Lab Manual

DISTRIBUTED OPERATING SYSTEM

Practical No. 1

Aim: Study of LOCUS distributed operating system.

Theory:

LOCUS is a Unix compatible, distributed operating system in operational use at UCLA on a set of 17 Vax/750's connected by a standard Ethernet. The system supports a very high degree of network transparency, i.e. it makes the network of machines appear to users and programs as a single computer; machine boundaries are completely hidden during normal operation. Both files and programs can be moved dynamically with no effect on naming or correct operation. Remote resources are accessed in the same manner as local ones. Processes can be created locally and remotely in the same manner, and process interaction is the same, independent of location. Many of these functions operate transparently even across heterogeneous cpus. LOCUS also provides a number of high reliability facilities, including flexible and automatic replication of storage at a file level, a full implementation of nested transactions, uad a substantially more robust data storage facility than conventional Unix systems. All of the functions reported here have been implemented, and most are in routine use.

Distributed Fllesystem:

I Filesystem overview:

The LOCUS filesystem presents a single tree structured naming hierarchy to users and applications. It is functionally a superset of the Unix tree structured naming system. There are three major areas of extension. First, the single tree structure in LOCUS covers all objects in the filesystem on all machines. LOCUS names are fully transparent; it is not possible from the name of a resource to discern its location in the netwol'k. Such location transparency is critical for allowing data and programs in general to move or even be executed from different sites. The second direction of extension concerns replication. Files in LOCUS can be replicated to varying degrees, and it is the LOCUS system's responsibility to keep all copies up to date, assure that access requests are served by the most recent available version, and support partitioned operation.

To a first approximation, the pathname tree is made up of a collection of filegroups, as in a conventional Unix environment. Each group is a wholly self contained subtree of the naming hierarchy, including storage for all files and directories contained in the subtree. Gluing together a collection of filegroups to construct the uniform naming tree is done via the mount mechanism. Logically mounting a filegroup attaches one tree (the filegroup being mounted} as a subtree within an already mounted tree. The glue which allows smooth path traversals up and down the expanded naming tree is kept as operating system state information. Currently this state information is replicated at all sites. To scale a LOCUS network to hundreds or thousands of sites, this "mount" information would be cached. The term filegroup in this paper corresponds directly to the Unix term filesystem.

2 File Replication:

Replication of storage in a distributed filesystem serves multiple purposes. First, from the users' point of view, multiple copies of data resources provide the opportunity for substantially increased availability. This improvement is clearly the ease for read access, although the situation is more complex when update is desired, since if some of the copies are not accessible at a given instant, potential inconsistency problems may preclude update,

thereby decreasing availability as the level of replication is increased. The second advantage, from the user viewpoint, concerns performance. If users of the file exist on different machines, and copies are available near those machines, then read access can be substantially faster compared to the necessity to have one of the users always make remote accesses. This difference can be substantial; in a slow network, it is overwhelming, but in a high speed local network it is still significant.

In a general purpose distributed computing environment, such as LOCUS, some degree of replication is essential in order for the user to be able to work at all. Certain files used to set up the user's environment must be available even when various machines have failed or are inaccessible. The startup files in Multics, or the various Unix shells, are ob- In the LOCUS system, which is highly optimized for remote access, the cpu overhead of accessing a remote page is twice local access, and the cost of a remote open is significantly more than the case when the entire open can be done locally. Mail aliases and routing information are others. Of course, these cases can generally be handled by read-only replication, which in general imposes fewer problems. From the system point of view, some form of replication is more than convenient; it is absolutely essential for system data structures, both for availability and performance. Consider a file directory. A hierarchical name space in a distributed environment implies that some directories will have entries which
refer to files on different machines. There is strong motivation for storing a copy of all the directory entries in the backward path from a file at the site where the file is stored, or at least "nearby". The principal reason is availability. If a directory entry in the naming path to a file is not accessible because of network partition or site failure, then that file cannot be accessed, even though it may be stored locally.

