COMP30220 Distributed Systems Practical

Lab 6: Actor-based Distribution

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

You should use the basic version of LifeCo as a starting point for this lab. As a reminder, you can download the LifeCo source code from: https://gitlab.com/ucd-cs-rem/distributed-systems/lifeco-basic.

The broad objective of this practical is to adapt the LifeCo system to use the Actor Model. We will use Akka to implement the system. 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.

As with the previous labs, I have broken the problem up into a set of tasks. My advice is to create a separate project for your solution, and for you to copy code from the original project as needed.

Each task should be completed in turn and **your submission for each task must be stored in a separate folder** called "taskX" where X is the task number. For example, your answer for Task 1 should be stored in a folder called "task1", while your answer for Task 2 should be stored in a folder called "task2". When moving onto the next task, simply duplicate and rename the previous task folder to represent the next task (i.e. after completing Task 1, copy the folder "task1" and rename the copy "task2". Modify the code in "task2" as required to complete the task.

Task 1: Setting up the Project   Grade: E

As in the previous lab the first task is to break the original codebase up into a set of projects: one of each of the web services we are going to create; a project for the client and a shared core project that contains the code that is common across the web services. Again, we will also set up the core project.

a) You should create a folder "lifeco-akka" and inside that create a folder called "task1" which will be the root of the project file structure. As before, this project should contain the following sub-folders:
   - **core**: contains the common code (distributed object interfaces, abstract base classes & any data classes)
   - **auldfellas**: The Auldfella's Quotation Service
   - **dodgygeezers**: The Dodgy Geezers Quotation Service
   - **girlsallowed**: The Girls Allowed Quotation Service
   - **broker**: The broker service
   - **client**: The client service

   You should create a multi-module pom.xml file in the root folder of the project that should reference each subfolder. A standard "jar" packaging pom.xml file should be added to each subfolder. The artifactId for each sub-project should be the same as the name of the folder containing the subproject (e.g. the "core" projects artifactId should be "core" and the "girlsallowed" project artifactId should be "girlsallowed".) The artifactId for the main project should be based on the task (e.g. for this task it should be "task1" while for the next task it should be "task2"). The groupId should be "lifeco-akka".

b) Next, copy the "service.core" package from the basic project into "src/main/java" folder of the core folder. Remember that you should replicate the package structure. Perform the following modifications:
   - Delete the `Constants` class.
   - Remove all references to the `ServiceRegistry` and `Service` classes.
   - Delete the `QuotationService` and `BrokerService` interfaces.
   - Make sure that the `Quotation` and `ClientInfo` classes implement the default constructor.

- Compile and install the “core” project

Task 2: Designing the Quotation Services & Implementing Auldfellas **Grade: D**

This task involves the creation of an actor for the quotation services. Remember, the Actor Model is message oriented. The actor that implements the quotation service should work as follows: On receipt of a message containing an application for a quote, the actor should generate a quotation and send the quotation back to the sender of the message.

We could potentially do this by simply using the `ClientInfo` and `Quotation` classes as messages. However, the actor model has the same issue as message-oriented middleware: messages are ***asynchronous***. That means that there is no link between an actor sending a message and receiving a message. The actor must manage this itself. The solution is to use the same messages that were used in the message-oriented middleware solution. For the quotation service, these were `ClientMessage` and `QuotationMessage`:

```
public class ClientMessage {
    private long token;
    private ClientInfo info;

    public ClientMessage(long token, ClientInfo info) {
        this.token = token;
        this.info = info;
    }

    public long getToken() {
        return token;
    }

    public ClientInfo getInfo() {
        return info;
    }
}

public class QuotationMessage {
    private long token;
    private Quotation quotation;

    public QuotationMessage(long token, Quotation quotation) {
        this.token = token;
        this.quotation = quotation;
    }

    public long getToken() {
        return token;
    }

    public Quotation getQuotation() {
        return quotation;
    }
}
```

Notice that both messages include a `token`. This is used to match the request to the response. These classes should be created in the `service.message` package in the **core** project because the will be used in multiple projects.

