MEERUT INSTITUTE OF ENGINEERING & TECHNOLOGY, MEERUT

Department of Computer Science and Engineering

Department of Computer Science and Engineering

B. Tech. (Session 2019 –2020)

Lab Manual

Distributed System

Faculty Name: Mr.Shailendra Kumar

INDEX

LAB MANUAL OBJECTIVE

This Distributed System manual is primarily written according to the unified syllabus of "Distributed System" of fourth year B.tech(I.T.) G.B.T.U. syllabus.

This manual clearly explains the concepts in Distributed System.

My **Objective** in writing of this lab manual is to produce a general, comprehensive text that treats all the essential core area of Distributed System.

The **Goal** is to describe the real world constraint and realities of Distributed System.

I have done my best to hunt down and eradicate all errors in this manual.

INTRODUCTION TO DISTRIBUTED SYSTEM

A *distributed system*

> is an application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate together to perform a single or small set of related tasks.

Why build a distributed system? There are lots of advantages including the ability to connect remote users with remote resources in an open and scalable way. When we say *open*, we mean each component is continually open to interaction with other components. When we say *scalable*, we mean the system can easily be altered to accommodate changes in the number of users, resources and computing entities.

Thus, a distributed system can be much larger and more powerful given the combined capabilities of the distributed components, than combinations of stand-alone systems. But it's not easy – for a distributed system to be useful, it must be reliable. This is a difficult goal to achieve because of the complexity of the interactions between simultaneously running components.

To be truly reliable, a distributed system must have the following characteristics:

- Fault-Tolerant: It can recover from component failures without performing incorrect actions.
- Highly Available: It can restore operations, permitting it to resume providing services even when some components have failed.
- Recoverable: Failed components can restart themselves and rejoin the system, after the cause of failure has been repaired.
- Consistent: The system can coordinate actions by multiple components often in the presence of concurrency and failure. This underlies the ability of a distributed system to act like a non-distributed system.
- Scalable: It can operate correctly even as some aspect of the system is scaled to a larger size. For example, we might increase the size of the network on which the system is running. This increases the frequency of network outages and could degrade a "non-scalable" system. Similarly, we might increase the number of users or servers, or overall load on the system. In a scalable system, this should not have a significant effect.
- Predictable Performance: The ability to provide desired responsiveness in a timely manner.
- Secure: The system authenticates access to data and services [1]

These are high standards, which are challenging to achieve. Probably the most difficult challenge is a distributed system must be able to continue operating correctly even when components fail. This issue is discussed in the following excerpt of an interview with Ken Arnold. Ken is a research scientist at Sun and is one of the original architects of Jini, and was a member of the architectural team that designed CORBA.

APPLICATIONS OF DISTRIBUTED SYSTEM

There are two main reasons for using distributed systems and distributed computing. First, the very nature of the application may require the use of a communication network that connects several computers. For example, data is produced in one physical location and it is needed in another location.
Second, there are many cases in which the use of a single computer would be possible in principle, but the use of a distributed system is beneficial for practical reasons. For example, it may be more cost-efficient to obtain the desired level of performance by using a cluster of several low-end computers, in comparison with a single high-end computer. A distributed system can be more reliable than a non-distributed system, as there is no single point of failure. Moreover, a distributed system may be easier to expand and manage than a monolithic uniprocessor system.

Examples of distributed systems and applications of distributed computing include the following
Telecommunication networks:

Telephone networks and cellular networks.
Computer networks such as the Internet.
Wireless sensor networks.
Routing algorithms.
Network applications:
World wide web and peer-to-peer networks.
Massively multiplayer online games and virtual reality communities.
Distributed databases and distributed database management systems.
Network file systems.
Distributed information processing systems such as banking systems and airline reservation systems.
Real-time process control:
Aircraft control systems.
Industrial control systems.
Parallel computation:
Scientific computing, including cluster computing and grid computing and various volunteer computing projects; see the list of distributed computing projects.

LIST OF PRACTICALS

1-Program to implement non token based algorithm for Mutual Exclusion

2-Program to implement Lamport's Logical Clock

3-Program to implement edge chasing distributed deadlock detection algorithm.

4-Program to implement locking algorithm.

5-Program to implement Remote Method Invocation.

6-Program to implement Remote Procedure Call.

7-Program to implement Chat Server.

8-Program to implement termination detection

ALGORITHMS

1.)Lamport's Distributed Mutual Exclusion Algorithm

Lamport's Distributed Mutual Exclusion Algorithm is a contention-based algorithm for mutual exclusion on a distributed system.

