COMP30220 Distributed Systems Practical

Lab 6: AKKA

Work individually. Submit your code on csmoodle.ucd.ie by the deadline given on moodle. Please submit a single ZIP file (name is <student-num>.zip) containing your source code and any text files required for the solution.

Download the original version of Quoco again (do not attempt to adapt your answer to the previous lab).

The broad objective of this practical is to adapt the code provided to use Actors for interaction between each of the 3 quotation services and the broker and between the broker and the client. **In the final version, each of these components should be deployable as a separate docker image and you should provider a docker-compose file that can be used to deploy the images.**

To help you complete this challenge, I have broken the problem up into a set of tasks. My advice is to create a separate set of projects for your solution, and for you to copy code from the original project as needed. As an aside, this lab will have some similarities with the Message-Oriented Middleware lab. Communication between actors is based on asynchronous message passing.

Task 1: Setting up the Project Structure **Grade: E**

As with all other laboratories, we will create a multi-module maven project with the following subprojects:

- **core**: contains the common code (abstract base classes & any data classes)
- **auldfellas**: The Auldfella's Quotation Service
- **dodgydrivers**: The Dodgy Drivers Quotation Service
- **girlpower**: The Girl Power Quotation Service
- **broker**: The broker service
- **client**: The client service

For all projects, the **groupId** should be "lab6" and for the main project the artifactId should be "quoco-akka". For the main project, copy a **pom.xml** from one of the previous labs. Do the same for the core module. For the remainder of the modules, **copy the dependencies** from one of the AKKA projects I have provided on Brightspace. You will need to include the exec-maven-plugin for all the projects (apart from core). Also, remember to add the lab6:core:0.0.1 dependency to every module but the core module.

The following steps will help you set up the **core** project:

a) Create a "src/main/java" folder and copy the "service.core" package into it (remember that you have to create a "service/core" subfolder within the "src/main/java" folder and copy the Java source files into the "service/core" folder.

b) Delete the `BrokerService` interface.

c) Modify the `Quotation` and `ClientInfo` classes to be Java Beans.

Remember, a class is a Java Bean if it is a data class, if it has a default constructor (a constructor with no parameters), if its fields are private, and it has set/get methods for each field.

d) Compile & Install the "core" project

Note: We are going to add the messages to this package, so be prepared to re-install the "core" module multiple times throughout the lab.

Task 2: Creating and Testing the Auldfellas Quotation Services **Grade: D**

The second task involves creating Quotation Services Actors. I will start by explaining how to do it for one of the services – auldfellas – doing it for the other services should be trivial. Basically, the expected behaviour here is to have an actor that generates a quotation every time it receives a quotation request. The quotation request should contain the client info. It should send a quotation response message to the actor that sent the quotation request. The response should contain the quotation. Because we are doing things asynchronously, we will need to map responses to requests, so both the request and the response should include a unique identifier. The quotation actor should copy this identifier from the request to the response.

a) Create the "src/main/java" folder structure and copy the "service.auldfellas" package into it.

b) Copy the code below into the core project under the "service.messages" package.

```
public class QuotationRequest {
  private int id;
  private ClientInfo clientInfo;

  public QuotationRequest(int id, ClientInfo clientInfo) {
    this.id = id;
    this.clientInfo = clientInfo;
  }
  // add get and set methods for each field
}

public class QuotationResponse {
  private int id;
  private Quotation quotation;

  public QuotationResponse(int id, Quotation quotation) {
    this.id = id;
    this.quotation = quotation;
  }
  // add get and set methods for each field
}
```

c) Re-install the core project

d) Now, we create the `service.actor.Quoter` class which we will use to will implement the Actor which will receive incoming requests for quotations by generating and returning new quotations to the sender.

```
public class Quoter extends AbstractActor {
  private QuotationService service;

  @Override
  public Receive createReceive() {
    return receiveBuilder()
      .match(QuotationRequest.class,
        msg -> {
          Quotation quotation =
            service.generateQuotation(msg.getClientInfo());
          getSender().tell(
            new QuotationResponse(msg.getId(), quotation), getSelf());
        }).build();
    }
}
```

This is quite a nice solution – the code will work for ALL the services. The only problem is how to assign an initial value to the service field?

e) To implement support for initializing the actor, we need to create another message called `service.message.Init`. This message can be created in the auldfellas project because it will only be used to initialize the Quoter actor, which will be done within the project. This message should have one field – an instance of the `QuotationService` called `service`.

