PoDS 2021, 2nd Course Project: Cab Hailing Application using Akka

The objective and overall requirements of this project are essentially the same as with Phase 1. The differences will be primarily in the implementation details, as Phase 2 is in Akka.

Phase 1 of the project

In Phase 1, the objective is to create a single node (or single Actor System) Akka application. We list below the requirements from this application.

Naming convention: in this document, whenever we mention the name of an actor type beginning with a capital letter (e.g., Main, Cab, Wallet, RideService), you should use the exact same name for the corresponding actor class. Also, for simplicity, you should place all your classes (except the test script, mentioned below) in a single package, with the name `pods.cabs`.

The actor system

In this project, instead of using a normal ActorSystem, we will use an ActorTestKit, which makes it easier to write test scripts. Actor test kits are alternatives to actor systems, and are covered in the Akka Quickstart Example. Unzip the example file, and run "mvn test" to run the test program in the directory src/test/java/com/example/AkkaQuickstartTest.java. In this test program you will note that a variable called testKit is first initialized. After initialization, this variable refers to a TestKit, which is similar to an actor system (or user-guardian actor) in a normal program. Then, in the test program, one or more test methods are to be provided; in the example, there is a single test method, named testGreeterActorSendingOfGreeting. A test method is supposed to spawn one or more actors (as children of the TestKit actor), send them messages, and then check if the response messages are satisfactory. This is the structure of any test program in Akka.

In your project, your test method should:

1. spawn a "Main" actor, and pass an ActorRef<Main.Started> that refers to a test probe created within the test method to Main.create()

2. using the probe, wait for a Main.Started message from Main.create() (this message would signify that Main.create has spawned all the actors that it needs to)
3. proceed with the *testing steps* (discussed towards the end of this document)

The Main actor

In its static .create() method this actor should carry out the following sequence of actions:

1. Read the input file (which will be in the same format as in Project 1), and spawn all the Cab and Wallet actors. There will be one Cab actor per cab and one Wallet actor per customer, and these actors will internally store the respective state information. Store the cab actors in a Map of type <String -> actor refs to Cab actors>, where the keys are the cabIds. Similarly, store the Wallet actors in a Map whose keys are customer IDs (Strings). These two maps should be called Globals.cabs and Globals.wallets, respectively; i.e., Globals is a class, and “cabs” and “wallets” are “public static final” fields of this class. (Normally, shared state in the form of global variables is not encouraged in Akka programs. However, since these maps will be populated before the program does any actual work and will remain read-only after they are populated, it would be ok.)
2. Spawn 10 different RideService actors. We want more than one of these actors in order to be able to receive requests concurrently. The idea is that the RideService actors should effectively provide a load-balanced stateless service. Keep these actors in an array called Globals.rideService.
3. Send a Main.Started message to the test probe in the test method via the ActorRef<Main.Started> actor ref sent by the test method
4. Return Behaviors.empty() [meaning, it does want to receive any messages]

The RideService actors

These actors should be able to receive the following commands:

- RideService.CabSignsIn(String cabId, int initialPos)
- RideService.CabSignsOut(String cabId)
- RideService.RequestRide(String custId, int sourceLoc, int destinationLoc, ActorRef<RideService.RideResponse> replyTo)

(RideService would need to receive additional commands on top of the ones listed above. This will become clear later on in this document. We do not concretely specify how you should design those additional commands. The three commands above need to be supported exactly as specified.)

Each RideService actor maintains an internal cache-table of the last known locations and statuses of all the cabs. Since any command from the test program or from Cab can come to any of these actors, and since these actors need to appear stateless to the outside world, it follows that these cache tables need to be kept (eventually) consistent. You can do this by sending internal “update” messages from the original RequestRide actor that received any of the three commands mentioned in the bullet list above to all the other RequestRide actors. These internal message types are not listed in the bullet list above, but you will need to have them.

The first two commands in the bullet list above do not need responses; they should just be used to update the cache table(s). The last command above should be handled as follows. Since finding a cab and deducting from the wallet can be a time consuming activity, the RideService actor should simply spawn a FulfillRide actor, and pass-off the RequestRide arguments to this actor. A copy of the cache-table should also be given to this actor. The FulfillRide actor will then find a ride and respond to the “replyTo” actor (i.e, the test program in our case).

The FulfillRide actor

The FulfillRide actor is a bit like the DeviceGroupQuery actor in the IoT example. It should work as per the specifications given for the end-point RideService.requestRide in Project 1. It should communicate with the cabs (in decreasing order of their distance from the starting point of the ride) using commands Cab.RequestRide, Cab.RideStarted, and Cab.RideCanceled (these commands should have the appropriate parameters). It should also communicate with the Wallet actor of the customer. Finally, it should respond to the “replyTo” actor with the message RideService.RideResponse(int rideId, String cabId, int fare, ActorRef<FulfillRide.Command> fRide), where the last parameter in the message is a “self” actor ref of the current FulfillRide actor.

