it is instantiated.

```
object RingServer {
  def props(myNodeID: Int, numNodes: Int,
            storeServers: Seq[ActorRef], burstSize: Int): Props = {
    Props(classOf[RingServer], myNodeID, numNodes, storeServers, burstSize)
  }
}
```

6.1 Options, Some, and None

`RingService` illustrates a concept of Scala of *optional values*. In Java or C it is common to use "null" to indicate that an object reference value is empty, and does not refer to any object. Scala does not have null pointers/references. It does not allow them. There is no concept of an uninitialized or null value. So Scala needs another construct for that purpose.

Instead, Scala has a parameterized type called `Option`. A value of an `Option` type either contains a value of its base type, or it has the value `None`. Reads and writes on the key-value store return `Option` types. In particular, if the store has no value associated with a specified key, then a read or a write on that key returns `None`. Indeed, the underlying collections (e.g., `HashMap`) also have this behavior. It is common for collections to use `Option` values.

Let's consider the `RingService.read` method as an example:

```
private def read(key: BigInt): Option[RingCell] = {
  val result = cellstore.read(key)
  if (result.isEmpty) None else
    Some(result.get.asInstanceOf[RingCell])
}
```

This method returns an `Option` type whose base type is `RingCell`. The returned value is the result of the `if...else` expression.

The `if...else` expression also illustrates how to handle instances of `Option` types, since `KVClient` also returns them. We can ask if an `Option` type `isEmpty` (value is `None`) or if the option value `isDefined` (it has some value). If an option type has a defined value, we use the `Option.get` method to obtain it. We can return a base value as an `Option` type by wrapping it up as a `Some`. Or we can assign `None` to any `Option` type to indicate that there is no value.

As you become an expert in writing concurrent code in Scala, you will often find yourself working with futures and options. But there is no money involved! (That's a joke. You should laugh when I tell a joke. And of course being an expert in building networked systems with Akka/Scala is a marketable skill.)

6.2 Garbage and shared references

A final point to note about `RingService` is that it periodically sends its `Stats` object to the master, and replaces it with a new `Stats` object.

```
var stats = new Stats
...
private def incoming(master: ActorRef) = {
  stats.messages += 1
  if (stats.messages >= burstSize) {
    master ! BurstAck(myNodeID, stats)
    stats = new Stats
  }
}
```

You should be sure to understand the subtleties of how code like this behaves.

First, the old `Stats` object is sent by reference to the master actor. If the master resides on a different machine (JVM), then the `Stats` object is copied, i.e., it is passed by value. But if the receiving actor resides in the same JVM, then the object is passed by reference. If the object is passed by reference, and the sender

retains a reference after sending it, then the sending and receiving actor share the same copy of the object in memory. If either actor modifies the object, then that change may (or may not) be visible to the other actor. A shared mutable object must be thread-safe!

Akka dogma considers such object sharing to be taboo. You are strongly encouraged to ensure that objects passed by reference from one actor another are either *immutable* (such as a `case` object or an immutable collection), or are not shared. In this case, the object is mutable (`Stats` is full of `var` fields, as noted above), but the sender overwrites its only reference to the object after sending it, so the object is not shared.

Second, by overwriting a reference in this way, the `incoming` method is creating garbage. Scala objects are JVM objects: they are allocated dynamically on a heap, and an object is reclaimed by the JVM garbage collector at some time after all references to it have been overwritten or otherwise destroyed. In this case, the receiver obtains a reference, so the object is not garbage—yet. But if you look at the receiver code in `LoadMaster` you will see that the receiver quickly discards its reference too.

In general, you should be aware when your programs are creating garbage. Too much garbage is a major cause of performance problems for Java and Scala programs. Scala encourages programming in a functional style with immutable values, which creates a risk of excessive overhead for cloning data structures and then abandoning them as garbage. However, Scala immutable collection classes are carefully optimized to minimize this overhead. It is important to be mindful of this issue.

7 LoadMaster

Finally we arrive at the top of our actor tree. `LoadMaster` is an actor that represents a synthetic request load generator for our application service. It sprays requests at the app service tier and gathers measurements and statistics. After a configured number of requests have completed, it sends the final statistics to another actor that has registered as a listener with a `Join` message. Actually, the listener is the main `TestHarness`, which is not created as an actor, but can send and receive messages just like a real actor.

Such load generators are common for testing and benchmarking services. `LoadMaster` has a few interesting properties.

Most importantly, `LoadMaster` is self-clocked. It sends commands to the app tier in bursts, and sends the next burst only after a server acknowledges the previous burst. Why? Message-passing in Akka is not flow-controlled: if one actor sends messages to another without waiting for the receiver to catch up, it may fill up memory with buffered messages in the receiving actor's mailbox, and die horribly with an out-of-memory exception. If your Akka program runs out of memory, the second step back to health and sanity is to review your flow control scheme. The first step is to be sure that you are giving it enough memory to begin with, e.g., by putting the line `export SBT_OPTS=-XX:MaxPermSize=512M` in a plain text file called `.sbtconfig` in your home directory.

In contrast, Unix pipes and TCP sockets are flow-controlled: they use a bounded, fixed amount of buffer space for data that has been sent but not yet accepted at the receiver. The sender automatically slows down to stay within that buffer space if the receiver cannot keep up. Akka does not behave that way by default, so you have to build it into your application somehow.

Second, `LoadMaster` is generic. It does not care what the application service tier is actually doing. What it does is send bare `Command` messages, which every application service must accept. The application server decides for itself what to do on each `Command` to emulate a suitable application load. For example, it might make some randomized choices, as in `RingServer`. Leaving this part of load generation to the app itself is an unconventional choice, but it has the advantages that it is generic and it distributes the work of generating the load across the app tier. (Note that this is a disadvantage if our purpose is to benchmark the app service.)

`LoadMaster` also uses an `akka.event.Logging` object to log an info message. If an actor logs a message, the message appears on the output with a timestamp and a string identifying the actor that generated the

message. This logging facility is used to convey info and debug information to a developer or administrator: it should not be confused with data logging for recovery (e.g., as we discussed in class with transactions).