Once you have created the message, you should implement a second handler in the `Quoter` actor that, upon receipt of an `Init` message, sets the `service` field to be equal to the value passed through the message:

```
service = msg.getQuotationService();
```

f) Now that we have our actor implementation, we should test that it works as expected. To do this, you should follow the guidance of the Unit Testing with Akka lecture. Once set up, you should implement a unit test that creates a `Quoter` actor and sends an `Init` message to the actor followed by a `QuotationRequest` message. A sketch of the unit test can be seen below:

```
@Test
public void testQuoter() {
  ActorRef quoterRef = system.actorOf(Props.create(Quoter.class), "test");
  TestKit probe = new TestKit(system);

  quoterRef.tell(new Init(new AFQService()), null);
  quoterRef.tell(new QuotationRequest(1,
      new ClientInfo("Niki Collier", ClientInfo.FEMALE, 43, 0, 5, "PQR254/1")),
    probe.getRef());
  probe.awaitCond(probe::msgAvailable);
  probe.expectMsgClass(QuotationResponse.class);
}
```

Create and run the unit test. The output should look something like this:

```
rem@Krytan:/mnt/c/Development/comp30220/quoco-akka$ mvn test -pl auldfellas
[INFO] Scanning for projects...
[INFO]
[INFO] ---------------------------< lab6:auldfellas >---------------------------
[INFO] Building auldfellas 0.0.1
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ auldfellas ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /mnt/c/Development/comp30220/quoco-akka/auldfellas/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ auldfellas ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ auldfellas ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /mnt/c/Development/comp30220/quoco-akka/auldfellas/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ auldfellas ---
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent!
[INFO] Compiling 1 source file to /mnt/c/Development/comp30220/quoco-akka/auldfellas/target/test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ auldfellas ---
[INFO] Surefire report directory: /mnt/c/Development/comp30220/quoco-akka/auldfellas/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running QuoterUnitTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.018 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.559 s
[INFO] Finished at: 2020-11-05T12:32:27Z
[INFO] ------------------------------------------------------------------------
rem@Krytan:/mnt/c/Development/comp30220/quoco-akka$
```

g) Repeat for the other two quotation services…

Task 3: Implementing the Broker I **Grade: C**

In break with tradition, the next task will focus on implementing part of the broker. The focus of this task will be on understanding how to use Akka Remoting (the distribution technique we will use in this project). To do this, we will explore how to get the quotation services to register with the broker. **This will require a number of changes to the code AND to the pom files.**

a) First, lets update the pom.xml file for each quotation service. You need to add the following dependencies to enable Akka Remoting:

```
<dependency>
      <groupId>com.typesafe.akka</groupId>
      <artifactId>akka-remote_2.12</artifactId>
      <version>2.6.0-M8</version>
</dependency>
<dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty</artifactId>
      <version>3.10.6.Final</version>
</dependency>
<dependency>
      <groupId>com.typesafe.akka</groupId>
      <artifactId>akka-cluster-typed_2.12</artifactId>
      <version>2.6.0-M8</version>
</dependency>
<dependency>
      <groupId>com.typesafe.akka</groupId>
      <artifactId>akka-serialization-jackson_2.12</artifactId>
      <version>2.6.0-M8</version>
</dependency>
```

b) The next step is to implement message serialisation. This is done through a combination of adding a Java Interface and creating an Akka configuration file. **The interface will be created in the core project (which will have to be reinstalled).** The name of the interface should be `service.messages.MyInterface`. It should be empty:

```
package service.messages;

public interface MySerializable { }
```

The `QuotationRequest` and `QuotationResponse` classes should be modified to implement this interface.

Next, we need to create an Akka configuration file. This should be located in the "src/main/resources" folder of each quotation service project and should look like this:

```
akka {
  actor {
    provider = cluster
    serialization-bindings {
      "service.messages.MySerializable" = jackson-json
    }
    serializers {
      jackson-json = "akka.serialization.jackson.JacksonJsonSerializer"
    }
  }
  remote.artery.enabled = false
  remote.classic {
    enabled-transports = ["akka.remote.classic.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
 }
}
```

The port number should be unique for each service (e.g. 2551 for the broker, 2552 for auldfellas, 2553 for dodgydrivers and 2554 for girlpower). Akka automatically searches for and loads this file when it starts up.

Notice the reference to MySerializable in the serialization-bindings section. This is the link that enables serialization for the messages. We could have done it without the interface, but then we would need a line for each message class.

c) Now, lets' move on to the broker. Lets create a new class `service.actor.Broker` in the broker project. You should make sure that the dependencies for this project are the same as those used in the quotation service projects. As a first step, lets think about how we can set the service up so that we don't have to statically configure the link between the quotation services and the brokers. The easiest way to solve this is to get the quotation services to register with the broker. That way, the broker can use the list of registered services when requesting quotations.