LOCUS supports replication at the granularity of the entire directory {as opposed to the entry granularity} to address this issue. Second, directories in general experience a high level of read access compared to update. As noted earlier, this 'characteristic is precisely the one for which a high degree of replicated storage will improve system performance. In the case of the file directory hierarchy, this improvement is critical. In fact, the access characteristics in a hierarchical directory system are, fortunately, even better behaved than just indicated. Typically, the top of the hierarchy exhibits a very high level of lookup, and a correspondingly low rate of update. This pattern occurs because the root of the tree is heavily used by most programs and users as the starting point for name resolution. Changes disrupt programs with embedded names, and so are discouraged. The pattern permits (and requires} the root directories to be highly replicated, thus improving availability and performance simultaneously. By contrast, as one moves down the tree toward the leaves, the degree of shared use of any given directory tends to diminish, since directories are used to organize the name space into more autonomous subspaces. The desired level of replication for availability purposes tends to decrease as well. Further, the update traffic to directories near the leaves of the naming tree tends to be I The problems which remain are present because few files are strictly read-only; it is just that their update rate is low. When an update is done, some way to make sure that all copies are consistent is needed. If the rate is low enough, manual methods may suffice. greater, so one would have less directory replication to improve performance. The performance tradeoffs between update/read rates and degree of replication are well known, and we have already discussed them. However, there are other costs as well. For example, concurrency control becomes more expensive. Without replication the storage site can provide concurrency control for the object since it will know about all activity. With replication some more complex algorithm must be supported. In a similar way, with replication, a choice must be made as to which copy of an object will supply service when there is activity on the object.

This degree of freedom is not available without replication. If objects move, then, in the no replication case, the mapping mechanism must be more general. With replication a move of an object is equivalent to an add followed by a delete of an object copy.

3 Accessing the Filesystem:

There were several goals directing the design of the network-wide file access mechanism. The first was that the system call interface should be uniform, independent of file location. In other words, the same system call with the same parameters should be able to access a file whether the file is stored locally or not. Achieving this goal of transparency would allow programs to move from machine to machine and allow data to be relocated. The primary system calls dealing with the filesystem are open, create, read, write, commit, close and unlink. After introducing the three logical sites involved in file access and the file access synchronization aspect of LOCUS, these system calls are considered in the context of the logical tasks of file reading, modifying, creating and deleting.

3.1 LOCUS Logical Sites for Filesystem Activities:

LOCUS is designed so that every site can be a full function node. As we saw above, however, filesystem operations can involve more than one host. In fact there are three logical functions in a file access and thus three logical sites. These are: a. using site, (US), which issues the request to open a file and to which pages of the file are to be supplied, b. storage site, (SS), which is the site at which a copy of the requested file is stored, and which has been selected to supply pages of that file to the using site, c. current synchronization site, {CSS), which enforces a global access synchronization policy for the file's filegroup and selects SSs for each open request. A given physical site can be the CSS for any number of filegroups but there is only one CSS for any given filegroup in any set of communicating sites (i.e. a partition).

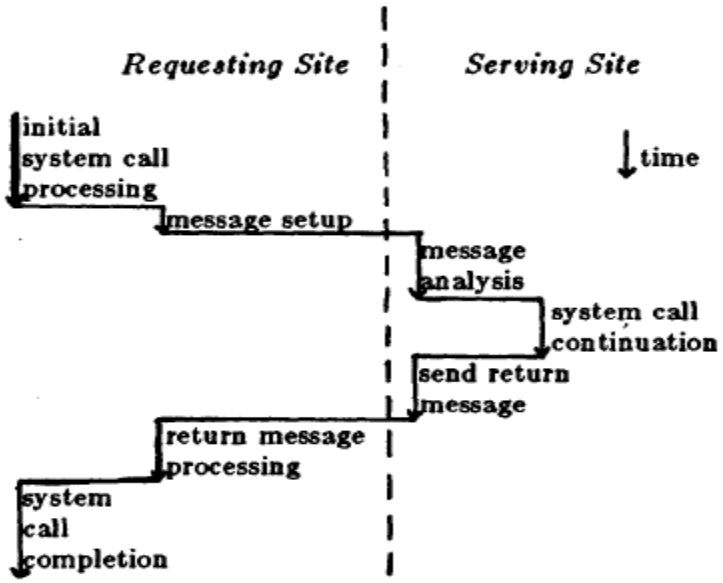


Figure 1: Processing a System Call Requiring Foreign Service

The CSS need not store any particular file in the filegroup but in order for it to make appropriate access decisions it must have knowledge of which sites store the file and what the most current version of the file is. Since there are three possible independent roles a given site can play (US, CSS, SS), it can therefore operate in one of eight modes. LOCUS handles each combination, optimizing some for performance. Since all open requests for a file go through

the CSS function, it is possible to implement a large variety of synchronization policies. In LOCUS, so long as there is a copy of the desired resource available, it can be used. If .there are multiple copies present, the most efficient one to access is selected. Other copies are updated in background, but the system remains responsible for supplying a mutually consistent view to the user. Within a set of communicating sites, synchronization facilities and update propagation mechanisms assure consistency of copies, as well as guaranteeing that the latest version of a file is the only one that is visible. Since it is important to allow modification of a file even when all copies are not currently accessible.

