**Department of Computer Engineering**

**Academic Term: January - May 2021**

**Class: *B.E (Computer) Semester VIII***

**Subject Name: *Distributed Computing***

**Subject Code: *CSC802***

|  |  |
| --- | --- |
| **Practical No:** | **1** |
| **Title:** | **Remote Procedure Call** |
| **Date of Performance:** |  |
| **Date of Submission:** |  |
| **Roll No:** | **8364** |
| **Name of the Student:** | **Vedant Sahai** |

**Evaluation:**

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| **1** | **On time Completion & Submission** |  |
| **2** | **Coding Standards** |  |
| **3** | **Output/Test Cases Used** |  |
| **4** | **Conclusion & Post Lab Assignment** |  |

**Signature of the Teacher :**

**Code:**

**rpcsqaure. x**

program PROGRAM

{

version VERSION

{

int func(int)=1;

} =1;

} =0x200002;

**server.c**

#include <stdio.h>

#include <rpc/rpc.h>

#include "rpcsquare.h"

int \*func\_1\_svc (int \*a, struct svc\_req \*x)

{

    int \*b = malloc(sizeof(int));

    \*b = (\*a) \* (\*a);

    return (b);

}

**client.c**

#include <stdio.h>

#include <rpc/rpc.h>

#include "rpcsquare.h"

int main ()

{

    CLIENT \*c1 = clnt\_create ("127.0.0.1", PROGRAM, VERSION, "udp");

    int n;

    printf("enter no:\t");

    scanf("%d", &n);

    int\* result = func\_1(&n, c1);

    printf("the result is : %d\n", \*result);

}

**start.sh**

sudo rpcbind start || true

rpcgen rpcsquare.x

gcc server.c rpcsquare\_svc.c -o server.out

./server.out &

P1=$!

gcc client.c rpcsquare\_clnt.c -o client.out

./client.out

kill -9 $P1

**Output:**

enter no: 5

the result is: 25

**Conclusion:**

Remote Procedure Call (RPC) is an interprocess communication technique. It is used for client-server applications. RPC mechanisms are used when a computer program causes a procedure or subroutine to execute in a different address space, which is coded as a normal procedure call without the programmer specifically coding the details for the remote interaction. This procedure call also manages low-level transport protocol, such as User Datagram Protocol, Transmission Control Protocol/Internet Protocol etc. It is used for carrying the message data between programs. The Full form of RPC is Remote Procedure Call.

**Post Lab Assignments**

1. What is an Interface file? (.x file)

Ans: -

The interfaces file is a dictionary of connection information for Adaptive Servers and Open Server applications. For every server to which a client might connect, the interfaces file contains an entry that includes the server’s name and the necessary information to connect to that server. The interfaces file is the default directory for Client-Library. However, applications may be configured to use a Sybase directory driver so that Client-Library uses a network-based directory service provider. For information on configuring Sybase directory drivers, see the Open Client and Open Server Configuration Guide. For information on network-based directory services, see [“Directory services”](http://infocenter.sybase.com/help/topic/com.sybase.help.sdk_12.5.1.ctref/html/ctref/X84280.htm).On most platforms, the interfaces file is an operating system file in text format. On these systems, the default name, default location, and internal format of the interfaces file differ by platform. Other platforms use an alternate form of storage.

1. What are the functions of stubs?

Ans: -

A stub in [distributed computing](https://en.wikipedia.org/wiki/Distributed_computing) is a piece of code that converts parameters passed between client and server during a [remote procedure call](https://en.wikipedia.org/wiki/Remote_Procedure_Call) ([RPC](https://en.wikipedia.org/wiki/Remote_procedure_call)).

