Distributed System Lab

Final Year B. Tech. (COMPUTER ENGINEERING)
Semester - VII

Laboratory Manual

Department of Computer Engineering

R. C. Patel Institute of Technology, Shirpur

Experiment No.1

TITLE: Program for Implementation of RPC

THEORY:

High-level programming through remote procedure calls (RPC) provides logical client-to-server communication for network application development - without the need to program most of the interface to the underlying network. With RPC, the client makes a remote procedure call that sends requests to the server, which calls a dispatch routine, performs the requested service, and sends back a reply before the returns to the client. RPC does not require the caller to know about the underlying network (it looks similar to a call to a C routine). RPC model - Figure below illustrates the basic form of network communication with the RPC.

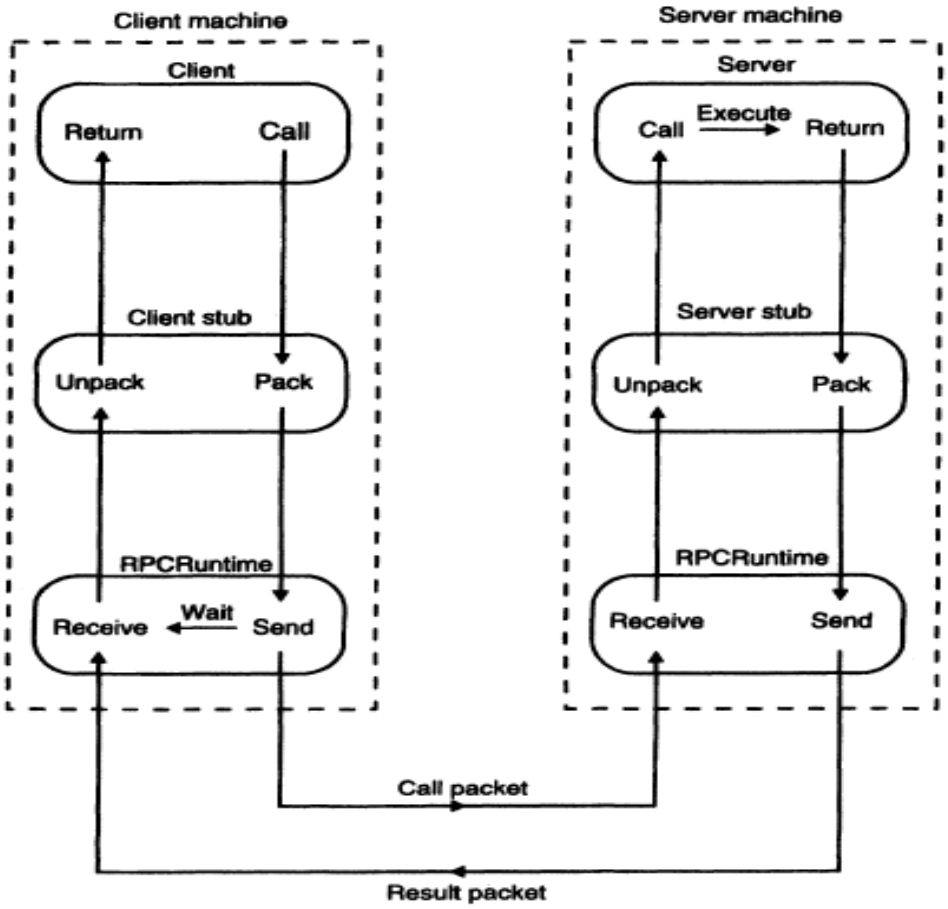


Figure : basic form of network communication with the RPC.

Writing RPC applications with the rpcgen protocol compiler - It accept a remote program interface definition written in RPC language (which is similar to C). It then produces C language output consisting Skeleton versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, a header file that contains common

definitions, and optionally dispatch tables that the server uses to invoke routines that are based on authorization checks. The client skeleton interface to the RPC library hides the network from its callers, and the server skeleton hides the network from the server procedures invoked by remote clients.

The client handle is created in the client program that is generated by rpcgen compiler. The RPC is called through the client. This request passes over the network and reaches server side wherein server stub calls this procedure. The RPC returns the contents of file to the server stub and it travels through the network to the client stub and back to client where it is displayed.

STEPS:

- Create the Specification file, a file with .x extension.
- Compile it using rpcgen compiler which creates the stubs and the client and server programs

Client:

- Specify the RPC header client.
- Using this Create Client handle and invoke a protocol (UDP).
- Call the remote procedure using the handle.
- On return ,display the file contents
- Destroy the client handle
- Stop

Server:

- Declare the static variables for result.
- Open the file for which request came from client
- Read its contents into a buffer.
- Return the buffer as result

Execution Procedure:

- rcpgen compiles the specification file for instance, file.x and generates client stub, server stub, client program and server program.

```
$ rpcgen –a file.x
$ ls
file_client.c file.h file_svc.c Makefile.x file_clnt.c file_server.c file.x
```

- After Stub and programs are generated, Compile the server program and server stub. Similarly do for the Client program and Client.

```
$ cc file_server.c file_svc.c –o obj
$ ./obj& [2]5030
$ cc file_client.c file_clnt.c
$ ./a.out 192.100.100.6
```

Experiment No.2

TITLE: Program for Implementation of RMI

THEORY:

WHAT IS RMI?

