EXPERIMENT NO. 04

AIM: To integrate software components using a middleware

THEORY:

Software architectures promote development focused on modular building blocks and their interconnections. Since architecture-level components often contain complex functionality, it is reasonable to expect that their interactions will also be complex. Modeling and implementing software connectors thus becomes a key aspect of architecture-based development. Software interconnection and middleware technologies such as RMI, CORBA, ILU, and ActiveX provide a valuable service in building applications from components. We have to understand the tradeoffs among these technologies with respect to architectures. Several off-the-shelf middleware technologies are used in implementing software connectors.

The Role of Middleware

Middleware is a potentially useful tool when building software connectors. First, it can be used to bridge thread, process and network boundaries. Second, it can provide pre-built protocols for exchanging data among software components or connectors. Finally, some middleware packages include features of software connectors such as filtering, routing, and broadcast of messages or other data.

Java's Remote Method Invocation (RMI) [24] is a technology developed by Sun Microsystems to allow Java objects to invoke methods of other objects across process and machine boundaries. RMI supports several standard distributed application concepts, namely registration, remote method calls, and distributed objects. Currently, RMI only supports Java applications, but there is indication of a forthcoming link between RMI and CORBA that would remedy this. Each RMI object that is to be shared in an application defines a public interface (a set of methods) that can be called remotely. This is similar to the RPC mechanism. These methods are the only means of communication across a process boundary via RMI. Because RMI is not a software bus, it has no

concept of routing, filtering, or messages. However, Java's RMI built-in serialization and deserialization capabilities handle marshalling of basic and moderately complex Java objects, including C2 messages. RMI is fully compatible with the multithreading capabilities built into the Java language, and is therefore well suited for a multithreaded application. It allows communication among objects running in different processes which may be on different machines. Communication occurs exclusively over the TCP/IP networking protocol. RMI supports application modification at run-time, a capability enabled by Java's dynamic class loading.

Addition of 2 numbers using JAVA RMI:

Define an interface that declares remote methods.

The first file AddServerIntf.java that defines the remote interface remains the same.

```
import java.rmi.*;

public interface AddServerIntf extends Remote {
  double add(double d1, double d2) throws RemoteException;
}
```

Implement the remote interface and the server

The second source file AddServerImpl.java (it implements the remote interface) also remains the same, with one minor variation: it calls the super-class constructor explicitly.

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

public class AddServerImpl extends UnicastRemoteObject
  implements AddServerIntf {

  public AddServerImpl() throws RemoteException {
  super();
  }

  public double add(double d1, double d2) throws RemoteException {
    return d1 + d2;
  }
}
```

The revised version of the third source file AddServer.java includes the security manager, assumes that you will run the server on jupiter using the port 56789, and uses a slightly modified name "MyAddServer" for registration purposes.

```
import java.net.*;
import java.rmi.*;
public class AddServer {
  public static void main(String args[]) {

    // Create and install a security manager

    if (System.getSecurityManager() == null) {
           System.setSecurityManager(new RMISecurityManager());
    }
    try {
      AddServerImpl addServerImpl = new AddServerImpl();

       // You want to run your AddServer on jupiter using the port 56789
       // and you want to use rmi to connect to jupiter from your local
machine
       // Note that to accomplish this you have to start on jupiter
       // rmiregistry 56789 &
       // in the directory that contains NO classes related to this server
!!!

      Naming.rebind("rmi://jupiter.scs.ryerson.ca:56789/MyAddServer",
addServerImpl);
    }
    catch (Exception e) {
           System.out.println("Exception: " + e.getMessage());
           e.printStackTrace();
    }
  }
}
```

Copy files **AddServerIntf.java**, **AddServerImpl.java**, and **AddServer.java** to your directory on jupiter and compile them as usual using javac.

Develop a client (an application or an applet) that uses the remote interface

The revised version of the fourth source file AddClient.java has a few new features.

- It includes also the security manager;

- it requires 4 command-line arguments: the name of the remote server, the port where the naming **rmiregistry** will be running, and two numbers;
- it prints the URL that will be used to connect to the server.

