COMP30220 Distributed Systems Practical

Lab 2: RMI-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.

Please read AND run the LifeCo scenario before attempting this practical. You can download the LifeCo source code from: https://gitlab.com/ucd-cs-rem/distributed-systems/lifeco-basic. There is also a document describing the scenario in the root folder of this project.

The broad objective of this practical is to adapt the LifeCo system to use RMI 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 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.

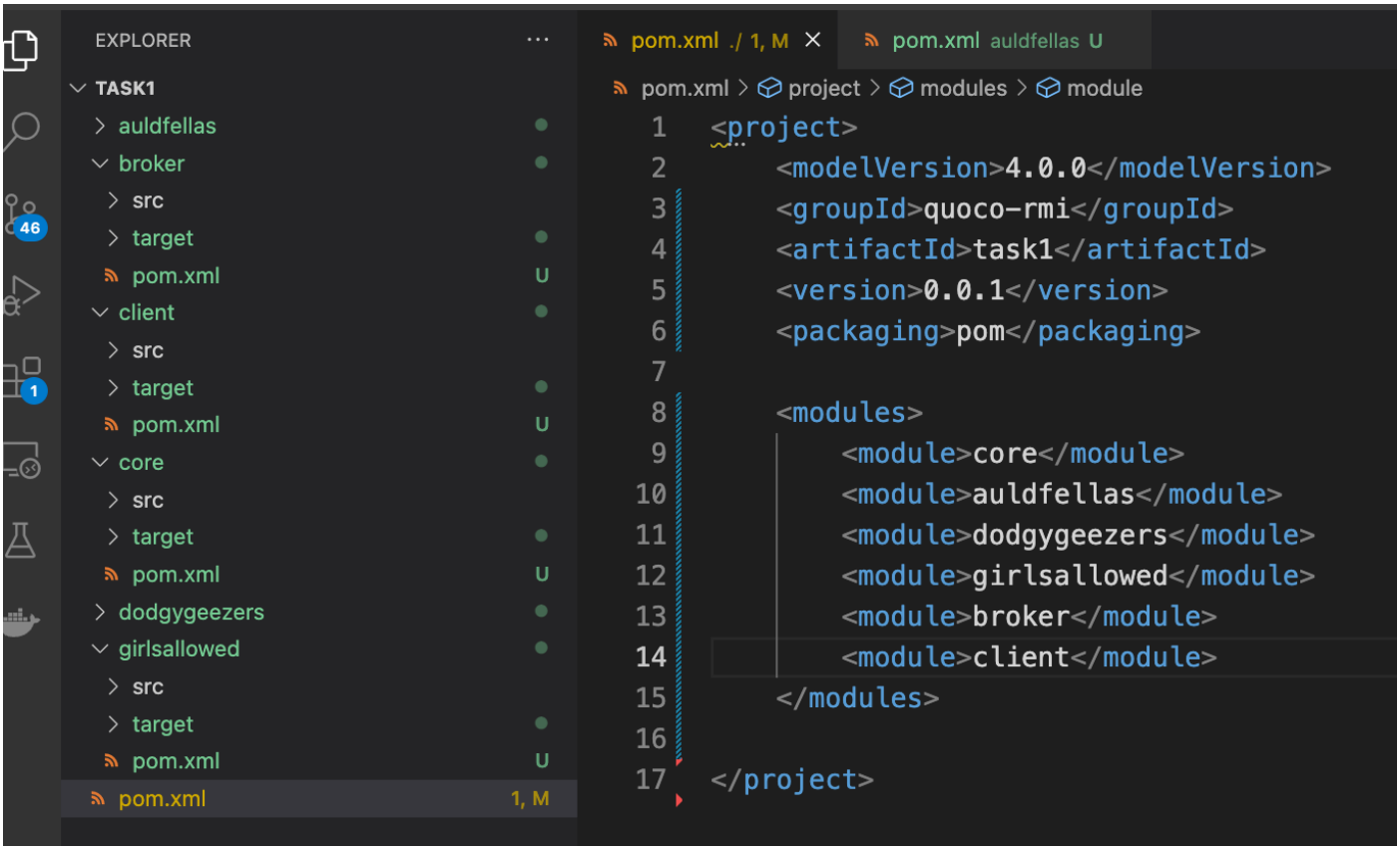
Task 1: Setting up the Project Structure **Grade: E**

To create a distributed system, we need to break the original codebase up into a set of projects: one of each of the distributed objects we are going to create, and a core project that contains the code that is common across those distributed objects. These projects should be created as modules within a single multi-module maven project. Please watch the video on how to set up such a project if you are not sure how to do it.