RMI stands for Remote Method Invocation. RMI is a mechanism for communicating only between two machines running Java Virtual Machines. When Java code on machine A needs a service or a method, respectively, of obj B on machine B it starts a remote method invocation.
It does this the same way as invoking a Java Local object's method. The whole communication behind the remote invocation is done by the RMI mechanism.

STEPS TO DEVELOP RMI BASED APPLICATION:

1) Create an interface that defines the exported methods that the remote object implements (that is, the methods that the server implements and that clients can invoke remotely). This interface extends the java.rmi.Remoteinterface.
2) Define a subclass of java.rmi.server.UnicastRemoteObject that implements your Remote interface. This class represents the remote object (or server object). The UnicastRemoteObject and the rest of the RMI infrastructure handle this automatically.
3) Write a program (a 'server') that creates an instance of your remote object. Export the object, making it available for use by clients, by registering the object by name with a registry service. This is usually done with the java.rmi.Naming class and the rmiregistry program. A server program may also act as its own registry server by using the LocateRegistry class and registry interface of the java.rmi.registry package.
4) After you compile the server program (with javac) use rmic to generate a 'stub' and 'skeleton' for the remote object. Invoke rmic with the name of the remote object class (not the interface) on the command line. It creates and compiles two new classes with the suffixes _Stub and _Skel. This can be a complicated system but fortunately application programmers never have to think about stubs and skeletons; they are generated automatically by the rmic tool.
5) If the server uses the default registry service provided by the Naming class you must run the registry server, if it is not already running. You can run the registry server by invoking the rmiregistry program. Note however that, as we mentioned at step 3 a server program may also act as its own registry server by using the LocateRegistry class and registry interface of the java.rmi.registry package so you need to run the registry server (or make sure it is running) only if the server uses

the default registry service provided by the Naming class. Once this client has this remote object it can invoke methods on it exactly as it would invoke the methods of a local object.

The only thing that it must be aware of is that all remote methods can throw RemoteException objects, and that in the presence of network errors, this can happen at unexpected times.

6) Finally, start up the server program, and run the client!

Experiment No.3

TITLE: Program to implement clock synchronization (Lamport Algorithm)

THEORY :

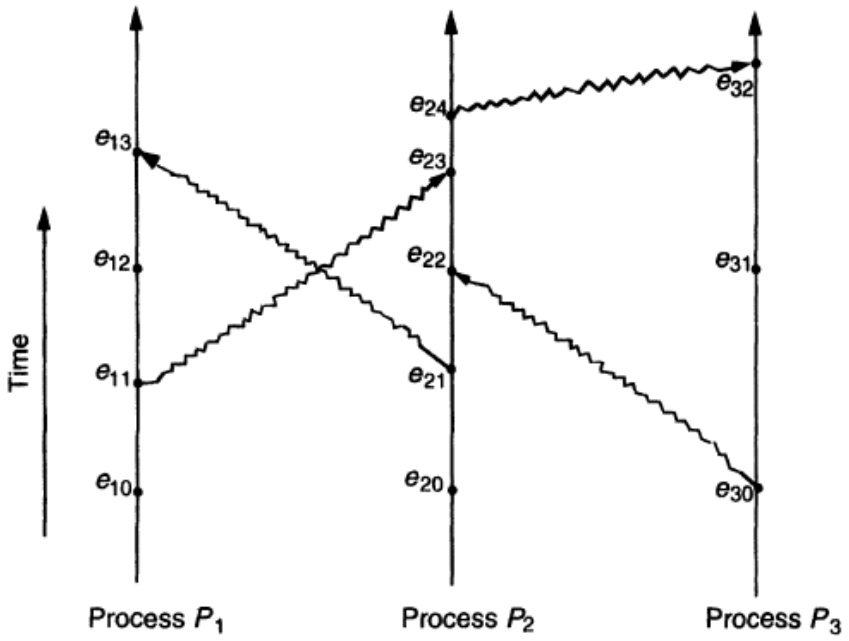
Keeping the clocks in a distributed system synchronized to within 5 or 10msec is an expensive and nontrivial task. Lamport [1978] observed that for most applications it is not necessary to keep the clocks in a distributed system synchronized. Rather, it is sufficient to ensure that all events that occur in a distributed system be totally ordered in a manner that is consistent with an observed behavior.
For partial ordering of events, Lamport defined a new relation called happened before and introduced the concept of logical clocks for ordering of events based on the happened-before relation.

