Lab Notes
CMPU4021 Distributed Systems

Java Remote Method Invocation

Java RMI

- Java mechanism for calling methods of objects which do not run on the same Java Virtual Machine.

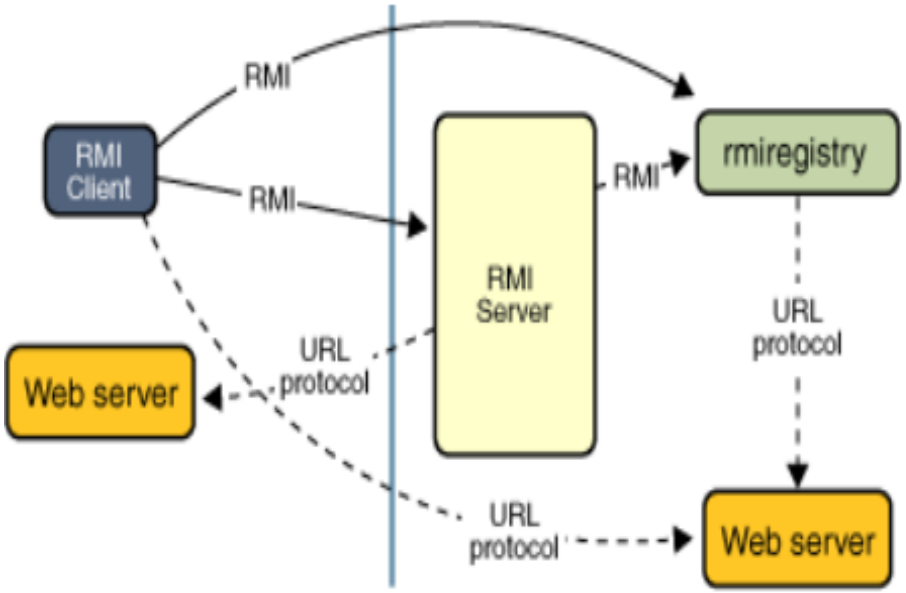
- Integral part of the Java language

Java RMI

- Remote interfaces are defined in the Java language.
- Allows objects to invoke methods on remote objects using the same syntax as for local invocations.
- Type checking applies equally to remote invocations as to local ones.
- An object making a remote invocation is aware that its target is remote because it must handle *RemoteExceptions*.
- Behaviour in a concurrent environment must be considered.

RMI Distributed Application

- Uses the RMI registry to obtain a reference to a remote object.
- The server calls the registry to associate (or bind) a name with a remote object.
- The client looks up the remote object by its name in the server's registry and then invokes a method on it.
- The RMI system uses an existing web server to load class definitions
  - from server to client and from client to server, for objects when needed.



Dynamic Code Loading

- Ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine.
- All of the types and behaviour of an object, previously available only in a single Java virtual machine, can be transmitted to another, possibly remote, Java virtual machine.
- RMI passes objects by their actual classes, so the behaviour of the objects is not changed when they are sent to another Java virtual machine.
- This capability enables new types and behaviours to be introduced into a remote Java virtual machine, thus dynamically extending the behaviour of an application.

JAVA RMI: Remote Interfaces, Objects, and Methods

- A distributed application built by using Java RMI is made up of interfaces and classes.
  - The interfaces declare methods.
- The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well.
- In a distributed application, some implementations might reside in some Java virtual machines but not others.
- Remote objects
  - Objects with methods that can be invoked across Java virtual machines are called.