We can implement this as a simple String message that is sent from each quotation service to the broker actor. The broker actor can then maintain a list of the `ActorRef` for each quotation service that contact it. From the brokers perspective, all that is needed is for it to be able to handle an incoming registration request message:

```
return receiveBuilder()
  .match(String.class,
    msg -> {
      if (!msg.equals("register")) return;
        actorRefs.add(getSender());
  }).build();
```

In the quotation service, we can then add some code to send the registration message to the broker. We do with by creating a Main class with a main() method:

```
public static void main(String[] args) {
  ActorSystem system = ActorSystem.create();
  ActorRef ref = system.actorOf(Props.create(Quoter.class), "auldfellas");
  ref.tell(new Init(new AFQService()), null);

  ActorSelection selection =
    system.actorSelection("akka.tcp://default@127.0.0.1:2551/user/broker");
  selection.tell("register", ref);
}
```

The first three lines of code start the local actor system and create and initialize the Quoter actor (this one is the auldfellas actor).

The last two lines send the message to the broker. I could have done this through the Quoter actor (it would be purer because the Quoter would include the code to register with the broker), but this just adds pointless complexity…

The ActorSelection object is used to define an actor that you want to interact with when you don't have its ActorRef object. The string is a URI that uniquely defines the actor you are interested in. The 127.0.0.1 is the localhost IP address (Akka is picky and will not accept localhost instead of the IP address); 2551 is the port of the ActorSystem that the broker is running on; and /user/broker is the fully qualified identifier of the broker actor on that system (all actor names are prefixed by /user/). The identifier is basically a path. For example, if actor a1 created actor a2, then the identifier of a2 would be: /user/a1/a2. You are going to have to "mess" with these selector strings to make them work correctly for your own machine.

Create the Main class for each quotation service and use the exec maven plugin to execute this class. Do the same for the broker project (you will only need to use an adapted version of the first 3 lines of the main method above).

d) To test that this all works, modify the "registration" behaviour of the Broker to: (1) print something out, and (2) to send a Quotation Request to each registering quotation service (use one of the example ClientInfo objects). Add a message rule to handle the response (it is enough to just print out the reference number). Check that everything works…

Task 4: Implementing the Broker II **Grade: B**

Now, lets complete the broker implementation. Specifically, lets look at the part related to the incoming client request and the resultant response. To handle this, we need to create two new messages (that will be part of the core project), named: `service.messsages.ApplicationRequest` and `service.messsages.ApplicationResponse`. The former message should have 1 field: clientInfo, and the latter should have 2 fields: clientInfo and a list of quotations. Both message classes should be Java Beans and should implement `MySerializable`. Don't forget to reinstall the core. Lets now focus on the behaviour of the broker:

a) The first step is to handle the incoming request from the client. You need to write a message handler for this, with the message type being the `ApplicationRequest` message. The expected behaviour for this message is to associate the (clientinfo part of the) request with a unique identifier and to send the request onto all the known quotation services:

```
for (ActorRef ref : actorRefs) {
  ref.tell(
    new QuotationRequest(SEED_ID, msg.getClientInfo()), getSelf()
  );
}
```

NOTE: You will need to do something similar to what you did in the JMS lab (in terms of using a Map to keeping a record of any quotations received for the given application).

As with the JMS lab, you need to get the actor to wait until a given deadline, after which it returns any messages that it has received. In Akka, you can do this through a built in scheduler:

```
getContext().system().scheduler().scheduleOnce(
  Duration.create(2, TimeUnit.SECONDS),
  getSelf(),
  new RequestDeadline(SEED_ID++),
  getContext().dispatcher(), null);
```

Here, the scheduleOnce(…) method is used to schedule the sending of a message to "getSelf()" after 2 seconds. The message to be send is an instance of the `service.messages.RequestDeadline` class which should be created locally in the broker project. This class should include a field that can be used to refer to the unique identifier of the associated application.

b) Implement a message handler for the `service.messages.QuotationResponse` message. The unique identifier associated with this message should be used to match the quotation to the request.

c) Finally, you need a message handler for the `service.messages.RequestDeadline` message. Upon receipt of this message, the broker should send an `ApplicationResponse` message back to the client that combines both the clientInfo (of the initial request) and the list of quotations.

Task 5: Implement the Client **Grade: A**

The final task is to implement he client. This requires the creation of a Client Actor that is able to receive and output the contents of ApplicationResponse messages. The client code should also send ApplicationRequest messages to the broker.

Additional Marks

+ grades (e.g. A+) can be attained through consideration of boundary cases, good exception handling, nice features that enhance the quality of your solution.

- grades (e.g. A-) can be attained though lack of commenting and indentation, bad naming conventions or sloppy code.