The main idea of an [RPC](https://en.wikipedia.org/wiki/Remote_procedure_call) is to allow a local computer ([client](https://en.wikipedia.org/wiki/Client_(computing))) to remotely call procedures on a different computer ([server](https://en.wikipedia.org/wiki/Server_(computing))). The client and server use different [address spaces](https://en.wikipedia.org/wiki/Address_space), so parameters used in a function (procedure) call have to be converted, otherwise, the values of those parameters could not be used, because pointers to parameters in one computer's memory would point to different data on the other computer. The client and server may also use different data representations, even for simple parameters (e.g., [big-endian](https://en.wikipedia.org/wiki/Big-endian) versus [little-endian](https://en.wikipedia.org/wiki/Little-endian) for integers). Stubs perform the conversion of the parameters, so a remote procedure call looks like a local function call for the remote computer.

Stub libraries must be installed on both the client and server-side. A client stub is responsible for the conversion [(marshalling)](https://en.wikipedia.org/wiki/Marshalling_(computer_science)) of parameters used in a function call and deconversion of results passed from the server after execution of the function. A server [skeleton](https://en.wikipedia.org/wiki/Class_skeleton), the stub on the server-side, is responsible for deconversion of parameters passed by the client and conversion of the results after the execution of the function.

1. How the stubs are generated in RPC?

Ans: -

Manually: In this method, the RPC implementer provides a set of translation functions from which a user can construct his or her stubs. This method is simple to implement and can handle very complex parameter types. Automatically: This is the more commonly used method for stub generation. It uses an [interface description language](https://en.wikipedia.org/wiki/Interface_description_language) (IDL) to define the interface between client and server. For example, an interface definition has information to indicate whether each argument is input, output or both; only input arguments need to be copied from client to server and only output elements need to be copied from server to client.

1. What do you mean by binding?

Ans: -

## Binding is the process of connecting the client and server

## The server, when it starts up, exports its interface, identifying itself to a network name server and telling the local runtime its dispatcher address.

## The client, before issuing any calls, imports the server, which causes the RPC runtime to look up the server through the name service and contact the requested server to set up a connection.

## The import and export are explicit calls in the code.

**Department of Computer Engineering**

**Academic Term: January - May 2021**

**Class: *B.E (Computer) Semester VIII***

**Subject Name: *Distributed Computing***

**Subject Code: *CSC802***

|  |  |
| --- | --- |
| **Practical No:** | **2** |
| **Title:** | **Remote Method Invocation** |
| **Date of Performance:** |  |
| **Date of Submission:** |  |
| **Roll No:** | **8364** |
| **Name of the Student:** | **Vedant Sahai** |

**Evaluation:**

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| **1** | **On time Completion & Submission** |  |
| **2** | **Coding Standards** |  |
| **3** | **Output/Test Cases Used** |  |
| **4** | **Conclusion & Post Lab Assignment** |  |

**Signature of the Teacher :**

**Code:**

**Hello.java**

import java.rmi.Remote;

import java.rmi.RemoteException;

// Creating Remote interface for our application

public interface Hello extends Remote {

   void printMsg(String msg) throws RemoteException;

}

**ImplExample.java**

public class ImplExample implements Hello {

   // Implementing the interface method

   public void printMsg(String msg) {

      System.out.println(msg);

   }

}

**Server.java**

import java.rmi.registry.Registry;

import java.rmi.registry.LocateRegistry;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class Server extends ImplExample {

   public Server() {}

   public static void main(String args[]) {

      try {

         // Instantiating the implementation class

         ImplExample obj = new ImplExample();

         // Exporting the object of implementation class

         // (here we are exporting the remote object to the stub)

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

                  // Binding the remote object (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();

      }

   }

}

**Client.java**

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import java.util.Scanner;

public class Client {

   private Client() {}

   public static void main(String[] args) {

      try {

         // Getting the registry

         Registry registry = LocateRegistry.getRegistry(null);

         // Looking up the registry for the remote object

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

         Scanner scan  = new Scanner(System.in);

         while(true){

            String msg = scan.next();

            if (msg == "q"){

               break;

            }

            stub.printMsg(msg);