Algorithm

Nodal properties

1. Every process maintains a queue of pending requests for entering critical section order. The queues are ordered by virtual time stamps derived from Lamport timestamps.

Algorithm

Requesting process

1. Enters its request in its own queue (ordered by time stamps)
2. Sends a request to every node.
3. Wait for replies from all other nodes.
4. If own request is at the head of the queue and all replies have been received, enter critical section.
5. Upon exiting the critical section, send a release message to every process.

Other processes

1. After receiving a request, send a reply and enter the request in the request queue (ordered by time stamps)
2. After receiving release message, remove the corresponding request from the request queue.
3. If own request is at the head of the queue and all replies have been received, enter critical section.

Message complexity

This algorithm creates $3(N-1)$ messages per request, or $(N-1)$ messages and 2 broadcasts.

Drawbacks

1. There exist multiple points of failure.

2.) Lamport's Logical Clock Algorithm

Lamport logical clock

→: happen before

1) a → b two events occur at the same process
2) a → b for sending event and receiving event of a
3) if a → b and b → c then a → c

→ is a transitive relation.
→ may be referred to as *causally affect*.

Concurrent events
a || b if (not a → b) and (not b → a)

For any two events: either a → b or b → a or a || b.

General assumption: no two events occur at the same time.
Reason: no referencing global time to prove two events occurred at the same time.

Logical clock Algorithm:

Each process I maintains an integer variable X.i.
[IR1] An event occurs at Pi: X.i = X.i + 1
[IR2] When Pi receives a message with time stamp (TS,j) from Pj:
X.i = max(X.i, TS) + 1

Each event on Pi has an associated time stamp (Tsi,I).
Tsi is the X.i right after the event (after applying [IR1] or [IR2].

Total ordering: Any two time stamps of different events can be totally ordered.

(Tsi,I) < (TSj,j) if (Tsi < TSj) or ((Tsi = TSj) and (I < j))

Logical time is a discrete virtual time.
Pi waits for a specific logical time (e.g. X.i = 5) is risky because X.i may jump over 5 (from X.i < 5 to X.i > 5.

Limitations of Lamport's logical clock.

Two events a and b with time stamps Tsa and TSb
If a → b then Tsa < TSb
The reverse is not necessary true.
But, most of the time, we like:
If Tsa < TSb then a → b.

Events a and b may not be causally related because each clock may independently advance due to local events.

3.) Edge Chasing distributed deadlock detection algorithm

Edge-chasing is an algorithm for deadlock detection in distributed systems.

Whenever a process *A* is blocked for some resource, a probe message is sent to all processes *A* may depend on. The probe message contains the process id of *A* along with the path that the message has followed through the distributed system. If a blocked process receives the probe it will update the path information and forward the probe to all the processes it depends on. Non-blocked processes may discard the probe.

If eventually the probe returns to process *A*, there is a circular waiting loop of blocked processes, and a deadlock is detected. Efficiently detecting such cycles in the "wait-for graph" of blocked processes is an important implementation problem.

5.. Remote Method Invocation

RMI (Remote Method Invocation) is a way that a programmer, using the Java programming language and development environment, can write object-oriented programming in which objects on different computers can interact in a distributed network. RMI is the Java version of what is generally known as a remote procedure call (RPC), but with the ability to pass one or more objects along with the request. The object can include information that will change the service that is performed in the remote computer. Sun Microsystems, the inventors of Java, calls this "moving behavior." For example, when a user at a remote computer fills out an expense account, the Java program interacting with the user could communicate, using RMI, with a Java program in another computer that always had the latest policy about expense reporting. In reply, that program would send back an object and associated method information that would enable the remote computer program to screen the user's expense account data in a way that was consistent with the latest policy. The user and the company both would save time by catching mistakes early. Whenever the company policy changed, it would require a change to a program in only one computer.

Sun calls its object parameter-passing mechanism *object serialization*. An RMI request is a request to invoke the method of a remote object. The request has the same syntax as a request to invoke an object method in the same (local) computer. In general, RMI is designed to preserve the object model and its advantages across a network.

RMI is implemented as three layers:

- A stub program in the client side of the client/server relationship, and a corresponding skeleton at the server end. The stub appears to the calling program to be the program being called for a service. (Sun uses the term *proxy* as a synonym for stub.)
- A Remote Reference Layer that can behave differently depending on the parameters passed by the calling program. For example, this layer can determine whether the request is to call a single remote service or multiple remote programs as in a multicast.
- A Transport Connection Layer, which sets up and manages the request.

