# CMPU4021 Distributed Systems – Labs

## *Java RMI*

**Learning Outcomes:**

1. Be able to create remote objects using Java RMI.
2. Be able to use the rmiregistry naming service.
3. Be able to register and look up remote objects.
4. Be able to invoke remote methods.

## *Tasks*

**T1.**
Extract T1\Hello.java (a remote interface),
T1\HelloClient.java (the client - a simple client that invokes a method of the remote interface) and
T1\HelloServer.java (the server - a remote object implementation that implements the remote interface).

```
javac Hello.java Server.java Client.java
```

Examine the source code.

Open two new command prompts in the same directory where you have your compiled code. In one of these, start the naming service as shown. This is a process that runs constantly waiting for clients to request the stub for some remote object. Before you start the `rmiregistry` you must set the classpath of that dos session to include the current directory. This is done so that the registry can locate the stub class file. This would previously have been done as a switch to the java command.

```
:/> set CLASSPATH=%CLASSPATH%;.
```

In one window (or in the background), start the `rmiregistry`:

```
:/> start rmiregistry
```

**Note:** `rmiregistry` binds to port 1099 by default. If this port is in use, you can start the rmiregistry on another port. For example, to start the registry on port 2001 on a Windows platform, you would execute command :

```
:/> start rmiregistry 2001
```

If the registry will be running on a port other than 1099, you'll need to specify the port number in the calls to `LocateRegistry.getRegistry` in the `Server` and `Client` classes. For example, if the registry is running on port 2001 in this example, the call to `getRegistry` in the server would be:

```
Registry registry = LocateRegistry.getRegistry(2001);
```

At one of the other command prompts, start the server. The server will create your remote object (note the server is NOT itself the remote object), and register it with the naming service.

```
:/> java -classpath . Server
```

The output of Server will look like: `Server ready`

At your third command prompt, start the client. Look at the code to see how it performs lookup, receives the stub (notice how the stub can be referenced as a Hello (the interface)), and invokes the remote method.

```
:/> java -classpath . Client
```

The output of Client will look like: `response: Hello, world!`

**T2.** Extract, examine and compile the java files from T2 directory.

`ReverseClient.java is` client program that looks up to `"rmi://hostname:port/Reverse"` service. Takes the hostname of the serve and the port of the rmiregistry as arguments. `Reverse` is the name of the service on localhost.

`Reverse.java` implements the remote service. It contains an empty constructor and a function `InvertStr()` that takes a string as input and returns that string with its characters reversed.

`ReverseInterface.java` defines the remote interface that is provided by the service It is a single function that takes a string and returns a string.

`ReverseServer.java` provides the remote service and contacts rmiregistry with an instance of the service under the name "Reverse".

In one window (or in the background), start the `rmiregistry`:
`rmiregistry` binds to port 1099 by default. If this port is in use, you can start the rmiregistry on another port. For example:

```
    > rmiregistry 12345
```

Start the server in another window (or in the background)

```
> java ReverseServer 12345
```

Run the client in another window:

```
> java ReverseClient localhost 12345 abcd "DS RMI Reverse"
```

where the second argument is the port number of the `rmiregistry`. You should see the following output (the strings you gave reversed):

**dcba**
**esreverR IMR SD**

## T3. Shared whiteboard example

A shared whiteboard is a distributed program that allows a group of users to share a common view of a drawing surface containing graphical objects, each of which has been drawn by one of the users.
A server maintains the current state of a drawing by providing an operation for clients to inform it about the latest shape the users have drawn and keeping a record of all the shapes drawn so far. The server also provides operations for clients to add a shape, retrieve a shape and retrieve all the shapes.
The server has a version number (an integer) that it increments each time a new shape arrives and attaches it to the new shape. The server provides operations allowing clients to retrieve its version number or the version number of a shape, so that they may avoid fetching shapes they already have.

*Define the shared whiteboard interfaces in Java RMI.*

***Possible solution:***

`GraphicalObject` is a class that holds the state of graphical object (type, position); it must implement the Serializable interface.

Consider the interface Shape first: the `getVersion` method returns an integer, whereas the `getAllState` method returns an instance of the class `GraphicalObject`. Now consider the interface `ShapeList`: its `newShape` method passes an instance of `GraphicalObject` as its argument but returns an object with a remote interface (that is, a remote object) as its result. An important point to note is that both ordinary objects and remote objects can appear as arguments and results in a remote interface. The latter are always denoted by the name of their remote interface. In the next subsection, we discuss how ordinary objects and remote objects are passed as arguments and results.

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
     int getVersion() throws RemoteException;
     GraphicalObject  getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
     Shape newShape(GraphicalObject g) throws RemoteException;
     Vector allShapes() throws RemoteException;
     int getVersion() throws RemoteException;
}
```

**Further reading:**
https://docs.oracle.com/en/java/javase/19/docs/specs/rmi/objmodel.html#distributed-object-applications