         }

               // System.out.println("Remote method invoked");

      } catch (Exception e) {

         System.err.println("Client exception: " + e.toString());

         e.printStackTrace();

      }

   }

}

**Output:**

PS D:\Desktop\BE\experiment\Batch-C-Codes\dc\exp2> java Client

hi

i am vedant

PS D:\Desktop\BE\experiment\Batch-C-Codes\dc\exp2> java Server

Server ready

hi

i

am

vedant

**Conclusion:**

RMI provides a solid platform for truly object-oriented distributed computing. You can use RMI to connect to Java components or existing components written in other languages. As Java proves itself in your environment, you can expand your Java use and get all the benefits-no of porting low maintenance costs, and a safe, secure environment. RMI gives you a platform to expand Java into any part of your system in an incremental fashion, adding new Java servers and clients when it makes sense. As you add Java, its full benefits flow through all the Java in your system. RMI makes this easy, secure, and powerful.

**Post Lab Assignments**

1. Is there any difference between Distributed objects and remote objects?  Can an Object be distributed on multiple machines?

Ans: -

Remote and distributed objects differ in many respects.[[3]](https://en.wikipedia.org/wiki/Distributed_object#cite_note-3)[[4]](https://en.wikipedia.org/wiki/Distributed_object#cite_note-4) Here are some of them:

* Life cycle: Creation, migration and deletion of distributed objects is different from Remote objects
* Reference: Remote references to distributed objects are more complex than simple pointers to memory addresses
* Request Latency: A distributed object request is orders of magnitude slower than Remote method invocation
* Object Activation: Distributed objects may not always be available to serve an object request at any point in time
* Parallelism: Distributed objects may be executed in parallel.
* Communication: There are different communication primitives available for distributed objects requests
* Failure: Distributed objects have far more points of failure than typical Remote objects.
* Security: Distribution makes them vulnerable to attack.

In a general distributed object, the state itself may be physically distributed across multiple machines, but this distribution is also hidden from clients behind the object's interfaces. Objects in distributed systems appear in many forms. The most obvious form is the one that is directly related to language-level objects such as those supported by Java, C++, or other object-oriented languages, which are referred to as compile-time objects.  .In this case, an object is defined as the instance of a class.

1. What do the commands rmic and rmiregistry do?

Ans: -

Rmic-

The rmic compiler generates stub and skeleton class files (JRMP protocol) and stub and ties class files (IIOP protocol) for remote objects. These classes files are generated from compiled Java programming language classes that are remote object implementation classes. A remote implementation class is a class that implements the interface *java.rmi.Remote*. The class names in the *rmic* command must be for classes that have been compiled successfully with the javac command and must be fully package qualified. For example, running rmic on the class file name *HelloImpl* as shown here:

*rmic hello.HelloImpl*

creates *the HelloImpl\_Stub.class* file in the *hello* subdirectory (named for the class's package).

A skeleton for a remote object is a JRMP protocol server-side entity that has a method that dispatches calls to the actual remote object implementation.

A tie for a remote object is a server-side entity similar to a skeleton, but which communicates with the client using the IIOP protocol.

A stub is a client-side proxy for a remote object which is responsible for communicating method invocations on remote objects to the server where the actual remote object implementation resides. A client's reference to a remote object, therefore, is a reference to a local stub.

By default, rmic generates stub classes that use the 1.2 JRMP stub protocol version only, as if the [-v1.2 option](https://docs.oracle.com/javase/7/docs/technotes/tools/windows/rmic.html#v1.2) had been specified. (Note that the [-vcompat option](https://docs.oracle.com/javase/7/docs/technotes/tools/windows/rmic.html#vcompat) was the default in releases before 5.0.) Use the [-iiop option](https://docs.oracle.com/javase/7/docs/technotes/tools/windows/rmic.html#iiop) to generate stub and tie classes for the IIOP protocol.