Result: Thus, we have studied LOCUS distributed operating system.

Implementing Lamport's Logical Clocks

Theory:

Lamport's Logical Clocks

An important paper to read - "Time, clocks, and the ordering of events in a distributed system" by Lamport (1978).The important contribution of Lamport is that in a distributed system, clocks need not be synchronized absolutely.

If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus not cause problems.It is not important that all processes agree on what the actual time is, but that they agree on the order in which events occur.

Rules of Lamport's Logical Clocks:

Defines a relationship called "happens-before".

1.ifaa -> bb is read as "a happens before b"if aa and bb are events in the same process and aa occurs before bb, then aa -> bb is true.

2. ifaa is the event of a message being sent by one process and bb is the event of the message being received by another process, then aa -> bb is true

"happens-before" is transitive, meaning if aa -> bb and bb -> cc, then aa -> cc

ifaa -> bb happens between two process, and events xx and yy occur on another set of processes and these two sets of processes don't exchange messages then:we cannot say whether xx ->yy or yy -> xx from the perspective of the first set of processes

- When a message is transmitted from P1 to P2, P1 will encode the send time into the message.
- When P2 receives the message, it will record the time of receipt
- If P2 discovers that the time of receipt is before the send time, P2 will update its software clock to be one greater than the send time (1 milli second at least)
- If the time at P2 is already greater than the send time, then no action is required for P2
- With these actions the "happens-before" relationship of the message being sent and received is preserved.

Lamport Clocks:

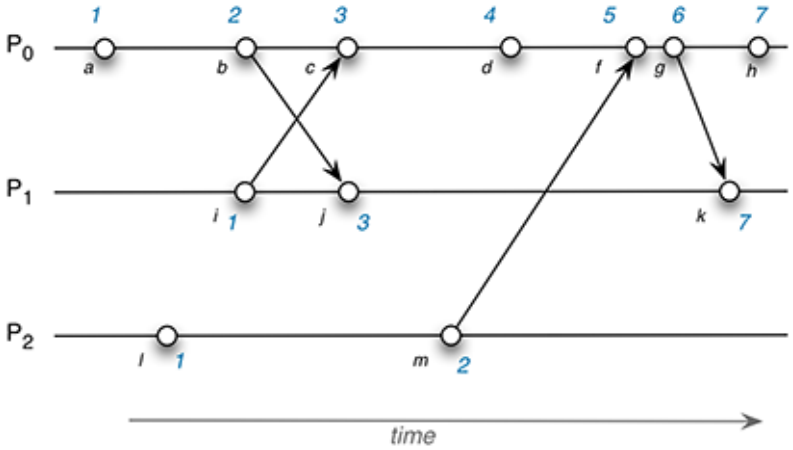
Each process maintains a single Lamport timestamp counter. Each event on the process is tagged with a timestamp from this counter. The counter is incremented before the event timestamp is assigned. If a process has four events, a, b, c, d, they would get Lamport timestamps of 1, 2, 3, 4.

If an event is the sending of a message then the timestamp is sent along with the message. If an event is the receipt of a message then the the algorithm instructs you to compare the current value of the process' timestamp counter (which was just incremented before this event) with the timestamp in the received message. If the timestamp of the received message is greater than that of the current system, the system timestamp is updated with that of the timestamp in the received message plus one. This ensures that the timestamp of the received event and all further

timestamps will be greater than that of the timestamp of sending the message as well as all previous messages.

In the figure below, event k in process P1 is the receipt of the message sent by event g in P0. If event k was just a normal local event, the P1 would assign it a timestamp of 4. However, since the received timestamp is 6, which is greater than 4, the timestamp counter is set to 6+1, or 7. Event k gets the timestamp of 7. A local event after k would get a timestamp of 8.