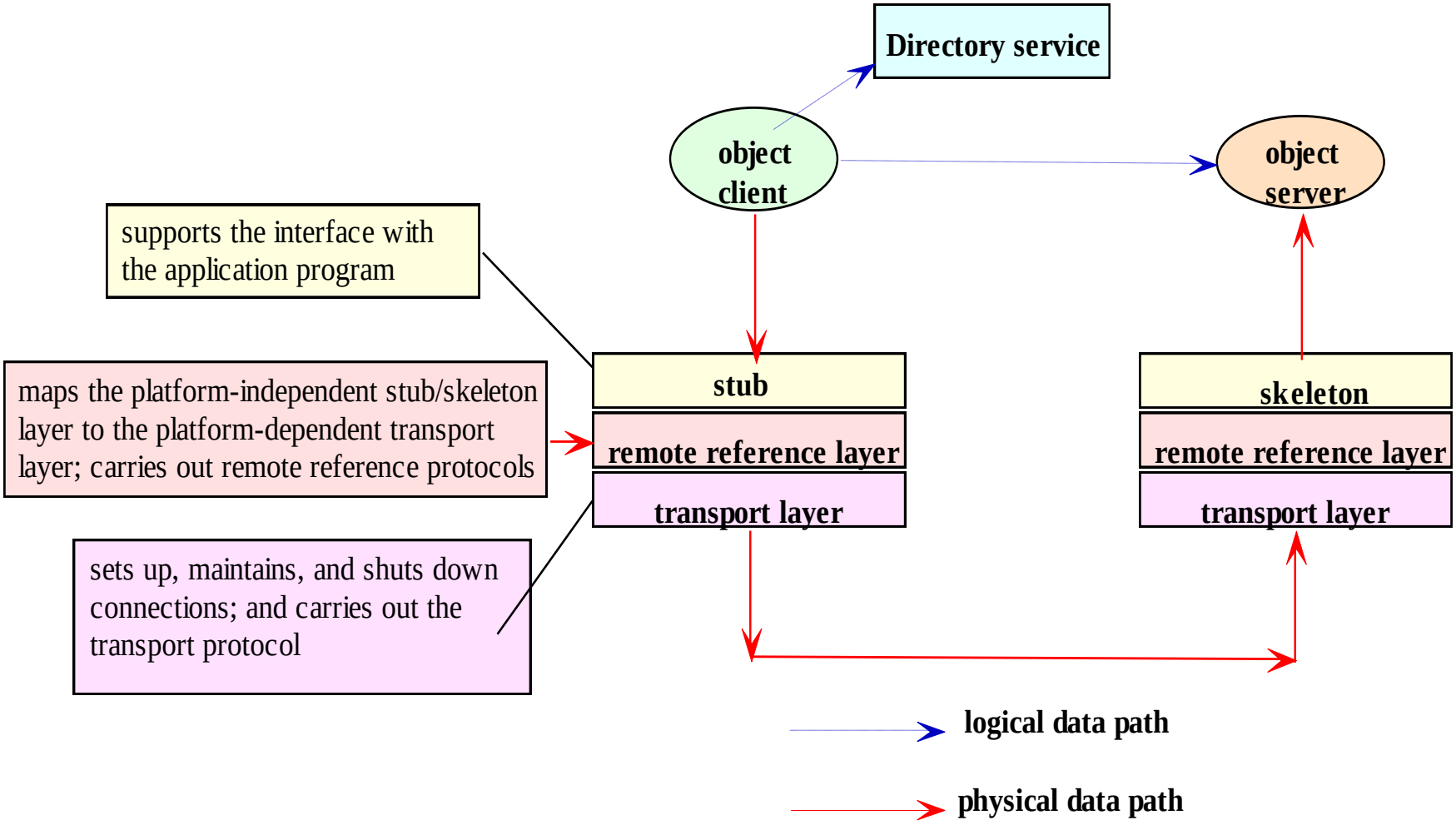
The Java RMI Architecture: Client-Side Architecture

1. The Stub layer
   - A client process's remote method invocation is directed to a proxy object, known as a stub. The stub layer lies beneath the application layer and serves to intercept remote method invocations made by the client program.
   - Then it forwards them to the next layer below, the Remote Reference Layer.
2. The Remote Reference Layer
   - Interprets and manages references made from clients to the remote service objects and issues the IPC operations to the next layer, the transport layer, to transmit the method calls to the remote host.
3. The Transport layer
   - Connection-oriented
   - Carries out the inter-process communication (IPC), transmitting the data representing the method call to the remote host.

The Java RMI Architecture: Server-side Architecture

1. The Skeleton layer
   - Lies just below the application layer and serves to interact with the stub layer on the client side.
2. The Remote Reference Layer
   - Manages and transforms the remote reference originating from the client to local references that are understandable to the Skeleton layer.
3. The Transport layer
   - Connection-oriented transport layer

The Java RMI Architecture



The Java RMI Architecture :Object Registry – Directory Service

- The RMI API allows a number of directory services to be used for registering a distributed object.
  - One such service is the Java Naming and Directory Interface (JNDI), which is more general than the RMI registry, in the sense that it can be used by applications that do not use the RMI API.

- We will use a simple directory service called the RMI registry, `rmiregistry`, which is provided with the Java Software Development Kit (SDK).
  - The RMI Registry is a service whose server, when active, runs on the object server's host machine, by convention and by default on the TCP port 1099.

RMI applications

- Often comprise two separate programs
  - a server and a client.
- A typical server program
  - creates some remote objects,
  - makes references to these objects accessible, and
  - waits for clients to invoke methods on these objects.
- A typical client program
  - obtains a remote reference to one or more remote objects on a server; and
  - invokes methods on them.
- RMI provides the mechanism by which the server and the client communicate and pass information back and forth.
- Such an application is sometimes referred to as a
  - *distributed object application*.

