COMP30220 Distributed Systems Practical

Lab 4: Message-Oriented Middleware-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 Message-Oriented Middleware for interaction between each of the 3 quotation services and the broker and between the broker and the client. Specifically, we will use JMS to implement inter-service interaction using asynchronous message passing via topics or queues. 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-jms" 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-jms".

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. Peform 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 `java.io.Serializable` interface.
- Compile and install the “core” project

Task 2: Designing and Preparing the System **Grade: D**

The second task involves designing how the system will work and setting up the codebase to support that design. Implementing the service is a little more complex that for the previous labs because we are moving to use asynchronous messaging.

In synchronous messaging, the client sends the request and then waits for the response. In asynchronous messaging, the client does not wait for the response. In fact, a response is not necessary.

The consequence of this is that there is no direct link between request and response and that a response message received is not guaranteed to be related to the last request sent. In such cases, mechanisms must be put in place to allow different parts of the system to connect related messages when necessary. You can think of the interactions in this type of system as more like flows than request-response pairs.

So, the overall design follows the following flow: `ClientMessage` messages are generated by the client containing client information which are then consumed (published) via a TOPIC called APPLICATIONS. At the other end of the flow, the client consumes `OfferMessage` messages from a QUEUE called OFFERS. `OfferMessage` messages combine the `ClientInfo` and any `Quotation`s from `QuotationMessage` messages received in a fixed time period.

Quotation services consume messages from the APPLICATIONS topic. Specifically, they invoke the `generateQuotation(…)` method for each `ClientMessage` received, creating `Quotation` object. The quotation service then produces a `QuotationMessage` message that contains this object that it places on a QUEUE called QUOTATIONS. The broker service monitors both the APPLICATIONS topic and the QUOTATIONS queue and is responsible for matching Quotations to Clients. After a fixed timeout, the broker service produces an OfferMessage message, which is consumed by the OFFERS queue.

A key requirement here is to be able to match the `QuotationMessage` messages to the `ClientMessage` messages that triggered their creation. We achieve this by associating each `ClientMessage` with a unique numerical token. This token should then be included in any subsequent `QuotationMessage` messages.

The steps below set up the message broker and create the messages required In the rest of the lab.

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) Create the following three message classes and add them to the core project:

```
package service.message;

import service.core.ClientInfo;

public class ClientMessage implements java.io.Serializable {
    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;
    }
}
```

```
package service.message;

import service.core.Quotation;

public class QuotationMessage implements java.io.Serializable {
    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;
    }
}
```

```
package service.message;

import java.util.LinkedList;
import service.core.ClientInfo;
import service.core.Quotation;

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;
    }
}
```

c) A third step we must perform in this first task is to provide the support necessary to deploy JMS based applications. Specifically, we need to install a Message Broker. For this module, we will use the Apache ActiveMQ product. The best way to do this is to use a pre-made docker image. For simplicity, we will do this using docker compose .

   Create a `docker-compose.yml` file in the root folder of the project and copy in the content below:

```
version: '3.6'

services:
  activemq:
    container_name: activemq
    image: rmohr/activemq:latest
    ports:
    - 8161:8161
    - 61616:61616
```

d) Run the docker compose file and log into the console (http://localhost:8161/admin) username/password = admin/admin

Task 3: Building the Quotation Services **Grade: C**

Now, we are ready to start implementing the quotation services. In this task, we will build and test the quotation services part of the system. As in previous labs, we will do this by first developing a solution with auldfellas and then applying the solution to the other services.

a) First, remember to duplicate the “task2” folder creating a new folder called “task2”. All the changes discussed here should be made to “task3”.

b) Next, copy the “service.auldfellas” package into the “src/main/java” folder of the auldfellas project.