With Lamport timestamps, we're assured that two causally-related events will have timestamps that reflect the order of events. For example, event c happened before event k in the Lamport causal sense: the chain of causal events is c→d, d→f, f→g, and g→k. Since the happened-before relationship is transitive, we know that c→k (c happened before k). The Lamport timestamps reflect this. The timestamp for c (3) is less than the timestamp for k (7). However, just by looking at timestamps we cannot conclude that there is a causal happened-before relation. For instance, because the timestamp for l (1) is less than the timestamp for j (3) does not mean that l happened before j. Those events happen to be concurrent but we cannot discern that by looking at Lamport timestamps. We need need to employ a different technique to be able to make that determination. That technique is the use of vector timestamps.
lamport clocks



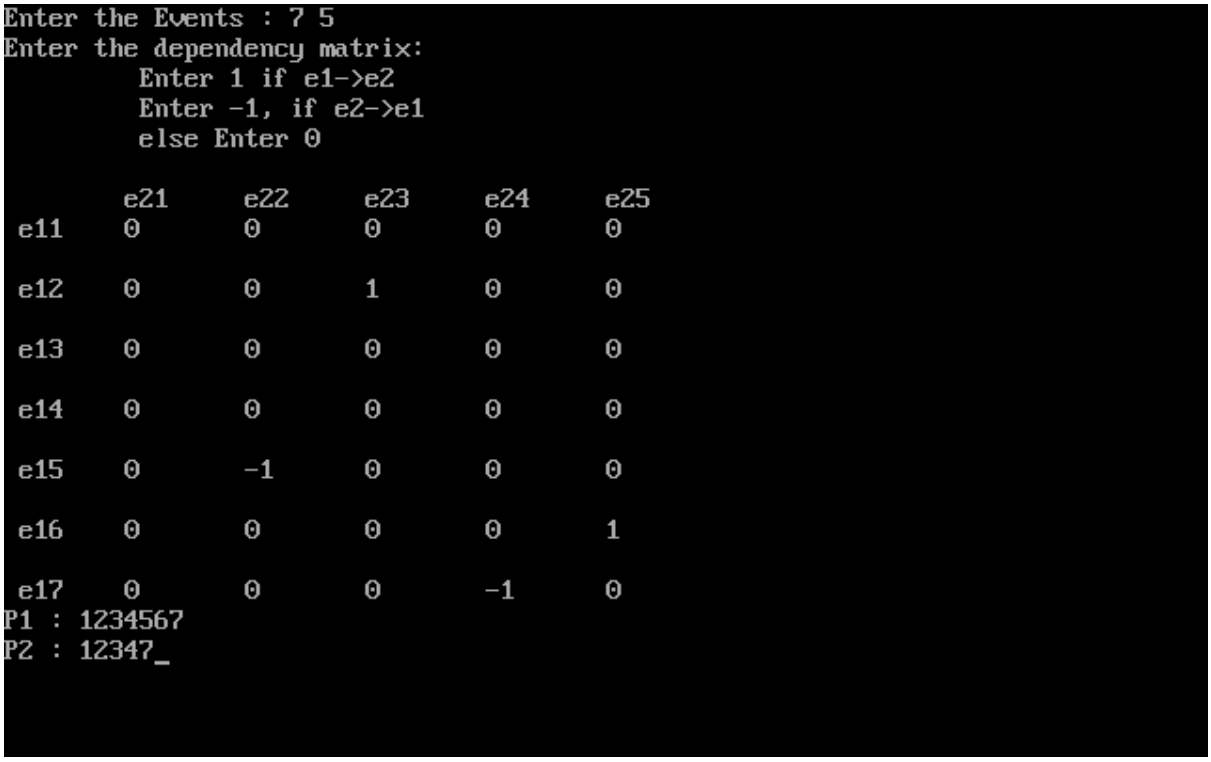
Lamport Clock Assignments

Program: 01

```
#include<stdio.h>
#include<conio.h>
int max1(int a, int b)
{
        if (a>b)
        return a;
else
return b;
}

int main()
{
int i,j,k,p1[20],p2[20],e1,e2,dep[20][20];
```

```
printf("enter the events : ");
scanf("%d %d",&e1,&e2);
for(i=;i<e1;i++)
p1[i]=i+1;
for(i=;i<e2;i++)
p2[i]=i+1;
printf("enter the dependency matrix:\n");
printf("\t enter 1 if e1->e2 \n\t enter -1, if e2->e1 \n\t else enter 0 \n\n");
for(i=;i<e2;i++)
printf("\te2%d",i+1);
for(i=;i<e1;i++)
{
printf("\n e1%d \t",i+1);
for(j=;j<e2;j++)
scanf("%d",&dep[i][j]);
}

for(i=;i<e1;i++)
{
        for(j=;j<e2;j++)
        {
                if(dep[i][j]==1)     //change the timestamp if dependency exist
                {       p2[j]=max1(p2[j],p1[i]+1);
                        for(k=j;k<e2;k++)
                        p2[k+1]=p2[k]+1;
                }
                if(dep[i][j]==-1)    //change the timestamp if dependency exist
                {
                        p1[i]=max1(p1[i],p2[j]+1);