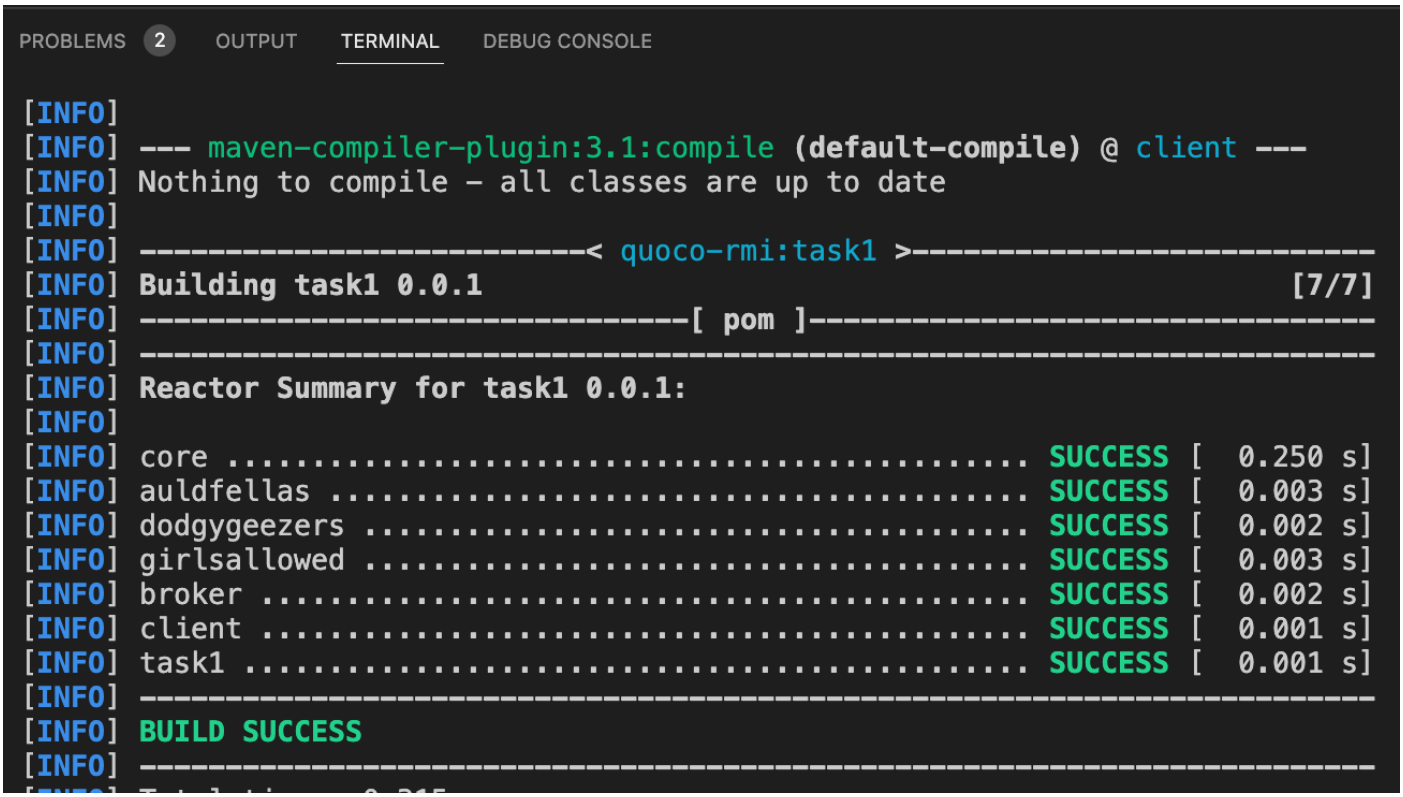
Based on this, you should create a folder called "task1" for the main project and add a set of subfolders as follows:

- **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. Use the "calculator" project from class as an example, noting that the groupId in the screenshot below. 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").