If the rideId returned is -1, then the FulfillRide actor can stop itself by returning Behaviors.stopped(), else it should stay alive to later receive a FullfillRide.RideEnded command from the Cab actor. The FulfillRide.RideEnded command, when it is received, does not need a response. However, the FulfillRide actor should send an appropriate command to the RideService actor that spawned it to notify it of the current availability of the corresponding cab, and that actor should in turn forward this information to the other RideService actors via appropriate internal messages.

The Wallet actor

This actor should accept the following commands:

- Wallet.GetBalance(ActorRef<Wallet.ResponseBalance> replyTo)
- Wallet.DeductBalance(int toDeduct, ActorRef<Wallet.ResponseBalance> replyTo)
- Wallet.AddBalance(int toAdd)
- Wallet.Reset(ActorRef<Wallet.ResponseBalance> replyTo)

Note, both the FulfillRide actor and the test program need to send the messages above to Wallet actors. Hence, we use a common response message type Wallet.ResponseBalance, which has one integer field denoting the balance. FulfillRide should plug in an adapter in the place of the replyTo fields above (see a sample usage of adapters in the IoT example), while the test script will plug in an ActorRef obtained from a TestProbe.

AddBalance does not need a response. For the other three commands, the response should contain the balance after the command is handled. For DeductBalance, if balance deduction could not be performed due to insignificant balance, the balance should not be disturbed and -1 should be sent back in the response.

The Cab actor

The cab actor needs to support the following messages in addition to the ones mentioned earlier:

- Cab.RideEnded(int rideId)
- Cab.SignIn(int initialPos)
- Cab.SignOut
- Cab.NumRides(ActorRef<Cab.NumRidesReponse> replyTo)
- Cab.Reset(ActorRef<Cab.NumRidesResponse> replyTo)

All the commands named above would be sent from the test program. Only the last two commands above need responses.

The Cab actor needs to keep an ActorRef to the FulfillRide actor in case the cab is currently giving a ride. When a Cab actor receives a Cab.RideEnded command, it should react by sending a FulfillRide.RideEnded command to the FulfillRide actor who gave it its current ride (if the cab is currently giving a ride).

When a Cab actor receives a Cab.SignIn or Cab.SignOut request, it should forward this information to a randomly selected actor from the array Globals.rideService (random selection achieves load balancing). The selected RideService actor should in turn notify the other RideService actors via internal messages.

Any command received by a Cab actor should be ignored if the command is not applicable.

NumRides needs a response. The NumRidesResponse has a single parameter, which is the number of rides given (as in Project 1).

Upon receipt of the Reset command, the Cab should behave as if the test program sent it a RideEnded command for the ongoing ride (if there is an ongoing ride), and should then behave as if the test script sent it a SignOut command. The response should contain the number of rides given (before signing out).

The testing steps

Before any series of testing steps, the test method should first:

1. Send Wallet.Reset command to every Wallet actor in Globals.wallets
2. Send Cab.Reset command to every Cab actor in Globals.cabs
3. Wait for responses (using probes) for all commands sent in Steps 1 and 2 above.

Here is a sample series of testing steps (after the responses for the Reset commands have been received):

```
ActorRef<Cab.Command> cab101 = Globals.cabs.get(“101”);
cab101.tell(new Cab.SignIn(10));
ActorRef<RideService.Command> rideService = Globals.rideService[0];
        // If we are going to raise multiple requests in this script,
        // better to send them to different RideService actors to achieve
        // load balancing.
TestProbe<RideService.RideResponse> probe = testKit.createTestProbe();
rideService.tell(new RideService.RequestRide(“201”, 10, 100, probe.ref());
RideService.RideResponse resp = probe.receiveMessage();
        // Blocks and waits for a response message.
        // There is also an option to block for a bounded period of time
        // and give up after timeout.
assertEquals(resp.rideId != -1);
cab101.tell(new Cab.RideEnded(resp.rideId));
```

We may add some (minor) additional requirements later on, primarily to enable in-depth testing.

An important conceptual note about the happens-before ordering in Akka

It may not be obvious why we need the message Main.Started from the Main actor back to the test method. The reason is, if we did not have this message, the test method could end up continuing with the testing steps even before all the Cab, RideService, and Wallet actors get spawned. This could happen because after the Main actor is spawned and before the Main actor can spawn the other actors, the test method could continue to the testing steps.

It is for a similar reason that after sending all Reset commands from the test method, one has to block and wait for the responses to these commands. If one does not wait, the testing steps could get started and DeductBalance commands could start reaching Wallet actors even before the Reset commands reach the Wallet actors.