                        for(k=i;k<e1;k++)
                        p2[k+1]=p1[k]+1;
                }

        }
}
printf("P1 : ");     //to print the outcome of Lamport Logical Clock
for(i=;i<e1;i++)
{
printf("%d",p1[i]);
}
printf("\n P2 : ");
for(j=;j<e2;j++)
printf("%d",p2[j]);
```

getch();
return ;
}
Output:



Practical No. 02

Program :
Vector Clock Algorithm :

Vector Clocks

- Vector clocks allow causality to be captured
- **Rules of Vector Clocks:**
  - A vector clock $VC(a)$VC(a) is assigned to an event a
  - If $VC(a)<VC(b)$VC(a)<VC(b) for events a and b, then event a is known to causally precede b.
- **Each Process** $P_iP_i **maintains a vector** $VC_iVC_i **with the following properties:**
  - $VC_i[i]$VCi[i] is the number of events that have occurred so far at $P_iP_i.
    i.e. $VC_i[i]$VCi[i] is the local logical clock at process $P_iP_i

- If $VC_i[j]=kVCi[j]=k$ then P_iPi knows that kk events have occurred at P_jPj. It is thus P_iPi's knowledge of the local time at P_jPj

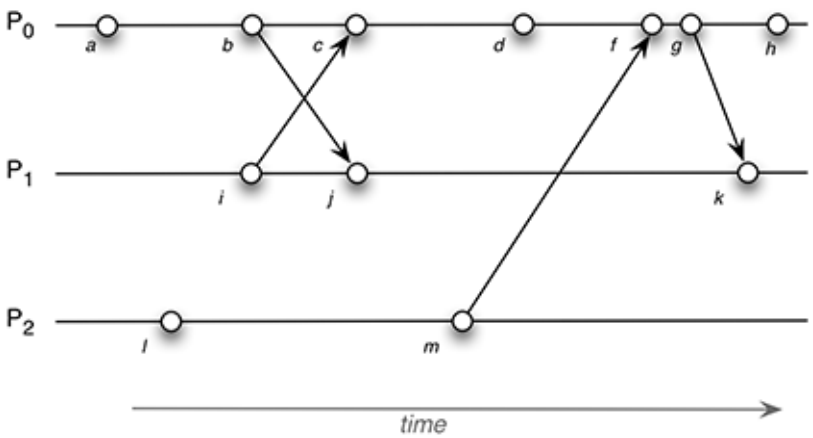
Implementing Vector Clocks

- The first property of the vector clock is accomplished by incrementing $VC_i[i]VCi[i]$ at each new event that happens at process P_iPi
- **The second property is accomplished by the following steps:**
  1. Before executing any event (sending a message or some internal event), P_iPi executes $VC_i[i]VCi[i] <- VC_i[i]+1VCi[i]+1$
  2. When process P_iPi sends a message mm to P_jPj, it sets m's (vector) timestamp $ts(m)=VC_its(m)=VCi$
  3. Upon receiving a message mm, process P_jPj adjusts its own vector by setting $VC_j[k]VCj[k] <- max(VC_j[k],ts(m)[k])max(VCj[k],ts(m)[k])$ for each k.

Vector Clocks

With vector clocks, we assume that we know the number of processes in the group. Our timestamp is now a vector of numbers, with each element corresponding to a process. Each process knows its position in the vector. For example, in the example below, the vector corresponds to the processes (P_0, P_1, P_2).