Creating Distributed Applications by Using RMI

Steps:

- Designing and implementing the components of your distributed application
- Compiling sources
- Making classes network accessible
- Starting the application

Designing and Implementing the Application Components

- Defining the remote interfaces.
  - A remote interface specifies the methods that can be invoked remotely by a client.
  - Clients program to remote interfaces, not to the implementation classes of those interfaces.

- Implementing the remote objects.
  - Remote objects must implement one or more remote interfaces.
  - The remote object class may include implementations of other interfaces and methods that are available only locally.

- Implementing the clients.
  - Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

The Remote Interface

- A Java interface is a class that serves as a template for other classes
  - it contains declarations or signatures of methods whose implementations are to be supplied by classes that implements the interface.

- A java remote interface is an interface that inherits from the Java `Remote` class, which allows the interface to be implemented using RMI syntax.
  - Other than the `Remote` extension and the `Remote` exception that must be specified with each method signature, a remote interface has the same syntax as a regular or local Java interface.

- An object becomes remote by implementing a remote interface, which has the following characteristics:
  - A remote interface extends the interface `java.rmi.Remote.`
  - Each method of the interface declares `java.rmi.RemoteException` in its throws clause, in addition to any application-specific exceptions.

The Server-side Software

- An object server is an object that provides the methods of and the interface to a distributed object. Each object server must
  - implement each of the remote methods specified in the interface,
  - register an object which contains the implementation with a directory service.
- It is recommended that the two parts be provided as separate classes.

The Client-side Software

- The program for the client class is like any other Java class.

- The syntax needed for RMI involves
  - locating the RMI Registry in the server host, and
  - looking up the remote reference for the server object; the reference can then be cast to the remote interface class and the remote methods invoked.

Example application - Hello

- Displays a greeting to any client that uses the appropriate interface registered with the naming service to invoke the associated method implementation on the server.

Create the interface

- Should import package `java.rmi`
- Must extend interface `Remote` – an interface that contains no methods.
- `Hello` example:
  - It declares just one method, sayHello, which returns a string to the caller:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Create the server process

- A "server" class, is the class which has a main method that creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name in a Java RMI registry.
- The class that contains this main method could be the implementation class itself, or another class entirely.
- In this example, the main method for the server is defined in the class Server which also implements the remote interface Hello.
- The server's main method does the following
  - Create and export a remote object
  - Register the remote object with a Java RMI registry

Create and export a remote object

- The main method of the server needs to create the remote object that provides the service.

- Additionally, the remote object must be exported to the Java RMI runtime so that it may receive incoming remote calls.

```
Server obj = new Server();
Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
```

Register the remote object with a Java RMI registry

- For a caller (client, peer) to be able to invoke a method on a remote object, that caller must first obtain a stub for the remote object.
- JAVA RMI provides a registry API for applications to bind a name to a remote object's stub and for clients to look up remote objects by name in order to obtain their stubs.
- Once a remote object is registered on the server, callers can look up the object by name, obtain a remote object reference, and then invoke remote methods on the object.

```
Registry registry = LocateRegistry.getRegistry();
registry.bind("Hello", stub);
```

- The static method LocateRegistry.getRegistry that takes no arguments returns a stub that implements the remote interface `java.rmi.registry.Registry` and sends invocations to the registry on server's local host on the default registry port of 1099.
- The bind method is then invoked on the registry stub in order to bind the remote object's stub to the name "Hello" in the registry.

Server code

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

public class Server implements Hello {

    public Server() {}

    public String sayHello() {
        return "Hello, world!";
    }

    public static void main(String args[]) {

        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);

            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

The client process

- The client program obtains a stub for the registry on the server's host, looks up the remote object's stub by name in the registry, and then invokes the `sayHello` method on the remote object using the stub.

- This client first obtains the stub for the registry by invoking the static `LocateRegistry.getRegistry` method with the hostname specified on the command line. If no hostname is specified, then null is used as the hostname indicating that the local host address should be used.

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

- Next, the client invokes the remote method lookup on the registry stub to obtain the stub for the remote object from the server's registry.

The client process

- Next, the client invokes the remote method lookup on the registry stub to obtain the stub for the remote object from the server's registry.

```
Hello stub = (Hello) registry.lookup("Hello");
```

The client process

- Finally, the client invokes the `sayHello` method on the remote object's stub, which causes the following actions to happen:

- The client-side runtime opens a connection to the server using the host and port information in the remote object's stub and then serializes the call data.
- The server-side runtime accepts the incoming call, dispatches the call to the remote object, and serializes the result (the reply string "Hello, world!") to the client.