6. Remote Procedure Call

Remote procedure call (**RPC**) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called **remote invocation** or **remote method invocation**.

An RPC is initiated by the *client*, which sends a request message to a known remote *server* to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process. There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution), unless the client sends an asynchronous request to the server, such as an XHTTP call.

An important difference between remote procedure calls and local calls is that remote calls can fail because of unpredictable network problems. Also, callers generally must deal with such failures without knowing whether the remote procedure was actually invoked. Idempotent procedures (those that have no additional effects if called more than once) are easily handled, but enough difficulties remain that code to call remote procedures is often confined to carefully written low-level subsystems.

Sequence of events during a RPC

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
3. The kernel sends the message from the client machine to the server machine.
4. The kernel on the server machine passes the incoming packets to the server stub.
5. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

Standard contact mechanisms

To let different clients access servers, a number of standardized RPC systems have been created. Most of these use an interface description language (IDL) to let various platforms call the RPC. The IDL files can then be used to generate code to interface between the client and server. The most common tool used for this is RPCGEN.

7.Chat Server

Chat server is a standlone application that is made up the combination of two-application, server application (which runs on server side) and client application (which runs on client side). This application is using for chatting in LAN. To start chatting you must be connected with the server after that your message can broadcast to each and every client.

For making this application we are using some core java features like swing, collection, networking, I/O Streams and threading also. In this application we have one server and any number of clients (which are to be communicated with each other). For making a server we have to run the MyServer file at any system on the network that we want to make server and for client we have to run MyClient file on the system that we want to make client. For running whole client operation we can run the Login.

There are five stages involved:

Step 1: A simple server that will accept a single client connection and display everything the client says on the screen. If the client user types ".bye", the client and the server will both quit.

Step 2: A server as before, but this time it will remain 'open' for additional connection once a client has quit. The server can handle at most one connection at a time.

Step 3: A server as before, but this time it can handle multiple clients simultaneously. The output from all connected clients will appear on the server's screen.

Step 4: A server as before, but this time it sends all text received from any of the connected clients to all clients. This means that the server has to receive and send, and the client has to send as well as receive

Step 5: Wrapping the client from step 4 into a very simple GUI interface but not changing the functionality of either server or client. The client is implemented as an Applet, but a Frame would have worked just as well (for a stand-alone program).

PROGRAMS

1.) Write a program in c for implementation of non token base algorithm for distributed mutual exclusion.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,d,p,a,c=0,aa[10],j,n;
char ch='y';
clrscr();
printf("enter no of processes");
scanf("%d",&n);
i=0;
do
{
printf("enter the process no which want to execute critical section");
scanf("%d",&a);
aa[i]=a;
i++;
c=c+1;
d=i;
printf("some other process want to execute cs? then press y");
fflush(0);
scanf("%c",&ch);
}
while(ch=='y');
for(j=1;j<=c;j++)
{
printf("\ncritical section is executing for process %d in queue.....",j);
printf("\ncritical section is finished for process %d",j);
printf("\nrelease msg has sent by process%d",j);
}
getch();
}
```

2.)**Write a program in C to implement Lamports logical clock**

```
#include<stdio.h>
#include<conio.h>
struct process
{
 int e;
 int ts[10];
}p[10];
void main()
{
  int i,j,n,m,t,e1,e2;
  char ch;
  clrscr();
   printf("enter the no. of process");
   scanf("%d",&n);
   for(i=0;i<n;i++)
    {
    printf("enter the no. of events in process %d",i+1);
    scanf("%d",&p[i].e);
    for(j=0;j<p[i].e;j++)
            {
             p[i].ts[j]=j+1;
            }
    }
    for(i=0;i<n;i++)
    {
     for(j=0;j<p[i].e;j++)
     printf("%d ",p[i].ts[j]);
     printf("\n");
     }
     do
     {    printf("enter the process no & event no. from which message is passing (less than %d)",n);
          scanf("%d %d",&m,&e1);
          printf("enter the process no & event no. on which msg is passing (less than %d)",n);
         scanf("%d %d",&t,&e2);
         if((p[m].ts[e1]+1)>p[t].ts[e2])
                {
                 p[t].ts[e2]=p[m].ts[e1]+1;
                  for(i=e2;i<p[t].e;i++)
                       {
                        p[t].ts[i+1]=p[t].ts[i]+1;
                       }
                }
          printf("is there more message(y/n)");
          fflush(0);
          scanf("%c",&ch);
    }while(ch=='y'&& ch=='Y');
     for(i=0;i<n;i++)
    {
     for(j=0;j<p[i].e;j++)
```

```
      printf("%d ",p[i].ts[j]);
      printf("\n");
     }
  getch();
}
```

3.) Write a program to implement edge chasing distributed deadlock detection algorithm.