As with Lamport's algorithm, the element corresponding to the processor in the vector timestamp is incremented prior to attaching a timestamp to an event. If a process P_0 has four events, *a, b, c, d*, they would get Vector timestamps of *(1,0,0), (2, 0, 0), (3, 0, 0), (4, 0, 0)*. If a process P_2 has four events, *a, b, c, d*, they would get Vector timestamps of *(0,0,1), (0, 0, 2), (0, 0, 3), (0, 0, 4)*.



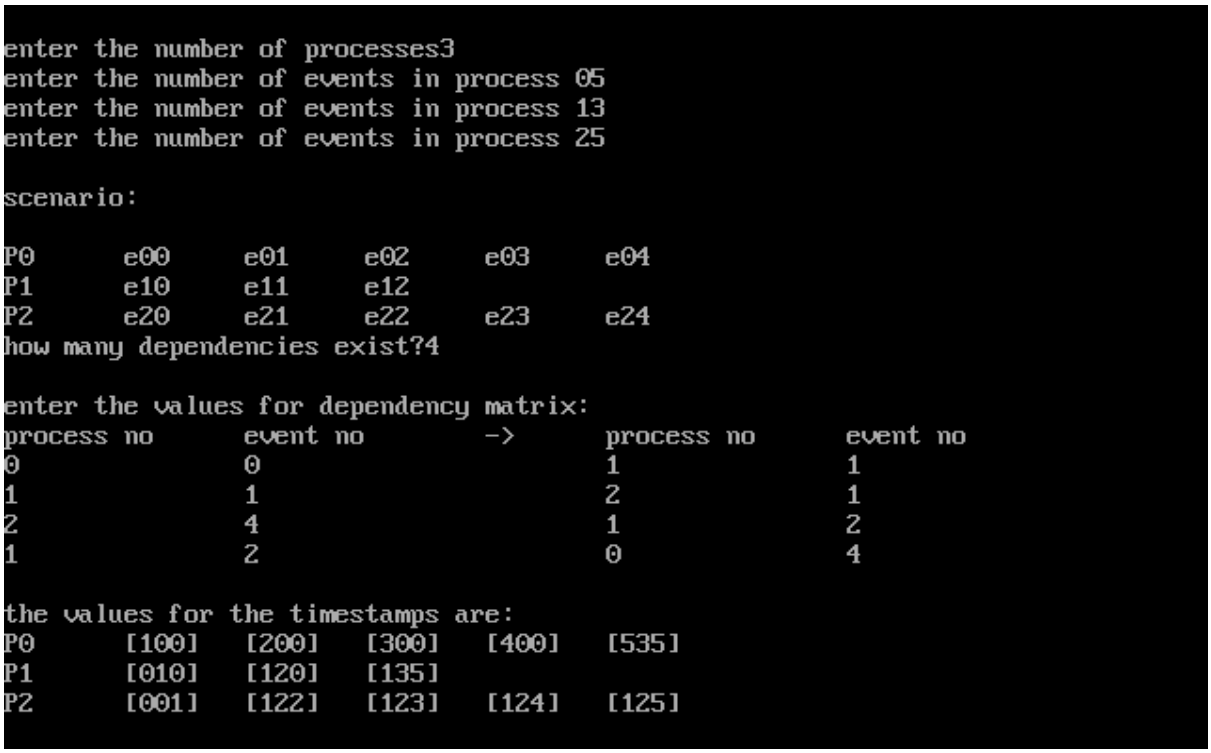

```
#include<stdio.h>
#define MAX 20
int d=1;
struct event
{
int tm[MAX];
 //intpno;
```

```
inteno;
};
struct process
{
int ne;
struct event e[MAX];
}
p[MAX];
main()
{
inti,j,k,pn,dno,tm;
intbefore_proc,after_proc,before_event,after_event;
int dependency[4][MAX];
clrscr();
printf("\nenter the number of processes");
scanf("%d",&pn);
for(i=0;i<pn;i++)
 {
printf("enter the number of events in process %d",i);
scanf("%d",&p[i].ne);
for(j=0;j<p[i].ne;j++)
   {
p[i].e[j].eno=j;
for(k=0;k<pn;k++)
p[i].e[j].tm[k]=0;
p[i].e[j].tm[i]=j+d;
   }
 }
 /*printf("\ntime stamps are:");
for(i=0;i<pn;i++)
 {
printf("\nP%d",i);
for(j=0;j<p[i].ne;j++)
   {
printf("\t[");
for(k=0;k<pn;k++)
printf("%d",p[i].e[j].tm[k]);
printf("]\t");
   }
 } */
printf("\nscenario:\n");
for(i=0;i<pn;i++)
 {
printf("\nP%d",i);
for(j=0;j<p[i].ne;j++)
```

```
printf("\te%d%d",i,p[i].e[j].eno);
 }
printf("\nhow many dependencies exist?");
scanf("%d",&dno);
printf("\nenter the values for dependency matrix:\nprocess no\tevent no\t->\tprocess no\tevent
no\n");
for(i=0;i<dno;i++)
for(j=0;j<4;j++)
scanf("%d",&dependency[i][j]);
 /* printf("\ndependency matrix");
for(i=0;i<dno;i++)
 {
for(j=0;j<4;j++)
printf("\t%d",dependency[i][j]);
printf("\n");
 } */
for(i=0;i<dno;i++)
 {
before_proc=dependency[i][0];
before_event=dependency[i][1];
after_proc=dependency[i][2];
after_event=dependency[i][3];
   //printf("\n%d",before_proc);
for(k=0;k<pn;k++)
if(p[after_proc].e[after_event].tm[k]<(p[before_proc].e[before_event].tm[k]))
   {
p[after_proc].e[after_event].tm[k]=p[before_proc].e[before_event].tm[k];
for(j=after_event+1;j<p[after_proc].ne;j++)
p[after_proc].e[j].tm[k]=p[after_proc].e[j-1].tm[k];
   }
 }
printf("\nthe values for the timestamps are:");
for(i=0;i<pn;i++)
 {
printf("\nP%d",i);
for(j=0;j<p[i].ne;j++)
   {
printf("\t[");
for(k=0;k<pn;k++)
printf("%d",p[i].e[j].tm[k]);
printf("]");
   }
 }
getch();
return 0;
```