Happened before relationship –

The happened-before relation (denoted by -->) on a set of events satisfies the following conditions:

1. If a and b are events in the same process and a occurs before b, then a → b.
2. If a is the event of sending a message by one process and b is the event of the receipt of the same message by another process, then a → b. This condition holds by the law of causality because a receiver cannot receive a message until the sender sends it, and the time taken to propagate a message from its sender to its receiver is always positive.
3. If a → b and b→ c, then a → c. That is, happened-before is a transitive relation.

In terms of the happened-before relation, two events a and b are said to be concurrent if they are not related by the happened-before relation. That is, neither a --> b nor b --> a is true. This is possible if the two events occur in different processes that do not exchange messages either directly or indirectly via other processes. two events are concurrent if neither can causally affect the other. Due to this reason, the happened-before relation is sometimes also known as the relation of causal ordering.



A space-time diagram (such as the one shown in Fig is often used to illustrate the concepts of the happened-before relation and concurrent events. In this diagram, each vertical line denotes a process, each dot on a vertical line denotes an event in the corresponding process, and each wavy line denotes a message transfer from one process to another in the direction of the arrow. From this space-time diagram it is easy to see that for two events a and b, a --> b is true if and only if there exists a path from a to b by moving forward in time along process and message lines in the direction of the arrows. For example, some of the events of Figure that are related by the happened-before relation are

```
e10 --> e11     e20 --> e24     e11 --> e23     e21 --> e13
e30 --> e24     (since e30 - - > e22  and  e22 --> e24)
e11 --> e32     (since e11 --> e23,  e23 --> e24 , and e24 --> e32 )
```

On the other hand, two events a and b are concurrent if and only if no path exists either from a to b or from b to a. For example, some of the concurrent events of Figure are:

```
e12 and e20     e21 and e30     e10 and e30     e11 and e31
e12 and e32     e13 and e22
```

Experiment

TITLE: Program for Mutual Exclusion.

THEORY :

There are several resources in a system that must not be used simultaneously by multiple processes if program operation is to be correct. For example, multiple processes must not simultaneously update a file. Therefore, exclusive access to such a. shared resource by a process must be ensured. This exclusiveness of access is called mutual exclusion between processes. The sections of a program that need exclusive access to shared resources are referred to as critical sections.
An algorithm for implementing mutual exclusion must satisfy the following requirements:

1. Mutual exclusion. Given a shared resource accessed by multiple concurrent processes, at any time only one process should access the resource. That is, a process that has been granted the resource must release it before it can be granted to another process.
2. No starvation. If every process that is granted the resource eventually releases it, every request must be eventually granted.

one of the processes in the system is elected as the coordinator (algorithms for electing a coordinator are described later in this chapter) and coordinates the entry to the critical sections. Each process that wants to enter a critical section must first seek permission from the coordinator. If no other process is currently in that critical section, the coordinator can immediately grant permission to the requesting process. However, if two or more processes concurrently ask for permission to enter the same critical section, the coordinator grants permission to only one process at a time in accordance with some scheduling algorithm. After executing a critical section, when a process exits the critical section, it must notify the coordinator so that the coordinator can grant permission to another process (if any) that has also asked for permission to enter the same critical section.

Experiment No.08

TITLE: Program for implementation of Election Algorithm.

THEORY:

Several distributed algorithms require that there be a coordinator process in the entire system that performs some type of coordination activity needed for the smooth running of other processes in the system. Since all other processes in the system have to interact with the coordinator, they all must unanimously agree on who the coordinator is. Furthermore, if the coordinator process fails due to the failure of the site on which it is located, a new coordinator process must be elected to take up the job of the failed coordinator. Election algorithms are meant for electing a coordinator process from among the currently running processes in such a manner that at any instance of time there is a single coordinator for all processes in the system.

Election algorithms are based on the following assumptions:

1. Each process in the system has a unique priority number.
2. Whenever an election is held, the process having the highest priority number among the currently active processes is elected as the coordinator.
3. On recovery, a failed process can take appropriate actions to rejoin the set of active processes.

A Ring Algorithm - In this algorithm it is assumed that all the processes in the system are organized in a logical ring. The ring is unidirectional in the sense that all messages related to the election algorithm are always passed only in one direction (clockwise/anticlockwise). Every process in the system knows the .structure of the ring, so that while trying to circulate a message over the ring, if the successor of the sender process is down, the sender can skip over the successor, or the one after that, until an active member is located.

Notice that in this algorithm two or more processes may almost simultaneously discover that the coordinator has crashed and then each one may circulate an election message over the ring. Although this results in a little waste of network bandwidth, it does not cause any problem because every process that initiated an election will receive the same list of active processes, and all of them will choose the same process as the new coordinator.