```
#include<stdio.h>
#include<conio.h>

void main()
{
int temp,process[10][15], site_count=0, process_count=0,i,j,k,waiting[15];
int p1,p2,p3;
clrscr();
printf("\n Enter the no. of sites (max 3) \n");
scanf("%d",&site_count);

for(i=1;i<=site_count;i++)
{
printf("\n Enter the no. of processes in %d site ( max 4)\n",i);
scanf("%d",&process_count);
for(j=0;j<process_count;j++)
{
process[i][j]=i+(i*j);
}
}
printf("\n Enter the blocked process \n");
scanf("%d",&k);
for(i=1;i<=3;i++)
{
for(j=0;j<=3;j++)
{
 if(k==process[i][j])
{
 printf("Process %d is at site %d ",k,i);
 temp=i;
}
if(k==process[i][j])
printf("It is a deadlock \n");
if(k==(process[temp][j])&&((process[temp][j])==waiting[process[i][j]]) && (temp!=i))
{
//probe(temp,j,process[i][j]);
if(process[i][j]==waiting[process[temp][j]]);
printf("It is a deadlock\n");
}
}
}getch();}
```

4.) Write a program in C to implement locking algorithm.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=0;
char b,c;
clrscr();
do
{
printf("if transaction T1 want to lock data object");
fflush(0);
scanf("%c",&b);
if(a==0 && b=='y')
{
a=1;
b='n';
}
else if(a==1)
printf("data object is locked");
printf("if transaction T2 want to lock data object");
fflush(0);
scanf("%c",&b);
if(a==0 && b=='y')
{
a=1;
b='n';
}
else
printf("data object is locked");
printf("\nif transaction want to release data object");
fflush(0);
scanf("%c",&b);
if(a==1 && b=='y')
a=0;
printf("do you want to continue");
fflush(0);
scanf("%c",&c);
}while(c=='y');
getch();
}
```

5.)Write a program to implement Remote Method Invocation

```
import java.rmi.*;
public class AddClient
{
public static void main(String args[])
{
try
{
String addServerURL = "rmi://" + args[0] + "/Addserver";
AddServerIntf addServerIntf = (AddServerIntf)Naming.lookup(addServerURL);
System.out.println("the first number is :"+args[1]);
double d1 =Double.valueOf(args[1]).doubleValue();
System.out.println("the second number is :"+args[2]);
double d2 =Double.valueOf(args[2]).doubleValue();
System.out.println("The sum is:" + addServerIntf.add(d1,d2));
}
catch (Exception e)
{
System.out.println(" Exception:" +e);
}
}
}
```

```
//Add Server

import java.net.*;
import java.rmi.*;
public class AddServer
{
public static void main(String [] args)
{
try
{
AddServerImpl addServerImpl = new AddServerImpl();
Naming.rebind("Addserver",addServerImpl);
}
catch(Exception e)
{
System.out.println("Exception: " +e);
}
}
}
```

```
// Add Server IMPL

import java.rmi.*;
```

```
import java.rmi.server.*;

public class AddServerImpl extends UnicastRemoteObject implements AddServerIntf
{
public AddServerImpl() throws RemoteException
{
}
public double add(double d1, double d2) throws RemoteException
{
return d1+d2;
}}



//ADD SERVER INTF

import java.rmi.*;
public interface AddServerIntf extends Remote
{
double add(double d1, double d2) throws RemoteException;
}
```

