1. Introduction of RPC / Java RMI

RPC is a common technique for constructing distributed, client-server based application. It enables the intercommunication between two processes - client process and server process, which might be located on a single host or on different hosts. The client process calls the remote procedures which is shared by the server process, and get the return result.

RMI (Remote Method Invocation) is a Java implementation of RPC, which includes a core Java API and class library. It allows Java objects on different hosts to communicate with each other. The server implements a remote interface that specifies which of its methods can be invoked by clients, so that the clients invoke the remote methods as normal local methods.

Java/RMI Setting:
RMI is included in the core Java API and class library. To use RMI, JDK (Java Development Kit) should be installed and Java environment has to be setup.

- Download and install the JDK 1.4.2 from http://java.sun.com
- Setup path to $JDK_HOME/bin, where $JDK_HOME is the home directory where JDK is installed. In case $JDK_HOME=/usr/java
  
  % setenv PATH=/usr/java/bin:$PATH or,
  % export PATH /usr/java/bin:$PATH

- Setup environmental argument CLASSPATH for JDK as .:$JDK_HOME/lib/rt.jar
  
  % setenv CLASSPATH=.:/usr/java/lib/rt.jar or,
% export CLASSPATH ./usr/java/lib/rt.jar

- Find the executable files java, javac, rmiregistry, rmic under the directory $JDK_HOME/bin.

RMI Example:

Here is a simple example to show how to create a RMI server and client.
A RMI server exposes one remote method int addOne(int i) to the clients. With the input arguments i from clients, addOne() simply returns i+1.

Server Side:
First, an interface class declaring the method is created

```java
package example;
import java.rmi.*;
public interface Calculator extends Remote{
    public int addOne(int i) throws RemoteException;
}
```

All the remote methods are declared to throw RemoteException when failure occurs during the remote invocation. Catching and handling the RemoteException is up to the clients which use the remote methods.

The class CalculatorImpl implements the interface:

```java
package example;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculatorImpl implements Calculator{
    public CalculatorImpl() throws RemoteException {
        UnicastRemoteObject.exportObject(this);
    }
    public int addOne(int i) throws RemoteException {
        return i+1;
    }
}
```

In the CalculatorImpl() constructor, UnicastRemoteObject.exportObject(this) exports the object by listening for connections on some port, so that the invocation from clients can be processed.

Now, the way to create a server CalculatorServer to handle the client’s request is as follows:
List 3. Class CalculatorServer

Naming.rebind("calculator", c) is to bind the calculator object with the name "calculator". The binding name is case-sensitive. Together with the IP address or URL name of the host where the server is running on, a URL address “rmi://server_ip/calculator” is used to address the calculator object.

Clients can use the URL “rmi://server_ip/calculator” to get the reference to the remote object and invoke the method. Here, server_ip is the IP address or URL name of the host where the server is running.

Client Side:
Before a client can call the remote method addOne(int), it needs to retrieve the remote reference to the calculator object. RMI provides a method lookup() in the class java.rmi.Naming for clients to get the reference to the remote object:

Calculator cal = (Calculator) Naming.lookup("rmi://server_ip/calculator");

The example of the client is shown below:
try {
    Calculator cal = (Calculator) Naming.lookup(args[0]);
    int input = (new Integer(args[1])).intValue();
    int output = cal.addOne(input);
    System.out.println("The output of addOne(" + input + ")" + " is " + output);
    } catch (MalformedURLException e) {
        System.err.println(args[0] + " is not a valid RMI URL");
    } catch (RemoteException e) {
        System.err.println("Remote object threw exception " + e);
    } catch (NotBoundException e) {
        System.err.println("Cannot find the requested remote object on the server");
    }
}
The server `MessagePoolServer` implements two remote methods defined in the interface class `MessagePool`:

```java
import java.rmi.*;

public interface MessagePool extends Remote {
    public void put(String msg) throws RemoteException;
    public String get() throws RemoteException
}
```

### List 5. Interface MessagePool

- `put()` method accepts a String message from the client, and stores it into the FIFO queue in the server. In case the queue is full, `put()` operation will fail, and a `QueueFullException` will be thrown. In case the message from the client is null, the server will throw a `MessageNullException`.

- `get()` method retrieves the message out of the queue to the client which invoke it. The retrieved message will be deleted from the queue. In case the queue is empty, `get()` operation will fail and a `QueueEmptyException` will be thrown.

- The FIFO message queue `MessageQueue` should have a size of 100 messages. Messages are strings with at most 500 characters.

- Implement two clients: `MessagePutClient` generates messages periodically (1 message per 1 second), and `MessageGetClient` retrieve messages periodically (1 message per 2 second). The message could be the timestamp of the client or any random generated string.

- Make sure all cases are handled:
  - Retrieving from an empty queue / Adding message to a full queue
  - Trying to add a message to a full queue

- For this exercise the program should be single threaded.