a) First, remember to duplicate the “task1” folder creating a new folder called “task2”. All the changes discussed here should be made to “task2”.

b) Copy the Auldfellas service from the “lifeco-basic” project into the auldfellas project. Remember to maintain the “service.auldfellas” package structure.

c) To support the use of Akka, add the following dependencies to the `pom.xml`:

```
<dependency>
  <groupId>com.typesafe.akka</groupId>
  <artifactId>akka-actor_2.12</artifactId>
  <version>${akka.version}</version>
</dependency>
```

The `akka.version` property should be set to `2.6.20`.

Note: the dependency should be added to the existing `<dependencies>` block. Also, you can replace the existing exec-maven-plugin plugin as we will not be using that in this project.

d) Create a class called Quoter in the "service" package with the following code:

```
package service;

public class Quoter extends AbstractActor {
  private AFQService service = new AFQService();

  @Override
  public Receive createReceive() {
    return new ReceiveBuilder()
      .match(ClientMessage.class,
        msg -> {
          Quotation quotation = service.generateQuotation(msg.getInfo());
          getSender().tell(new QuotationMessage(msg.getToken(), quotation), getSelf());
        })
        .build();
  }
}
```

This code implements a `Quoter` actor that responds to the receipt of a `ClientMessage` by generating a `Quotation` and sending a `QuotationMessage` back to the sender of the `ClientMessage`, including both the quotation and the token for reference.

e) We can test this code by creating an Akka Unit Test:

```
public class QuoterTests {
  static ActorSystem system;
  @BeforeClass public static void setup() {system = ActorSystem.create(); }
  @AfterClass public static void teardown() {
    TestKit.shutdownActorSystem(system); system = null;
  }

  @Test public void quoterTest() {
    final Props props = Props.create(Quoter.class);
    final ActorRef subject = system.actorOf(props);
    final TestKit probe = new TestKit(system);
    subject.tell(
      new ClientMessage(1l,
        new ClientInfo("Niki Collier", ClientInfo.FEMALE, 49, 1.5494, 80, false, false)),
      probe.getRef());
    probe.expectMsgClass(Duration.ofSeconds(2), QuotationMessage.class);
  }
}
```

This code creates an actor system and then runs the `quoterTest()` test, which creates a quoter actor and a test probe actor and then sends a `ClientMessage` to the quoter actor identifying the probe actor as the sender. The test then checks whether the probe actor receives a `QuotationMessage` object within 2 seconds.We can test this code by creating an Akka Unit Test:

f) Repeat for GirlsAllowed and DodgyGeezers.

Task 3: Build the Broker Service **Grade: C**

Repeat the solution presented in task 2 but for the GirlsAllowed and DodgyGeezers services.

Now we have working quotation services, the next task is to create the broker actor for the broker service. For this task, you will have to again take inspiration from the message-oriented middleware solution you developed previously. Specifically, you need to cater for the asynchronous nature of message passing but creating tokens to allow you to match client applications to quotations. You should use the caching concept that was used and include a timeout after which the response is sent. If you want to be smart here, you could use information about the number of expected quotations vs the number of received quotations to shortcut the timeout and send the response as soon as it is ready…

For the basic solution, I recommend you use the `ClientInfo` class as a message type (for client->broker interaction). The messages for broker->quoter and quoter->broker interaction were specified in task 2. For broker->client interaction, I recommend the message below:

```
public class OfferMessage implements java.io.Serializable {
    private ClientInfo info;
    private LinkedList<Quotation> quotations;

    public OfferMessage(ClientInfo info, LinkedList<Quotation> quotations) {
        this.info = info;
        this.quotations = quotations;
    }

    public ClientInfo getInfo() {
        return info;
    }

    public LinkedList<Quotation> getQuotations() {
        return quotations;
    }
}
```

As in the previous task, you should implement this actor as a stand-alone actor and write a unit test that checks that the actor responds to a `ClientInfoMessage` message by sending back a `OfferMessage` within a given window of time (based on the specified timeout). The `ClientInfoMessage` message should include a reference to a `ClientInfo` object. The expectation is that the `OfferMessage` will contain no quotations (for this task). You can check for this, but it is not required.