Running "`mvn compile`" in the project root should succeed and produce an output like that shown below. ***If it does not compile, then the project is not correctly configured and you need to talk to a demonstrator.***



Task 2: Creating the Core project **Grade: D**

The core project contains the shared code that is used across multiple projects. Typically, in an RMI project, this includes all the interfaces that will be implemented to create the distributed objects. It also typically contains any data classes that are associated with those interfaces. All data classes must implement `java.io.Serializable` so that they can be sent over a network connection.

Remember to create a copy of the "task1" folder called "task2". Make changes to the "task2" folder only.

The following steps will help you to transfer the existing code to the correct projects

a) Copy the "service.core" package into "src/main/java" folder of the "core" project. There should be some compilation errors when you do this.

b) Fix the errors by removing the reference to the `service.registry.Service` interface.

c) Now modify the `BrokerService` & `QuotationService` interfaces to extend `java.rmi.Remote` and for the method signatures to throw `java.rmi.RemoteException` (this is the same as we did when we created the Calculator interface in class).

d) Make the Quotation and ClientInfo data classes implement the `java.io.Serializable` interface.

e) Compile the (multi-module) project***. If it does not compile, you have done something wrong, so retrace the steps above.***

Task 3: Creating and Testing the Distributed Quotation Services **Grade: C**

The third task involves creating a distributed version of the Quotation Services. I will start by explaining how to do it for one of the services – auldfellas – and you will need to do the same thing for the other services.

a) Copy the auldfellas package into the "src/main/java" folder of the "auldfellas" module. By this, I mean, create a subfolder in the java folder called "service" and create a subfolder in that folder called "auldfellas" and then copy the AFQService.java file into "src/main/java/service/auldfellas". *Notice that I am asking you to maintain the package structure from the original code in the new codebase.*

b) To test the creation of a distributed object, you can use a Junit unit test. To do this, you must create a new folder structure: "src/test/java" (notice that the "test" folder is a sibling of the "main" folder). You must also add the following dependency to your auldfellas pom.xml file:

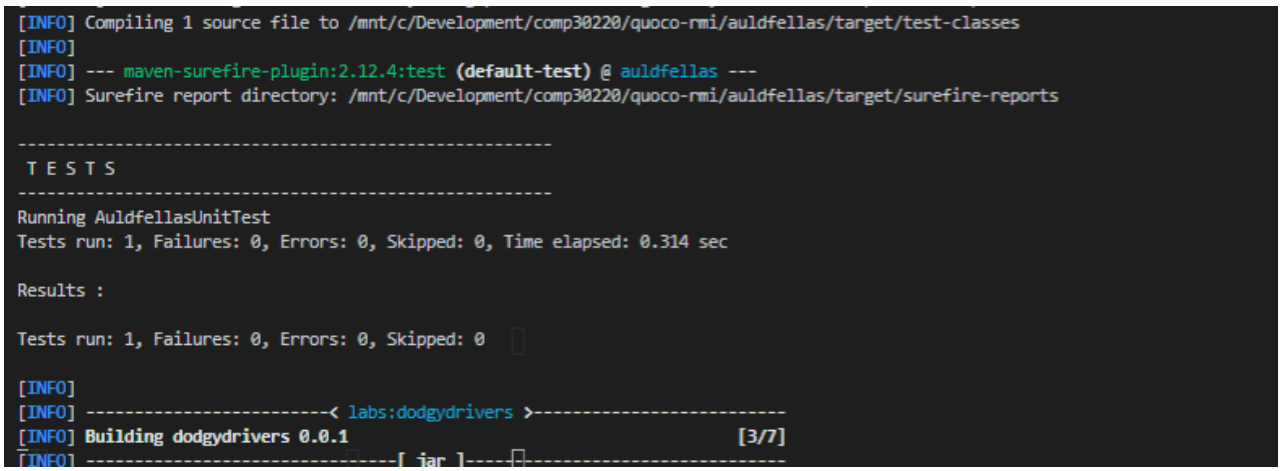
```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
```