}

Output:




```
#include<stdio.h>
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
long *p1(int i,long *comp);
long *p2(int i,long *comp);
long *p3(int i,long *comp);
void main()
{
   long start[]={0,0,0},*vector;
   clrscr();
   while(!kbhit())
   {
      p1(1,&start[0]);
   }
   printf("\n Process Vector\n");
   vector=p1(0,&start[0]);
   printf("p1[%ld%ld%ld]\n",*vector,*(vector+1),*(vector+2));
   vector=p2(0,&start[0]);
   printf("p2[%ld%ld%ld]\n",*vector,*(vector+1),*(vector+2));
   vector=p3(0,&start[0]);
   printf("p3[%ld%ld%ld]\n",*vector,*(vector+1),*(vector+2));
}
```

```
    long *p1(int i,long *comp)
    {
        static long a[]={0,0,0};
        int next;
       if(i==1)
        {
            a[0]++;
            if(*(comp+1)>a[1])
                a[1]=*(comp+1);
            if(*(comp+2)>a[2])
                a[2]=*(comp+2);
            next=random(2);
            if(next==0)
                p2(1,&a[0]);
            else if(next==1)
                p3(1,&a[0]);
            return(&a[0]);
        }
        else
            return(&a[0]);
    }
long *p2(int i,long *comp)
{
    static long b[]={0,0,0};
    int next;
    if(i==1)
    {
        b[i]++;
        if(*comp>b[0])
            b[0]=*(comp);
        if(*(comp+2)>b[2])
            b[2]=*(comp+2);
        next=random(2);
        if(next==0)
            p1(1,&b[0]);
        else if(next==1)
            p3(1,&b[0]);
        return &b[0];
    }
    else
        return &b[0];
}
long *p3(int i,long *comp)
{
    static long c[]={0,0,0};
```

```
    int next;
    if(i==1)
    {
        c[2]++;
        if(*comp>c[0])
            c[0]=*(comp);
        if(*(comp+1)>c[1])
            c[1]=*(comp+1);
        next=random(2);
        if(next==0)
            p1(1,&c[0]);
        return &c[0];
    }
    else
        return &c[0];
}
```