6.) Write a Program to implement Remote Procedure Call.

```
/*
* rdate.c client program for remote date program
*/
#include <stdio.h>
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */
main(int argc, char *argv[])
{
CLIENT *cl; /* RPC handle */
char *server;
long *lresult; /* return value from bin_date_1() */
char **sresult; /* return value from str_date_1() */
if (argc != 2) {
fprintf(stderr, "usage: %s hostname\n", argv[0]);
exit(1);
}
server = argv[1];
/*
* Create client handle
*/
if ((cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
/*
* can’t establish connection with server
*/
clnt_pcreateerror(server);
exit(2);
}
/*
* First call the remote procedure "bin_date".
*/
if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
clnt_perror(cl, server);
exit(3);
}
printf("time on host %s = %ld\n",server, *lresult);
/*


* Now call the remote procedure str_date
*/
if ( (sresult = str_date_1(lresult, cl)) == NULL) {
clnt_perror(cl, server);
exit(4);
}
printf("time on host %s = %s", server, *sresult);
clnt_destroy(cl); /* done with the handle */
exit(0);
}
```

```
/*
* date.x Specification of the remote date and time server
*/
/*
* Define two procedures
* bin_date_1() returns the binary date and time (no arguments)
* str_date_1() takes a binary time and returns a string
*
*/
program DATE_PROG {
version DATE_VERS {
long BIN_DATE(void) = 1; /* procedure number = 1 */
string STR_DATE(long) = 2; /* procedure number = 2 */
} = 1; /* version number = 1 */
} = 0x31234567;
```

```
/*
* dateproc.c remote procedures; called by server stub
*/
#include <time.h>
#include <rpc/rpc.h> /* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */
/*
* Return the binary date and time
*/
long *bin_date_1_svc(void *arg, struct svc_req *s)
{
static long timeval; /* must be static */
timeval = time((long *) 0);
return(&timeval);
}
/*
* Convert a binary time and return a human readable string
*/
char **str_date_1_svc(long *bintime, struct svc_req *s)
{
static char *ptr; /* must be static */
ptr = ctime((const time_t *)bintime); /* convert to local time */
return(&ptr);
}
```

7.)Write a program to implement Chat Server

//Server

```
import java.net.*;
import java.io.*;
public class servertcp
{
public static void main(String args[]) throws IOException
{
ServerSocket ss=new ServerSocket(12);
Socket s=ss.accept();
System.out.println("Connection from "+s);

PrintWriter pw1=new PrintWriter(s.getOutputStream(),true);
BufferedReader br3=new BufferedReader(new InputStreamReader(s.getInputStream()));
BufferedReader br4=new BufferedReader(new InputStreamReader(System.in));
System.out.println("i m ready");
String s1, s2;
/*while((s1=br4.readLine())!=null)
{
pw1.println(s1);
System.out.println(s2);
}
*/
while(true)
{
do
{
s1=br4.readLine();
pw1.println(s1);
}
while(!s1.equals("over"));
do
{
s2=br3.readLine();
System.out.println(s2);
}
while(!s2.equals("over"));
}
}
}
```

// Client

```
import java.net.*;
import java.io.*;
public class clienttcp
{
```

```
public static void main(String args[]) throws IOException
{
Socket cc=new Socket(InetAddress.getLocalHost(),12);
PrintWriter pw=new PrintWriter(cc.getOutputStream(),true);
BufferedReader br1=new BufferedReader(new InputStreamReader(cc.getInputStream()));
BufferedReader br2=new BufferedReader(new InputStreamReader(System.in));
String str1, str2;
/*while((str1=br2.readLine())!=null)
{
System.out.println(str1);
//pw.println(str2);
}
*/
while(true)
{
do
{
str1=br1.readLine();
System.out.println(str1);
}
while(!str1.equals("over"));
do
{
str2=br2.readLine();
pw.println(str2);
}
while(!str2.equals("over"));
}
}
}
```

8.) Write a Program to implement termination detection

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<stdlib.h>
#include<math.h>
void main()
{
int i,j,k=0,n,tw,total=0,we,ca,w[20];
clrscr();
printf("enter the no of process=");
scanf("%d",&n);
printf("\n\nassign a controlling agent=");
scanf("%d",&ca);
printf("\n\nenter the total weight=");
scanf("%d",&tw);
while((k<n))
{
randomize();
w[k]=random(tw);
tw=tw-w[k];
k++;
}
for(k=0;k<n;k++)
{total=total+w[k];
```

```
}
printf("%d",total);
w[n-1]=abs(tw-total);
printf("%d",w[n-1]);
printf("\n\n\t\t\t\tControlling agent%d %d\n\n\n",ca,w[ca]);
printf("\n\nsending computational message to...\n\n");
for(j=0;j<n;j++)
{
if(j!=(ca-1))
{
sound(700);
delay(2000);
printf("\tprocess%d %d",j+1,w[j]);
}}
nosound();
getch();
}
```