**University of Rajshahi**

**Faculty of Science**

## Department of Computer Science and Engineering

**Lab Manual for the course CSE 415P**

**Course Title:** *Parallel Processing and Distributed System Lab*

**Practical List**

# Problem 1: Displaying “Hello” using RMI- a sample program for RMI implementation.

**Problem 2:** Sum of numbers - a simple client/server application.

**Problem 3:** Client console command execution in server console - simple part of FTP application.

**Problem 4:** File transfer between a local and a remote host- advanced part of FTP application.

**Problem 5:** Sum of sequence using multiple RMI call - implementation of data parallelism.

**Problem 6:** Sum and multiplication of numbers using multiple RMI call - implementation of functional parallelism.

**Problem 7:** Task Bag using Java RMI.

**Problem 8:** Multi-level RMI calls in the same execution program.

**Problem 9:** Vector Task System - implementation of pipelining.

**Problem 10:** Collision Free Scheduling - implementation of Scheduling.

|  |
| --- |
| **Problem 1** |
| **Title:** Displaying **“**Hello**”** using RMI- a **s**ample program for RMI implementation |
| **RMI (Remote Method Invocation)**  RMI serves as a basic technique for supporting distributed objects in java. RMI allows a client program to call a server function as passing predefined objects to it and to receive a return object. This example describes the way to display Hello message using RMI. The steps involved in displaying message Hello are described below:-  **Step 1.** Define a server interface that inherits the remote class. This interface simply defines a prototype of all RMI functions that will be made available at your server. Create a remote interface named ***HelloInterface.java*** in the directory.  **HelloInterface.java:-**   |  | | --- | | **import**java.rmi.\*;  **public interface**HelloInterface **extends**Remote {   **public**String say() **throws**RemoteException; } |   **Step 2.** Create a remote class implementation for HelloWorld named ***HelloImplementation.java*** in the directory. An RMI server implementation program must satisfy the following two requirements (a) Extend UnicastRemoteObject and implements ServerInterface and (b) Define a constructor that throws RemoteException.  **HelloImplementation.java:-**   |  | | --- | | **import**java.rmi.\*; **import**java.rmi.server.\*;  **public class**Hello **extends**UnicastRemoteObject  **implements**HelloInterface {   **private**String message;   **public**Hello (String msg) **throws**RemoteException {   message = msg;   }   **public**String say() **throws**RemoteException {   **return**message;   } } |   **Step 3.** Compile the above two source file named ***HelloInterface.java*** and ***Hello.java****.*  **Step 4.** After compiling the above two classes type the following command "rmic Hello" in console. By running the "rmic Hello" command a new class will be created i.e "Hello\_Stub.class" in the directory  http://www.roseindia.net/tutorials/rmi/Screenshot-1.gif  **Step 5.** Create server application named ***HelloServer.java*** in the Directory. RMI server program must satisfy the following two requirements: (a) Include the main() function that instantiates the server itself and binds this instance to “rmi://localhost:port/sympoblic\_name”. and (b) Implements all RMI methods defined in the server interface.  **HelloServer.java:-**   |  | | --- | | **import**java.rmi.Naming;  **public class**HelloServer  {   **public static void**main (String[] argv)    {   **try**{   Naming.rebind     ("Hello", **new**Hello ("Hello,From Roseindia.net pvt ltd!"));   System.out.println    ("Server is connected and ready for operation.");   }    **catch**(Exception e) {   System.out.println ("Server not connected: " + e);   }   } } |   **Step 6.** Create client application named HelloClient.java. An RMI client program must follow the following instructions:   1. Import java.rmi.\*;   import java.rmi.\*;   1. Look for the server instance it wants to access.   ServerInterface server = ( ServerInterface )  Naming.lookup( “rmi://serverIp:serverPort/symbolic\_name” );   1. Catch exception when calling an RMI function.   **HelloClient.java:-**   |  | | --- | | **import**java.rmi.Naming;  **public class**HelloClient  {   **public static void**main (String[] argv) {   **try**{ HelloInterface hello =(HelloInterface)  Naming.lookup ("//192.168.10.201/Hello");   System.out.println (hello.say());   }    **catch**(Exception e){   System.out.println ("HelloClient exception: " + e);}   } } |   **Step 7.** Compile both (server and client program) of the files.  **Step 8.** Type "rmicregistry" on command prompt and press ENTER.  **http://www.roseindia.net/tutorials/rmi/Screenshot-4.gifStep 9.** Type java ***HelloServer*** in command prompt and press ENTER. The following message will be displayed on console.  **http://www.roseindia.net/tutorials/rmi/Screenshot-5.gifStep 10.** Now, open another separate command terminal, and run the client application like shown in the figure given below:-  **Step 11.** If the message similar to the above appears in figure comes means that you have implemented your RMI application. |
| **Problem 2** |
| **Title:** Sum of numbers - a simple client/server application. |
| **Outline:** The goal of this problem is to design and implement a very simple client/server application. Through this problem, you will understand that (1) the design of both client and server and (2) client needs to call server’s RMI functions. |