- The client-side runtime receives, deserializes, and returns the result to the caller.

- The response message returned from the remote invocation on the remote object is then printed to `System.out`.

Client code

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

public class Client {

    private Client() {}

    public static void main(String[] args) {

        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Steps to Run

1. Compile all the three java files.

2. Open a Command Prompt where you are going to run Server.
   - Start Registry with the following command:

```
>start rmiregistry
```

3. Start Server and Client in two separate Command Prompts

```
java Server
java Client
```

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

5. The output of Client will look like:

```
response: Hello, world!
```

RMI vs. Sockets

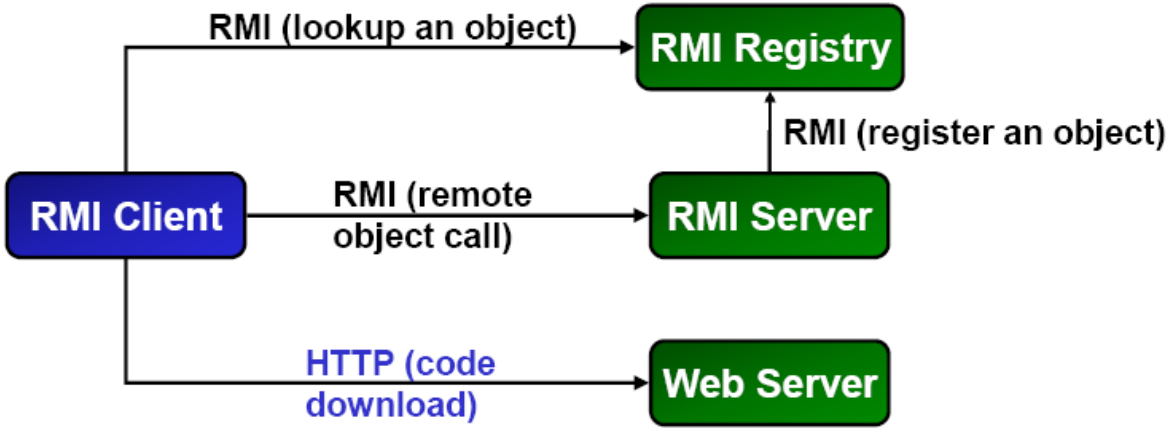
- RMI API can be used instead of the socket API in a network application.
- There are some tradeoffs:
  - The socket API is closely related to the operating system, and hence has less execution overhead.
- For applications which require high performance, this may be a consideration.
- The RMI API provides the abstraction which eases the task of software development.
  - Programs developed with a higher level of abstraction are more comprehensible and hence easier to debug.

RMI Dynamic class loading

- Facilitates updates of remote objects once the application has been deployed.
- If the server implementation is changed and a new stub class is generated the stub class will have to be distributed to all clients.
  - This is undesirable.
- A better approach would be to make the new stub class available online and whenever a client starts up it automatically loads the new class from the web server.

RMI Dynamic Class Loading

- Stub files can be downloaded on request by the client instead of distributing them manually to each client.
- The code (stub class files) can be made available for download on a web server.
- The interface file (class file containing the compiled remote interface) is still required both on the client and server side (otherwise client and server will not compile).



RMI Dynamic Class Loading

- Both client and server need to install a security manager to be able to load classes remotely:
- Client and server need a `security policy` file granting the necessary rights like opening network connections (the following security policy file simply grants everything = no security at all):

`mysecurity.policy` file:

```
grant {
permission java.security.AllPermission;
}
```

- Starting the server with the security policy file, e.g.

```
java –Djava.rmi.server.codebase="http://myserver/example/"
–Djava.security.policy=mysecurity.policy MyServer
```

- Starting the client with the security policy file:

```
java –Djava.security.policy=mysecurity.policy MyClient
```

RMI Callbacks

- In the client server model, the server is passive: the IPC is initiated by the client; the server waits for the arrival of requests and provides responses.
- Some applications require the server to initiate communication upon certain events. Examples applications are:
  - monitoring
  - games
  - auctioning
  - voting/polling
  - chat-room
  - message/bulletin board
  - groupware