You will need an internal (to the Broker) `TimeoutMessage` that can be used make the agent sleep for a fixed time period:

```
public class TimeoutMessage {
    private long token;

    public TimeoutMessage(long token) {
      this.token = token;
    }

    public long getToken() {
        return token;
    }
}
```

This message can be created in the broker codebase (still in the `service.message` package) as it is only used by the Broker.

Another problem to solve here is how to connect the broker and the quoter actors. This can be done by getting the quoter actors to register with the broker using a `RegisterMessage` message:

```
public class RegisterMessage {}
```

To recap the expected behaviour of this actor:

a) upon receipt of a `Register` message, the actor should store the actor reference of the sender (it will be a quoter actor).
b) Upon receipt of a `ClientInfoMessage` message, the actor should create a token that is mapped to the senders (clients) actor reference and to a template `OfferMessage`. It should then send a `ClientMessage` to each registered quoter actor and create a Thread that goes to sleep for a fixed time period (say 2 seconds) and then sends a `TimeoutMessage` to itself.
c) Upon receipt of a `TimeoutMessage` message, the actor should use the provided `token` to retrieve the client actor reference to whom it should send the associated `OfferMessage` message.
d) Upon receipt of a `QuotationMessage`, the broker should add the provided `Quotation` to the `OfferMessage` linked to the associated `token`.

Task 4: Implementing the Client & Distributing the Actors   Grade: B

The final tasks are to implement the client and to distribute and test the codebase.

a) Distributing the Broker requires Akka Remoting which uses the following dependencies:

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

You will also need to create a bespoke interface to support message serialisation (this must be added to the core project):

```
package service.message;

public interface MySerializable { }
```

All messages must implement this interface.

You need to add an `application.conf` file to `src/main/resources` (this is an Akka configuration file) with the following content:

```
akka {
  actor {
    provider = cluster
    serialization-bindings {
      "service.message.MySerializable" = jackson-json
    }
    serializers {
      jackson-json = "akka.serialization.jackson.JacksonJsonSerializer"
    }
  }

  remote.artery {
    enabled = false
```

```
      transport = tcp
    }

    remote.classic {
      enabled-transports = ["akka.remote.classic.netty.tcp"]
      netty.tcp {
        hostname = "localhost"
        port = 2550
        enable-ssl = false
      }
   }
  }
```

Finally, you should write a `main()` method in a class called `service.Main`. This class should create the `ActorSystem` and also the `Broker` actor.

b) Repeat the above for each of the quoter services with the following differences:

   a. Change the port in the `application.conf` file (auldfellas=2551, girlsallowed=2552, dodgygeezers=2553).

   b. Create a `Quoter` actor instead of the `Broker` actor

   c. Create a remote actor reference (see notes) to the `Broker` and send a `RegisterMessage` message to the Broker for each `Quoter`.

c) Create a `Client` actor that handles one message: the `OfferMessage` message whose contents should be printed out to the system console on receipt. The main method should send a `ClientInfoMessage` message to the broker for each `ClientInfo` object in the `clients` array, providing the actor reference of the client actor in the reply to field.

Task 5: Containerisation **Grade: A**

This task involves containerisation of the LifeCo-Akka system. The goal is again to dockerise all the services, but not the client. A `docker` file should be provided. The client should be run using Maven. Remember to make the folder "task5".

The only hint I will give you on this is that you need to adapt the `application.conf` as follows:

```
  remote.classic {
    enabled-transports = ["akka.remote.classic.netty.tcp"]
    netty.tcp {
      hostname = "localhost"
      hostname = ${?HOSTNAME}
      port = 2554
      enable-ssl = false
    }
 }
```

Here the HOSTNAME can be optionally set (to override "localhost") by the HOSTNAME environment variable (which should be the name of the container).