| **Task:** Using RMI read two integers from local machine console and calculates the sum in remote machine. Finally, print the result (including two integers) in local as well as remote machine.  Your RMI system must be composed of the following parts:   * An interface definition of the remote services that are provided; * The implementations of the remote services; * Stub and skeleton files; * A server to host the remote services; * An RMI Naming service that allows clients to find the remote services; * A client program that uses the remote services.   **Sample Client/Server Output:**  The first number is: 8  The second number is: 9  The sum is: 17.0 |
| **Mode of evaluation:** Experiments + Viva |
| **Percentage of weight:** |
| **Problem 3** |
| **Title:** Client console command execution in server console - simple part of FTP application. |
| **Outline:** The **ftp** uses the File Transfer Protocol (FTP) to transfer files between the local host and a remote host or between two remote hosts. At the ftp> prompt, you can enter subcommands to perform tasks such as listing remote directories, changing the current local and remote directory, transferring multiple files in a single request, creating and removing directories, and escaping to the local shell to perform shell commands. |
| **Task:** Read some simple ftp console command (e.g. cd, dir, mkdir, rename, copy etc.) from local machine execute the corresponding command in remote server machine. You will write interface, implementation, client and server programs, execute all of them, and perform console command execution. |
| **Mode of evaluation:** Experiments + Viva |
| **Percentage of weight:** |
| **Problem 4** |
| **Title:** File transfer between a local and a remote host- advanced part of FTP application using RMI |
| **Outline:** Developing distributed applications in RMI is simpler than developing with sockets since there is no need to design a protocol, which is an error-prone task. In RMI, the developer has the illusion of calling a local method from a local class file, when in fact the arguments are shipped to the remote target and interpreted, and the results are sent back to the callers. A very simple file transferring system in that a client retrieves a file from a central server and caches it in its directory for better performance. A user types the name of a file he/she wants to access as well as the file access mode (such as read or write). Through this project, you will understand that (1) the design of both client and server is based on FTP application; (2) the client needs to provide file name and access mode; and (3) client need to call server’s RMI functions. |
| **Task:** Suppose remote server machine contains plain text file. Send a request from local machine to read/modify the contents of that file from local machine and process the request in remote machine and return the contents into local machine. The first step is to define a remote interface that specifies the signatures of the methods to be provided by the server and invoked by clients.  Once invoked, the client program prints out the following message to start a file session.  **FileClient:** FTP application  **File name:**  **How(r/w):** |
| **Mode of evaluation:** Experiments + Viva |
| **Percentage of weight:** |
| **Problem 5** |
| **Title:** Sum of sequence using multiple RMI call - implementation of data parallelism. |
| **Outline:** A data-parallel model focuses on performing operations on a data set which is usually regularly structured in an array. A set of tasks will operate on this data, but independently on separate partitions. In a shared memory system, the data will be accessible to all, but in a distributed-memory system it will divide between memories and worked on locally. Data parallelism is usually classified as SIMD. |
| **Task:** RMI client will invoke method on other two RMI servers acting as clients (Server1 and Server 2) in the same execution program. Your program will call multiple servers from a single client to calculate the sum of numbers between two given numbers. The first server should return the sum of a portion of that range and the second server should return the sum of the rest of the range. Finally, the client should add the two sums and show the result. |
| **Mode of evaluation:** Experiments + Viva |
| **Percentage of weight:** |
| **Problem 6** |
| **Title:** Sum and multiplication of numbers using multiple RMI call - implementation of function parallelism. |
| **Outline:** Function parallelism (also known as Task parallelism and control parallelism) is a form of [parallelization](http://en.wikipedia.org/wiki/Parallelization) of computer code across multiple [processors](http://en.wikipedia.org/wiki/Central_processing_unit) in [parallel computing](http://en.wikipedia.org/wiki/Parallel_computing) environments. Task parallelism focuses on distributing execution processes (threads) across different parallel computing nodes. In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work. Communication usually takes place by passing data from one thread to the next. |
| **Task:** You will create a simple RMI application that uses one client and two different servers. The concurrent execution of two RMI call on those servers. When the client calls both servers, then the each server will response according to the RMI policy. Suppose you are given a set of n numbers. Call multiple servers from a single client to add and multiply the number sequence. One server will add the numbers and another will do the job of multiplying. |