At this point, the project should compile if you type in:

```
$ mvn compile -pl auldfellas
```

c) Next, add the ActiveMQ dependency to the Auldfellas pom.xml:

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-core</artifactId>
    <version>5.7.0</version>
</dependency>
```

d) Finally, create a Main class in the default package of the auldfellas project and lets add the code for linking the service to the messaging infrastructure. This class should include a static field that contains an instance of the AFQService() and an empty main method:

```
public class Main {
    private static AFQService service = new AFQService();

    public static void main(String[] args) {
    }
}
```

Now, the next step is to create a connection with the Message Broker. This can be achieved by the following 4 lines of code:

```
ConnectionFactory factory =
        new ActiveMQConnectionFactory("failover://tcp://localhost:61616");
Connection connection = factory.createConnection();
connection.setClientID("auldfellas");
Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
```

This code is a standard template with two bits that you might need to change in the future. In the first line of code, a URL references the Message Broker instance. The format of this URL is known as the failover transport protocol and it is specific to ApacheMQ. It allows the developer to supply a comma-delimited list of hosts. On connection, the client choses one of these hosts at random. If the connection fails, another host is chosen at random from the list. An example with two hosts is:

```
failover:(tcp://localhost:61616,tcp://remotehost:61616)?initialReconnectDelay=100
```

This protocol is how ActiveMQ delivers a resilient service given the message broker is a potential single point of failure. For this lab, we will only use one broker hence the format used in the code. That said, we are assuming that the host for the broker is localhost. When dockerising this service, you will need to modify the code to be able to specify it using an environment variable (but this should be done later in task 5).

The second part of the template that might need to change is the clientId. Each client connection must have a unique id. This means that the id will have to change for each service you implement. Where each service is unique, this is not a problem, but the issue become more challenging when you want to replicate services using the same code.

e) The next step is to connect the ActiveMQ client to the relevant queues (QUOTATIONS) and topics (APPLICATIONS). This is done through the following 4 lines of code

```
Queue queue = session.createQueue("QUOTATIONS");
Topic topic = session.createTopic("APPLICATIONS");
MessageConsumer consumer = session.createConsumer(topic);
MessageProducer producer = session.createProducer(queue);
```

f) Finally, we need to write the code that makes the service consume incoming `ClientMessage` message, generating a Quotation and then producing a `QuotationMessage` message that it submits to the QUOTATIONS queue. This can be achieved by the use of a `MessageListener` instance:

```
consumer.setMessageListener(new MessageListener() {
  @Override
  public void onMessage(Message message) {
    try {
      ClientMessage request = (ClientMessage)
                      ((ObjectMessage) message).getObject();
      Quotation quotation = service.generateQuotation(request.getInfo());
      Message response =
            session.createObjectMessage(
                new QuotationMessage(request.getToken(), quotation));
      producer.send(response);
      message.acknowledge();
    } catch (JMSException e) {
      e.printStackTrace();
    }
  }
});
```

The first line converts the Message object received to an ObjectMessage (here we are assuming that APPLICATIONS will only ever produce ObjectMessage message. It then extracts the content of the message, which is assumed to be a ClientMessage. Again, we are assuming that only one type of content will ever be send on this topic. If a non-ObjectMessage or non-ClientMessage message is received, then you will get a ClassCastException. You can use the *instanceof* operator to put in additional checks to stop this happening (or to provide support for different message types, but we do not need to do that here.

The second line of code generates the quotation; the third line creates a new ObjectMessage and rthe fourth line of code publishes this message on the QUOTATIONS queue.

g) To test this code, we can again make use of the Junit unit testing framework. Set it up as normal (add the junit dependency, create a src/test/java folder). Create the following class:

```
public class QuotationTest {
  @Test
  public void testService() throws Exception {
    Main.main(new String[0]);

    ConnectionFactory factory =
       new ActiveMQConnectionFactory("failover://tcp://localhost:61616");
    Connection connection = factory.createConnection();
    connection.setClientID("test");
    Session session =
        connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);