A stub implements only the remote interfaces, not any local interfaces that the remote object also implements. Because a JRMP stub implements the same set of remote interfaces as the remote object itself, a client can use the Java programming language's built-in operators for casting and type checking. For IIOP, the *PortableRemoteObject.narrow* method must be used.

Rmiregistry-

The rmiregistry command creates and starts a remote object registry on the specified port on the current host. If the port is omitted, the registry is started on port 1099. The rmiregistry command produces no output and is typically run in the background. For example:

*rmiregistry &*

A remote object registry is a bootstrap naming service that is used by RMI servers on the same host to bind remote objects to names. Clients on local and remote hosts can then look up remote objects and make remote method invocations.

The registry is typically used to locate the first remote object on which an application needs to invoke methods. That object in turn will provide application-specific support for finding other objects.

The methods of the *java.rmi.registry.LocateRegistry* class are used to get a registry operating on the local host or localhost and port.

The URL-based methods of the *java.rmi.Naming* class operate on a registry and can be used to look up a remote object on any host, and on the localhost: bind a simple (string) name to a remote object, rebind a new name to a remote object (overriding the old binding), unbind a remote object, and list the URLs bound in the registry.

1. Differentiate between Static RMI and dynamic RMI. Java RMI belongs to which type?

Ans: -

key differences between Static and Dynamic binding in Java :  
  
1) Static binding is resolved at compile-time, while Dynamic binding is resolved at runtime.  
  
2) Static binding only uses Type information, and method resolution is based upon the type of reference variable, while dynamic or late binding resolves method based upon an actual object.  
  
3) In the Java programming language, private, static, and final methods are resolved using static binding, while only virtual methods are resolved using dynamic binding.  
  
4) True Polymorphism is achieved using dynamic binding, and it's key to many design principles e.g. Strategy pattern, Open closed design pattern, etc.

In computing, the Java Remote Method Invocation (Java RMI) is a Java API that performs remote method invocation, the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage collection.

1. What is a proxy? what is a skeleton?

Ans: -

Proxy

The client-side object participating in distributed object communication is known as a stub or proxy and is an example of a [proxy object](https://en.wikipedia.org/wiki/Proxy_object).

The stub acts as a gateway for client-side objects and all outgoing requests to server-side objects that are routed through it. The stub wraps client object functionality and adding the network logic ensures the reliable communication channel between client and server. The stub can be written up manually or generated automatically depending on the chosen communication protocol.

The stub is responsible for:

* initiating the communication towards the server [skeleton](https://en.wikipedia.org/wiki/Class_skeleton)
* translating calls from the caller object
* [marshalling](https://en.wikipedia.org/wiki/Marshalling_(computer_science)) of the parameters
* informing the [skeleton](https://en.wikipedia.org/wiki/Class_skeleton) that the call should be invoked
* passing arguments to the [skeleton](https://en.wikipedia.org/wiki/Class_skeleton) over the network
* [unmarshalling](https://en.wikipedia.org/wiki/Serialization) of the response from the [skeleton](https://en.wikipedia.org/wiki/Class_skeleton)
* informing the caller that the call is complete

Skeleton

The server-side object participating in distributed object communication is known as a skeleton (or stub; term avoided here).

A skeleton acts as the gateway for server-side objects and all incoming clients requests are routed through it. The skeleton wraps server object functionality and exposes it to the clients, moreover adding the network logic ensures the reliable communication channel between clients and server. Skeletons can be written up manually or generated automatically depending on the chosen communication protocol.