| Mode of evaluation: Experiments + Viva |
| Percentage of weight: |
| **Problem 7** |
| **Title:** Task Bag using Java RMI |
| **Outline:** The objective of the program is to build a simple generic toolkit that enables processes running in different computers to carry out parts of a large computation in parallel. The general idea is that a *master process* places sub-tasks of the computation in a ***Task Bag*** and *worker processes* select and take tasks from the ***Task Bag*** and carry them out, returning the results to the Task Bag. The master then collects the results and combines them to produce the final result.  **How the workers know which task to perform next**  In many computations, there is a collection of tasks, numbered from *First* to *Last*. Each worker repeatedly carries out one (or a group) of tasks. Before a worker starts, it needs to know which task to do next. A *Pair* with the key “*Next Task”* can be used for this purpose.  The master process places the first task:  *placePair*(*"Next Task", First*)*;*  and each worker, in turn, takes the *Pair* out, increments its value and puts it back. e.g.  *nextElement = takePair*(*"Next Task"*)*;*  *placePair*(*"Next Task", nextElement + GRANULARITY*);  The number of tasks done together is a per application constant (GRANULARITY). When there are no more tasks to be done, the worker does not replace Next Task in the Task Bag.  When other workers attempt to remove it, they will have to wait. No more work will be done until the master supplies another collection of tasks to calculate. |
| **Task:** In this program you are asked to implement the *Task Bag* as a Remote Object and to use it as a basis for performing a parallel computation of a *big problem* on several computers. You can choose any big problem that can be performed in parallel to demonstrate the results. However, your implementation must be generic and it must be easy to adapt it to any type parallel computation.  Three entities are involved:  1. The Task Bag object,  2. The master process  3. The worker processes.  The master and worker processes are clients of the Task Bag object.  You should choose one application that requires fairly intensive computation to carry it out and which is easily divided into a number of identical subtasks. You could consider tasks such as: (i) number sorting (ii) finding prime numbers (iii) matrix multiplication (iv) Merge Sort or (v) Insertion sort. |
| **Mode of evaluation:** Experiments + Viva |
| **Percentage of weight:** |
| **Problem 8** |
| **Title:** Multi-level RMI calls in the same execution program. |
| **Outline:** RMI client will invoke method on RMI main Server and that RMI main server will invoke method on other two RMI child server acting as clients (child server1 and server 2) in the same execution program. |
| **Task:** You will create a RMI application that uses one client, main server, and two different servers. Call a main server from a single client to calculate the sum of numbers between two given numbers. The main server should calculate the sum by distributing the job to two child server. |
| **Mode of evaluation:** Experiments + Viva |
| **Percentage of weight:** |
| **Problem 9** |
| **Title:** Vector Task System (Pipelining) |
| **Outline:** Task systems obtain as input a sequence of requests such that each request assigns processing times to the states. The objective of a task system is to create a schedule that minimizes the overall cost incurred due to processing the tasks. |
| **Task: Implement a** vector task system V with T = {T1, T2, T3, T4}, t0 = 1, τ1 = 10, τ2 = 2, τ3 = 6, τ4 = 2. Schedule these tasks on two pipelines (m=2) so that finish time (ω) is minimized.  **Inputs** will consists of a number of tasks (n), production delay of each tasks (τi) and pipeline overhead due to startup and flushing (t0).  **Output** will display the schedule for the task system.  **Example:**   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | 0 | | 1 | | 2 | | 3 | | | | 4 | | | 5 | 6 | 7 | | 8 | | 9 | | 10 | 11 | | 12 | | | 13 | |  | | P1 | | t0 | | τ11 | | | | | | | | | | | | | | | t0 | | Τ31 | | | | | | | |  | | | P2 | | t0 | | Τ4 | | | | | t0 | | | τ12 | | | | |  | | t0 | | Τ2 | | | t0 | | | Τ32 | | | | |  | 0 | | 1 | | | 2 | | 3 | | 4 | | | 5 | | 6 | 7 | | 8 | | 9 | | 10 | 11 | | | 12 | | 13 | | 14 | | |
| **Mode of evaluation:** Experiments + Viva |
| **Percentage of weight:** |
| **Problem 10** |
| Title: Collision Free Scheduling (Scheduling) |
| **Outline:** When scheduling events in a pipeline, the main objective is to obtain the shortest average latency between initiations without causing collisions. |
| **Task:** A collision vector C = (Cm Cm-1 …C2 C1) (containing permissible and forbidden latencies) is given. Your task is to obtain a state diagram.  Inputs consist of a collision vector (a sequence of 1’s and 0’s; 1 for collision and 0 for permissible)  Output will be a three columns Table showing present state, latency, and next state.  **Example:** For input 1011010 the output will be   |  |  |  | | --- | --- | --- | | **Present State** | **Latency** | **Next State** | | 1011010 | 8 | 1011010 | | 1011010 | 3, 6 | 1011011 | | 1011010 | 1 | 1111111 | | 1011011 | 3, 6 | 1011011 | | 1011011 | 8 | 1011010 | | 1111111 | 8 | 1011010 | |
| **Mode of evaluation:** Experiments + Viva |
| **Percentage of weight:** |