You need to think carefully about issues such as this when you write Akka programs.

To learn more about these topics, you could read about Message Delivery Reliability and about Akka and Java Memory Model. Or, if you want to keep it simple, always remember that there is no definite ordering between actions that occur in different actors, and there is no guarantee that if a message m1 is sent before a message m2 then m1 will reach its target before m2 reaches its target (unless m1 and m2 share the same source and target actors).

Phase 2 of the project

In this phase of the project you will use Akka cluster, cluster sharding,and persistence.

Running the system

You will run a cluster of four nodes. Each node will have an identity, which is indicated by the port number on which it is listening. Let us call the nodes A, B, C, and D for convenience. Node A will host the shared journal. Therefore, if node A goes down the system cannot function, and all nodes will have to be restarted (Node A first). The other nodes are not single points of failure.

For the sake of simplicity, you will omit the Wallet entirely from this phase of the project. None of the Wallet related functionality is required.

Each cab will be a cluster-sharded entity with persistence. Therefore, if a Cab actor resides on a node and if that node goes, the cab will be migrated automatically to some other node while preserving its state. As you know, entities need names (so that they can be referred by name within calls to sharding.entityRefFor). The cab entities will have names as given in the input file. Any Cab actor's initial state should be "signed out", and it should set this in its constructor.

The RideService actors will also be cluster-sharded entities, but without persistence. You will use a total of 12 RideService actors across the entire cluster. You can let the names of these entities be "rideService1" to "rideService12".

You will have two separate programs, a "main" program and a "test" program. The main program will have a main() function, which will create an ActorSystem as usual (recall that in Phase 1 you never created an ActorSystem). The main program takes a port number on which it has to listen to as argument, and each of the four nodes should have a unique port number on which it listens to. The main program should also take an optional flag "-firstTime".

The execution procedure to run the system is as follows. For the first time you will start the "main" program four times, each time with the corresponding port number. To each of these four invocations you will pass the "-firstTime" flag. These four invocations represent the four nodes. Subsequently, whenever a node goes down, you will replace the node by manually restarting the main program with the port number of the downed node, but without the "-firstTime" flag. [In practice one would run different nodes on different machines in a network, but you may run them on the same machine.]

Functionality required in the “main” program

When any node starts up, the ActorSystem should be created and it should join the cluster. However, it need not do any further startup activities unless it was started with the "-firstTime" flag. If it was started with the "-firstTime" flag, the ActorSystem should spawn 3 RideService actors using sharding.entityRefFor (so that across the four nodes there will be 12 RideService actors). If Node A is starting up, it should spawn RideSystem actors with names "rideService1", "rideService2", and "rideService3", if Node B is starting up, it should create RideSystem actors with names "rideService4", "rideService5", and "rideService6", and so on.

As a drastic simplification, let each RideService actor not maintain any internal state. Let it spawn a FulfillRide actor without passing it any information on the cabs, and let the FulfillRide actor contact all cabs mentioned in the input file one by one (in increasing sorted order of cabId) until it finds one that is available and filling to give a ride. There is no limit that it should contact only 3 cabs. However, cabs will still reject every alternate request that they receive when they are in “available” state. If after asking all the cabs a FulfillRide actor could not find any that is willing to give a ride, it should return -1 to the client as in Phase 1.

Note, there is no need to spawn the Cab actors explicitly, as when the FulfillRide actors contact the cabs as mentioned in the list above, the cab actors will get spawned automatically if they are not already spawned.

Note that there is no longer a need to store the actor ref to the FulfillRide actors inside the Cab actors (we are not sure one can store an actor ref inside the state of a persistent actor). Also, each FulfillRide actor can stop itself right after sending the RideResponse to the client.

Note that throughout the application, due to the use of cluster sharding, contacting any Cab or RideService is a two-step process. First, an EntityRef to the cab is obtained using sharding.entityRefFor, and then a message can be sent to the cab.

The test program

The test program will consist of test methods. A test program can be run only when the main system is running, and should join the same cluster as the main system. Unlike in Phase 1, each test method will not spawn any Main actor. Instead, it will first send Reset commands to all cabs, wait for responses to these Reset commands, and then proceed with the testing steps. When a test method needs to contact a RideService, it can contact any one of them.

As in Phase 1, the test script will be sending   commands and expecting the corresponding responses as mentioned below:

- It can send the command RideService.RequestRide to any RideService actor. It will receive RideService.RideResponse from the FulfillRide actor that was spawned by the RideService Actor that it contacted.

- It can send any of the five commands mentioned in the section "The Cab actor" earlier in this document to any cab.

All other requirements and specifications that were mentioned in Phase 1 continue to hold unless they were modified above.