```
import java.rmi.*;

   // USAGE: java AddClient   firstNum secondNum

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

    // The client will try to download code, in particular, it will
    // be downloading stub class from the server.
    // Any time code is downloaded by RMI, a security manager must be present.

    if (System.getSecurityManager() == null) {
          System.setSecurityManager(new RMISecurityManager());
    }

    try {
      String addServerURL = "rmi://" + args[0] + ":" + args[1] +
"/MyAddServer";

     System.out.println("I will try to invoke the remote method from " +
addServerURL);

      AddServerIntf remoteObj =
                    (AddServerIntf) Naming.lookup(addServerURL);

      System.out.println("The first number is: " + args[2]);

      double d1 = Double.valueOf(args[2]).doubleValue();

      System.out.println("The second number is: " + args[3]);

      double d2 = Double.valueOf(args[3]).doubleValue();

      // Now we invoke from a local machine the remote method "add"

      System.out.println("The sum is: " + remoteObj.add(d1, d2));
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

Keep this file on your local machine together with the remote interface and the **rmi.policy** file that controls access to your local machine:

```
grant {
```

```
  // The simplest (but uncarefull) policy is allow everything:
  //    permission java.security.AllPermission;
  // More secure policy is the following.
  //
  // jupiter.scs.ryerson.ca   This is the rmihost - RMI registry and the
server
  // www.scs.ryerson.ca       This is webhost - HTTP server for stub classes
  permission java.net.SocketPermission
    "jupiter.scs.ryerson.ca:1024-65535", "connect,accept";
  permission java.net.SocketPermission
    "www.scs.ryerson.ca:80", "connect";

};
```

You also have to copy this policy file to your directory on jupiter that contains all server-related files.

Generate stubs and skeletons

Next, go to the server (jupiter), and change into the directory that contains **AddServerIntf.class** (interface), **AddServerImpl.class** (its implementation), and **AddServer.class** (server itself) and **rmi.policy** file. In that directory, run **rmic** compiler:

rmic AddServerImpl

This command generates two new files: **AddServerImpl_Skel.class** (skeleton) and **AddServerImpl_Stub.class** (stub).

Start the RMI registry

Before you proceed, copy stub and interface classes to a directory, where the web server can access them, e.g. to the world-accessible sub-directory **JavaClasses** of your **public_html** directory:

```
ls -t -l ~mes/public_html/JavaClasses/
total 24
-rw-r--r--   1 mes      mes          205     AddServerIntf.class
-rw-r--r--   1 mes      mes         1612     AddServerImpl_Skel.class
-rw-r--r--   1 mes      mes         3171     AddServerImpl_Stub.class
```

Check that all these files are accessible, e.g., try to download them using Netscape or IE: if you succeeded, then they are accessible.

On jupiter in the directory that does NOT contain any server related classes and assuming that those classes are NOT on you CLASSPATH, start rmiregistry:

rmiregistry 56789 &

For example, create the temporary directory tmpTEST, go to that directory and start there **rmiregistry** naming system. By default, rmiregistry naming system loads stub and skeleton files from directories mentioned in your **CLASSPATH**. But because you want to load them dynamically from your web directory `JavaClasses`, you want to hide these files from **rmiregistry**: this way you force to load required files from the codebase given below as a command-line argument.

Start the server

In the directory that contains all server related classes:

```
-rw-r--r--   1 mes      mes          1075       AddServer.class
-rw-r--r--   1 mes      mes           363       AddServerImpl.class
-rw-r--r--   1 mes      mes          1612       AddServerImpl_Skel.class
-rw-r--r--   1 mes      mes          3171       AddServerImpl_Stub.class
-rw-r--r--   1 mes      mes           205       AddServerIntf.class
-rw-r--r--   1 mes      mes           511       rmi.policy
```

we can run the server:

```
java -Djava.security.policy=rmi.policy
  -Djava.rmi.server.codebase=http://www.scs.ryerson.ca/~mes/JavaClasses/
AddServer &
```

Note that the URL given to "codebase" ends with / and use your own login name, of course.

Run the client

Now, go to the directory of your **local computer** that contains only 3 files:

- **AddClient.class** (client),
- **AddServerIntf.class** (interface),
- **rmi.policy**: your policy file.

For example, you can create a new directory and copy there 3 files mentioned in this section.

This time, you would like to load the stub class dynamically from the server. Assume that the server side was developed by a different company, you are responsible only for the client application and you do not have access to the server-related files when you start your client. All you know is that the RMI server will be running on jupiter and you can connect to the registry at the port 56789 if you need to invoke remote methods on the server.

Finally, you can run the client application on your local machine:

Implementation:

server.py

```
import socket

HOST = "127.0.0.1"
PORT = 65432
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Client.py

```
import socket

HOST = "127.0.0.1"
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

Output:

```
C:\Users\Lab 305\Desktop\New folder>python tcp.py
Connected by ('127.0.0.1', 52077)

C:\Users\Lab 305\Desktop\New folder>
```

```
C:\Users\Lab 305\Desktop\New folder>python tcp-c.py
Received b'Hello, world'
```

Learning Outcomes: The student should have the ability to

LO1: understand middle wares

LO2: write middlewares

Course Outcomes: Upon completion of the course students will be able to develop middlewares

Conclusion:

I Learned about Middleware. Implemented simple Middleware in python using library sockets.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				