* The skeleton is responsible for:
* translating incoming data from the [stub](https://en.wikipedia.org/wiki/Class_stub) to the correct up-calls to server objects
* [unmarshalling](https://en.wikipedia.org/wiki/Serialization) of the arguments from received data
* passing arguments to server objects
* [marshalling](https://en.wikipedia.org/wiki/Marshalling_(computer_science)) of the returned values from server objects
* passing values back to the client [stub](https://en.wikipedia.org/wiki/Class_stub) over the network

**Department of Computer Engineering**

**Academic Term: January - May 2021**

**Class: *B.E (Computer) Semester VIII***

**Subject Name: *Distributed Computing***

**Subject Code: *CSC802***

|  |  |
| --- | --- |
| **Practical No:** | **3** |
| **Title:** | **Multithreaded Server, Client** |
| **Date of Performance:** |  |
| **Date of Submission:** |  |
| **Roll No:** | **8364** |
| **Name of the Student:** | **Vedant Sahai** |

**Evaluation:**

|  |  |  |
| --- | --- | --- |
| **Sr. No** | **Rubric** | **Grade** |
| **1** | **On time Completion & Submission** |  |
| **2** | **Coding Standards** |  |
| **3** | **Output/Test Cases Used** |  |
| **4** | **Conclusion & Post Lab Assignment** |  |

**Signature of the Teacher :**

**Code:**

**Server.java**

import java.io.\*;

import java.net.\*;

// Server class

class Server {

    public static void main(String[] args)

    {

        ServerSocket server = null;

        try {

            // server is listening on port 1234

            server = new ServerSocket(1234);

            server.setReuseAddress(true);

            // running infinite loop for getting

            // client request

            while (true) {

                // socket object to receive incoming client

                // requests

                Socket client = server.accept();

                // Displaying that new client is connected

                // to server

                System.out.println("New client connected"

                                + client.getInetAddress()

                                        .getHostAddress());

                // create a new thread object

                ClientHandler clientSock

                    = new ClientHandler(client);

                // This thread will handle the client

                // separately

                new Thread(clientSock).start();

            }

        }

        catch (IOException e) {

            e.printStackTrace();

        }

        finally {

            if (server != null) {

                try {

                    server.close();

                }

                catch (IOException e) {

                    e.printStackTrace();

                }

            }

        }

    }

    // ClientHandler class

    private static class ClientHandler implements Runnable {

        private final Socket clientSocket;

        // Constructor

        public ClientHandler(Socket socket)

        {

            this.clientSocket = socket;

        }

        public void run()

        {

            PrintWriter out = null;

            BufferedReader in = null;

            try {

                // get the outputstream of client

                out = new PrintWriter(

                    clientSocket.getOutputStream(), true);

                // get the inputstream of client

                in = new BufferedReader(

                    new InputStreamReader(

                        clientSocket.getInputStream()));

                String line;

                while ((line = in.readLine()) != null) {

                    // writing the received message from

                    // client

                    System.out.printf(

                        " Sent from the client: %s\n",

                        line);

                    out.println(line);

                }

            }

            catch (IOException e) {

                e.printStackTrace();

            }

            finally {

                try {

                    if (out != null) {

                        out.close();

                    }

                    if (in != null) {

                        in.close();

                        clientSocket.close();

                    }

                }

                catch (IOException e) {

                    e.printStackTrace();

                }

            }

        }

    }

}

**Client.java**

import java.io.\*;

import java.net.\*;

import java.util.\*;

// Client class

class Client {

    // driver code

    public static void main(String[] args)

    {

        // establish a connection by providing host and port

        // number

        try (Socket socket = new Socket("localhost", 1234)) {

            // writing to server

            PrintWriter out = new PrintWriter(

                socket.getOutputStream(), true);

            // reading from server

            BufferedReader in

                = new BufferedReader(new InputStreamReader(

                    socket.getInputStream()));

            // object of scanner class

            Scanner sc = new Scanner(System.in);

            String line = null;

            while (!"exit".equalsIgnoreCase(line)) {

                // reading from user

                line = sc.nextLine();

                // sending the user input to server

                out.println(line);

                out.flush();

                // displaying server reply

                System.out.println("Server replied "

                                + in.readLine());

            }

            // closing the scanner object

            sc.close();

        }

        catch (IOException e) {

            e.printStackTrace();