RMI Callbacks

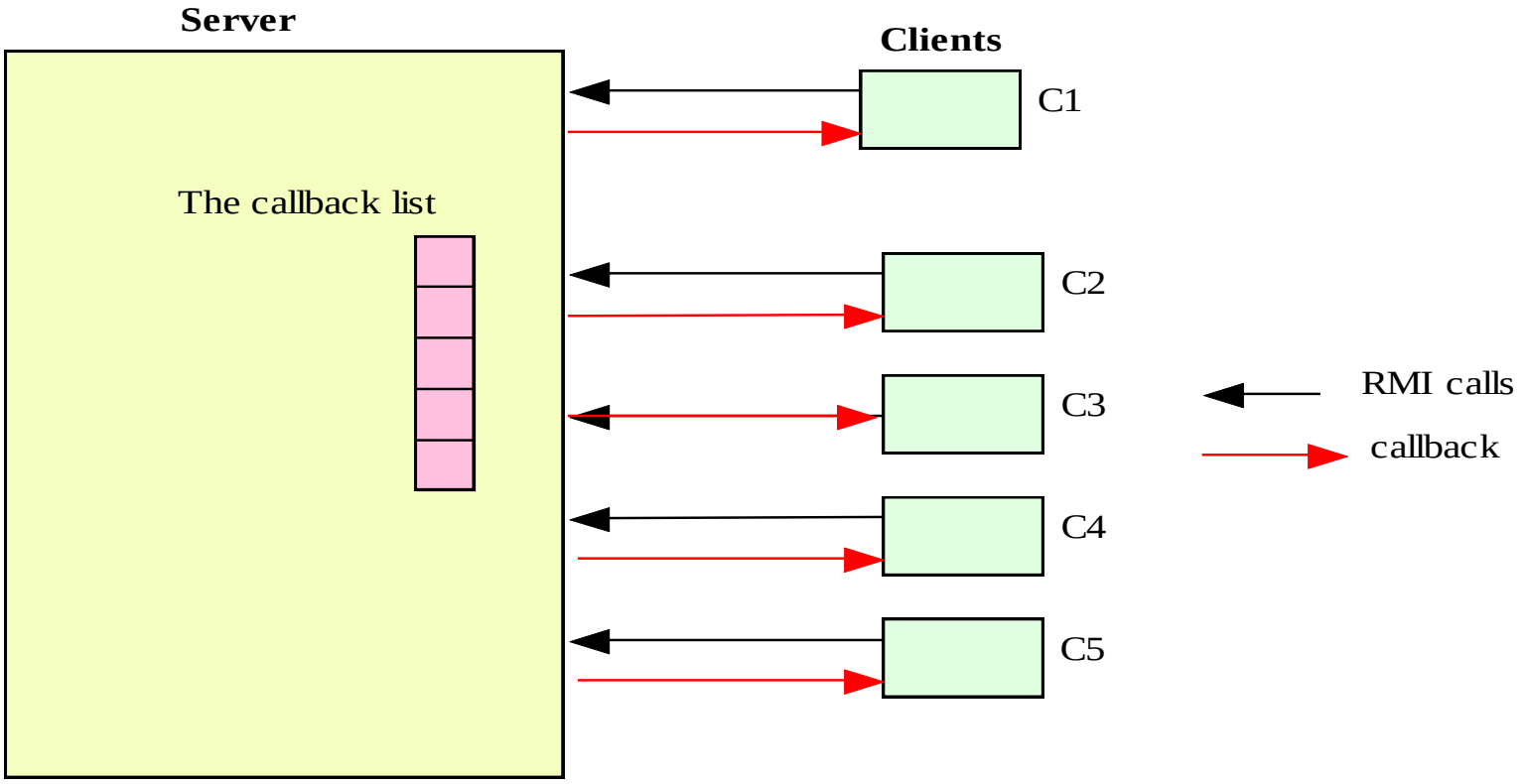
- Like any other callback
- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.

Callbacks

- Pass a reference to a remote object to another environment. A call to the stub of that object still call the original object i.e. a *call back*
- Call from server back to client.
- Remote reference to client must be available on server.
- Client object must be made remote.

RMI Callbacks

- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.



Testing and Debugging an RMI Application

1. Build a template for a minimal RMI program. Start with a remote interface with a single signature, its implementation using a stub, a server program which exports the object, and a client program which invokes the remote method. Test the template programs on one host until the remote method can be made successfully.

2. Add one signature at a time to the interface. With each addition, modify the client program to invoke the added method.

3. Fill in the definition of each remote method, one at a time. Test and thoroughly debug each newly added method before proceeding with the next one.

4. After all remote methods have been thoroughly tested, develop the client application using an incremental approach. With each increment, test and debug the programs.

References

- https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html

- https://docs.oracle.com/javase/tutorial/rmi/
- Chapter 5: Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 5ed, 2012