    Queue queue = session.createQueue("QUOTATIONS");
    Topic topic = session.createTopic("APPLICATIONS");
    MessageConsumer consumer = session.createConsumer(queue);
    MessageProducer producer = session.createProducer(topic);
    connection.start();

    producer.send(
      session.createObjectMessage(
        new ClientMessage(1L, new ClientInfo("Niki Collier",
                          ClientInfo.FEMALE, 49, 1.5494, 80, false,
                          false))));
    Message message = consumer.receive();
    QuotationMessage quotationMessage =
        (QuotationMessage) ((ObjectMessage) message).getObject();
    System.out.println("token: " + quotationMessage.getToken());
    System.out.println("quotation: " + quotationMessage.getQuotation());
    message.acknowledge();
    assertEquals(1L, quotationMessage.getToken());
  }
}
```

Notice that the first line of the unit test invokes the main() method we wrote earlier. The rest of the code is similar to what we put into the Main class with the exception that we are now sending a message to the APPLICATIONS topic and consuming a message from the QUOTATIONS queue. Also, we could not use the `MessageHandler` interface because the code does not block and the test finishes (and succeeds) before the `QuotationMessage` message is received. Instead we use the older approach that uses the receive() method.

h) Run the unit test. Don't forget to start the ActiveMQ message broker (sometimes this may cause a problem where you ran the broker for a previous task. You can use:

```
$ docker rm activemq
```

To remove the old container. This will allow docker compose to create a new container based on the new task.

i) Finally, repeat all of the above for DodgyGeezers and GirlsAllowed.

Task 4: Implementing the Broker and Client **Grade: B**

Now we have working quotation services, the next task is to create the broker and the client. By far the harder part of this task is creating the broker, which you should do first. The client code is relatively simple and involves producing messages for the APPLICATIONS topic and consuming messages from the OFFERS queue. Remember to create a new folder "task4" for the work you do here.

The broker code is more complex in that the broker consumes messages from both the APPLICATIONS topic and the QUOTATIONS queue. It then produces messages for the OFFERS queue (which are consumed by the client). The idea is to combine the content received from the APPLICATIONS topic and the QUOTATIONS queue to create `OfferMessage` messages that can be send to the OFFERS queue. While the client ultimately creates the tokens that are associated with each `ClientMessage` message, this token is used by the broker service to match the ClientInfo to corresponding quotations. The broker should set a timelimit for how long it will wait to receive quotations (say 3 seconds) before sending what it has to the OFFERS queue.

For this task, you don't need to copy any code from LifeCo-basic. You simply reconstruct the behaviour in a new Main class. A key challenge in the implementation is the waiting for a time limit. This must be implemented using a Thread. Threads are independent execution sequences within a program. They are what makes it possible for a program to do 2 (or more) things at the same time. Threads are easy to create in Java but difficult to debug. One of the easiest ways to create a thread is to create an anonymous object that instantiates a subclass the Thread class:

```
System.out.println("A");
new Thread() {
  public void run() {
    // this code is in the thread
    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("B");
  }
}.start();
System.out.println("C");
```

The above snippet of code prints out A, creates a Thread that sleeps for 3 seconds and then prints out B, and a third line that prints out C. The display order of the output is: ACB because the main thread creates a second thread that sleeps, but itself continues executing. So C is printed before B because the second thread sleeps for 3 seconds before printing B.

You should modify this code to implement a behaviour that on consuming a `ClientMessage`, the broker service creates a partially completed `OfferMessage`, creates a thread that waits for 3 seconds and then sends whatever is in the `OfferMessage`. Quotations are added to the `OfferMessage` in parallel based on the messages consumed from the QUOTATIONS queue. To make the `OfferMessage` messages available to this consumer, all `OfferMessage` messages are stored in a temporary cache (this is a Java Map with a Long key and `OfferMessage` value).

Task 5: Containerisation **Grade: A**

This task involves containerisation of the LifeCo-JMS system. The goal is again to dockerise all the services, but not the client. All the services should be integrated with the docker-compose.yml file provided in task 2. The client should be run using Maven. Remember to make the folder "task5"