        }

    }

}

**Output:**

PS D:\Desktop\BE\experiment\Batch-C-Codes\dc\exp5> java Client

heloo i am vedant

Server replied heloo i am vedant

PS D:\Desktop\BE\experiment\Batch-C-Codes\dc\exp5> java Client

hello i am darren

Server replied heloo i am darren

PS D:\Desktop\BE\experiment\Batch-C-Codes\dc\exp5> java Server

New client connected127.0.0.1

Sent from the client: i am vedant­

New client connected127.0.0.1

Sent from the client: i am darren

**Conclusion:**

Multithreading: Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc. Advantages of multithreading:

• It doesn't block the user because threads are independent and you can perform multiple operations at same time.

• You can perform many operations together so it saves time.

• Threads are independent so it doesn't affect other threads if exception occur in a single thread. Disadvantages of multithreading:

• Complex debugging and testing processes.

• Overhead switching of context.

• Increased potential for deadlock occurrence.

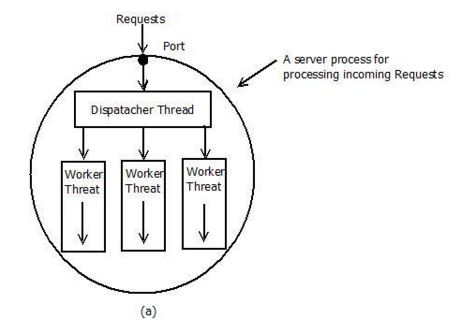
• Increased difficulty level in writing a program.

**Post Lab Assignments**

1. Explain what is a dispatcher-worker model of multiple threads

Ans: -

We have already seen the use of this model in designing a server process. In this model, the process consists of single dispatcher threads and multiples worker threads. The dispatcher the request to one of the free worker threads for further processing of the request. Each worker threads work on a different client request. Therefore, multiple client requests can be processed in parallel. An example of this model is shown in fig.8.7 (a)



1. Different types of multithreading

Ans: -

### Kernel Level Threads

Kernel-level [threads](https://t4tutorials.com/threads-in-operating-systems/) are supported within the kernel of the operating system. Allin system modern operating system support kernel-level threading. They allow the kernel to perform multiple tasks and to service multiple kernel system calls simultaneously.

User Level threads:

User-level threads are implemented in the user library instead of [system calls](https://t4tutorials.com/system-call-and-advantages-of-system-calls/). The thread switching does not need to call the operating system. It does not cause an interrupt to the kernel. The kernel knows nothing about the user-level thread. The user-level threads are very fast.

1. Difference between concurrent server and multi-threaded server?

Ans: -

Multi-threading is a vehicle for accomplishing some concurrency; you use multiple threads to execute your task, in many languages, e.g. in Java or C, you have to code how your task should split and combine across the different threads on your own.

Concurrency is the wider aim, which multi-threading may seek to accomplish. As a somewhat perverted counter-example, you may accomplish concurrency in responding to http requests to your server, by spawning multiple node.js processes. You’ll have concurrency, but you are not (directly) relying on multithreading to accomplish it, even if each node.js process multi-threads regardless of this special setup.

As another case in point, netty and node.js leverage the eventing paradigm for concurrency, so that you can use a single process for request loads orders of magnitude higher than classical http servers which rely on multi-threading for concurrent service.

In other implementations, for example under the so-called Actor Model: each actor may or may not be a different thread or even process, but the actor model abstraction abstracts any multi-threading from you, and you code to actors, not knowing how the actor system multi-threads (up until you need to performance-tune thread usage parameters and such).

So, multi-threading is a core vehicle for accomplishing concurrency on a single machine, but concurrency is a wider aim and concept, which can be met by additional vehicles.

Notwithstanding, multi-threading is a staple ingredient in much of the software and operating system you use, it is just, that for high concurrency it is not always sufficiently good on its own, and additional processor and architectural patterns need to be used for accomplishing extreme concurrency for some types of workloads.