Now, copy the code below into a file called `AuldfellasUnitTest.java` that should be created in the "src/test/java" folder:

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

import service.core.Constants;
import service.core.QuotationService;
import service.auldfellas.AFQService;

import org.junit.*;
import static org.junit.Assert.assertNotNull;

public class AuldfellasUnitTest {
    private static Registry registry;

    @BeforeClass
    public static void setup() {
        QuotationService afqService = new AFQService();
        try {
            registry = LocateRegistry.createRegistry(1099);

            QuotationService quotationService = (QuotationService)
                UnicastRemoteObject.exportObject(afqService,0);

            registry.bind(Constants.AULD_FELLAS_SERVICE, quotationService);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }

    @Test
    public void connectionTest() throws Exception {
        QuotationService service = (QuotationService)
            registry.lookup(Constants.AULD_FELLAS_SERVICE);
        assertNotNull(service);
    }
}
```

The test basically creates an RMI Registry followed by a DistributedObject based on the AFQService, which it registers with the registry. The unit test (connectionTest()) then looks up the service that you registered. The test passes if an object is returned and fails if null is returned (the object was not found in the registry).

To run the unit test, simply type “mvn test” in the root folder. You should get a lot of output, specifically look for the section below:



If the test fails, then the Maven build will stop and report the failure.

c) Add another unit test that checks whether the generateQuotation() method works (and returns a Quotation object). You should copy and paste one of the example ClientInfo objects from the client.Main file in the original codebase. The output should look similar to part (b) with the exception that “Tests run: “ will now be 2.

d) Now that we have tested that our “auldfellas” quotation service can be deployed and executed using RMI, we need to create a main method to deploy it for use in our new system. This is done through the code below. You will notice that the method looks a lot like the setup() method we used in the unit test. Create a class in the default package called `Main.java`. Copy the code below into it.

```
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;

import service.auldfellas.AFQService;
import service.core.QuotationService;
import service.core.Constants;


public class Main {
    public static void main(String[] args) {

        QuotationService afqService = new AFQService();
        try {
            // Connect to the RMI Registry - creating the registry will be the
            // responsibility of the broker.
            Registry registry = LocateRegistry.createRegistry(1099);

            // Create the Remote Object
            QuotationService quotationService = (QuotationService)
                UnicastRemoteObject.exportObject(afqService,0);

            // Register the object with the RMI Registry
            registry.bind(Constants.AULD_FELLAS_SERVICE, quotationService);

            System.out.println("STOPPING SERVER SHUTDOWN");
```

```
            while (true) {Thread.sleep(1000); }
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }
}
```

Try to understand what this class is doing – it is basically the same as the CalculatorServer class, but that it is creating the Auldfellas Quotation Service distributed object. Notice the infinite loop at the end. This code is necessary because different versions of Java can have differences in how they behave. Some versions of Java will block at the end of the execution of the main method while others will terminate (even though you have created objects). After completing part (e), try commenting out these lines to see what happens. Compare your results with your classmates.

e) Finally, lets try to run this project. When using a multi-module project, you can target a subset of modules by using the -pl flag (followed by a comma-delimited list of module names). For example, we can perform the exec:java goal on the auldfellas project by using the following command:

```
$ mvn exec:java -pl auldfellas
```

However, running this will cause an error:

```
[ERROR] Failed to execute goal on project auldfellas: Could not resolve dependencies for project labs:auldfellas:jar:0.0.1:
 Failure to find labs:core:jar:0.0.1 in https://repo.maven.apache.org/maven2 was cached in the local repository, resolution
 will not be reattempted until the update interval of central has elapsed or updates are forced -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/DependencyResolutionException
rem@Krytan:/mnt/c/Development/comp30220/quoco-rmi$
```

This happens because there is a dependency between the core and auldfellas modules. When you compile the project, the “Reactor” system takes care of this problem, but when you use the custom exec:java goal, it does not.

To run the auldfellas module, you need to first install the core module in the local maven repository. You do this my using the `mvn install -pl core` command in the project root. Once you have run this install command, you can then run the auldfellas service using the earlier maven command. You may find that this code does not run and may get an error like that shown below:

```
[WARNING]
java.lang.ClassNotFoundException: client.Main
    at java.net.URLClassLoader.findClass (URLClassLoader.java:476)
    at java.lang.ClassLoader.loadClass (ClassLoader.java:589)
    at java.lang.ClassLoader.loadClass (ClassLoader.java:522)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run (ExecJavaMojo.java:270)
    at java.lang.Thread.run (Thread.java:829)
[INFO]
```

The error occurs because the codehaus plugin (exec:java) in the pom.xml file is incorrectly configured. Specifically, the issue arises with the `<mainClass>` parameter of the plugin. This may refer to a class that does not exist or (as is shown below) refer to a property that specifies a class that does not exist. To make the code work either update the property to be `Main`, or simply replace whatever is written inside the `<mainClass>` parameter with `Main`. This will fix the problem.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>1.6.0</version>
    <configuration>
        <mainClass>${main.class}</mainClass>
    </configuration>
```

f) Once you are convinced that service works, you will need to make a last change to the Server class so that it is possible to point the service at an existing registry rather than having to create one every time– in the final system, the broker will create the registry, and the quotation services will register with it.

Replace:

```
Registry registry = LocateRegistry.createRegistry(1099);
```

With:

```
Registry registry = null;
if (args.length == 0) {
    registry = LocateRegistry.createRegistry(1099);
} else {
    registry = LocateRegistry.getRegistry(args[0], 1099);
}
```

To run auldfellas with its own registry, you still use:

```
$ mvn exec:java -pl auldfellas
```

To run auldfellas with an existing registry, you can use:

```
$ mvn exec:java -pl auldfellas -Dexec.args="localhost"
```

Some operating systems may use a slight variation:

```
$ mvn exec:java -pl auldfellas "-Dexec.args=localhost"
```

g) To get a B+, repeat the above to create and test the "girlsallowed" and "dodgygeezer" projects.

NOTE: To run multiple modules at the same time, you will need multiple terminal windows. When you run one of the quotation services, it blocks until you kill it (using CTRL-C). That is why you need a different terminal for each service. A good test to do when you have two services working independently is to run one service so that it creates a registry and the other service so that it uses the created registry (you cannot create two registries at the same time because they use the same port).

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

Now we have working quotation services, the next task is to create and test the broker.

a) Expose the LocalBrokerService as a distributed object and modify the local broker service to use the RMI Registry to find the quotation services. This should all be implemented in the "broker" project. Write unit tests to check that the code works (it should return an empty list of quotations if no services are registered).

b) Modify the test client to lookup the broker and modify the main() method to loop through and print out all the quotations returned by the broker service.

c) Compile and run both projects 😊

Task 5: Containerisation **Grade: A**

The final task is to convert the multi module project you have developed through the first 4 tasks into a set of docker images that can be run from a docker compose file. **You should make both the distributed objects and the client into docker images.** An additional complication is that RMI does not allow you to register distributed objects on remote servers (and docker images are treated as remote servers). I explain how to solve this in the class notes. You will have to adapt my solution for this lab.

Note: there is an alternative plugin that you can use to combine the dependencies into a single file, it is known as the `maven-shade-plugin`. The code for the plugin is given below and should be added to each distributed object pom.xml file. The <Main-Class> entry should be modified to reflect the main class that is to be invoked when running the code.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.5</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <shadedArtifactAttached>true</shadedArtifactAttached>
        <shadedClassifierName>allinone</shadedClassifierName>
        <artifactSet>
          <includes>
            <include>*:*</include>
          </includes>
        </artifactSet>
        <transformers>
          <transformer
              implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
            <resource>reference.conf</resource>
          </transformer>
          <transformer
              implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
            <manifestEntries>
              <Main-Class>clock.ClockServer</Main-Class>
            </manifestEntries>
          </transformer>
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>
```

When `mvn package` is run, this plugin creates a file called `*-allinone.jar` which is a runnable jar file that can be executed using `java -jar *-allinone.jar`. This plugin is significantly faster than the `maven-assembly-plugin`.

Note: To make a java program run after a delay (say 5 seconds) use the following format for the CMD line in the Dockerfile:

```
CMD sleep 5 && java -jar /my.jar
```