AIM: To study Client Server based program using RPC. THEORY: RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports. RPC makes the client/server model of computing more powerful and easier to program. When combined with the ONC RPCGEN protocol compiler clients transparently make remote calls through a local procedure interface. An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

```
#include <stdio.h>
#include <rpc.h>
#include
<pmapclnt.h>
#include <msg.h>

static void
messageprog_1(); static
char *printmessage_1();

static struct timeval TIMEOUT = {
```

```
25, 0 }; main()
{
   SVCXPRT *transp;

   (void)pmap_unset(MESSAGEPROG,

   MESSAGEVERS);

   transp =
   svcudp_create(RPC_ANYSOCK); if
   (transp == (SVCXPRT *)NULL)
      { (void)fprintf(stderr, "CANNOT CREATE UDP
      SERVICE.\n"); exit(16);
   }
   if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS, messageprog_1, IPPROTO_UDP))
   {
      (void)fprintf(stderr,
      "UNABLE TO REGISTER (MESSAGEPROG, MESSAGEVERS, UDP).\n");
      exit(16);
   }

   transp = svctcp_create(RPC_ANYSOCK,
   0, 0); if (transp == (SVCXPRT *)NULL)
   {
      (void)fprintf(stderr, "CANNOT CREATE TCP
      SERVICE.\n"); exit(16);
   }
   if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS, messageprog_1, IPPROTO_TCP))
   {
      (void)fprintf(stderr,
      "UNABLE TO REGISTER (MESSAGEPROG, MESSAGEVERS, TCP).\n");
      exit(16);
   }
   svc_run();
   (void)fprintf(stderr, "SVC_RUN
   RETURNED\n"); exit(16);
   return(0);
}

static void messageprog_1(rqstp,
transp) struct svc_req *rqstp;
SVCXPRT *transp;
{
   union
   {
      char *printmessage_1_arg;
   }

   argument;
   char
            *r
   esult;
```

```
    bool_t
    (*xdr_argument)();
    bool_t
    (*xdr_result)(); char
    *(*local)();

    switch (rqstp->rq_proc)
    {
       case NULLPROC:
          (void)svc_sendreply(transp, xdr_void, (char
          *)NULL); return;

       case PRINTMESSAGE:
          xdr_argument =
          xdr_wrapstring; xdr_result
          = xdr_int;
local = (char *(*)()) printmessage_1; break;
       default:
          svcerr_noproc(tra
          nsp); return;
    }
    bzero((char *)&argument, sizeof(argument));
    if (!svc_getargs(transp, xdr_argument, &argument))
    {
       svcerr_decode(tra
       nsp); return;
    }
    result = (*local)(&argument,
    rqstp); if (result != (char
    *)NULL &&
       !svc_sendreply(transp, xdr_result, result))
    {
       svcerr_systemerr(transp);
    }
    if (!svc_freeargs(transp, xdr_argument, &argument))
    {
       (void)fprintf(stderr, "UNABLE TO FREE
       ARGUMENTS\n"); exit(16);
    }
    return;
 }

 char
 *printmessage_1(msg)
 char **msg;
 {
    static char result;

    fprintf(stderr, "%s\n",
    *msg); result = 1;
    return(&result);
 }
```

Practical No. 2

Aim: Write a program to simulate the functioning of Lamport's logical clock.
Theory:

The algorithm of Lamport timestamps is a simple algorithm used to determine the order of events in a distributed computer system. As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead, and conceptually provide a starting point for the more advancedvector clock method. They are named after their creator, Leslie Lamport.

Distributed algorithms such as resource synchronization often depend on some method of ordering events to function. For example, consider a system with two processes and a disk. The processes send messages to each other, and also send messages to the disk requesting access. The disk grants access in the order the messages were sent. For example process A sends a message to the disk requesting write access, and then sends a read instruction message to process B. Process B receives the message, and as a result sends its own read request message

to the disk. If there is a timing delay causing the disk to receive both messages at the same time, it can determine which messagehappened-before the other: (happens-before if one can get from to by a sequence of moves of two types: moving forward while remaining in the same process, and following a message from its sending to its reception.) A logical clock algorithm provides a mechanism to determine facts about the order of such events.

Lamport invented a simple mechanism by which the happened-before ordering can be captured numerically. A Lamport logical clock is an incrementing software counter maintained in each process. Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender.

The algorithm follows some simple rules:

1. A process increments its counter before each event in that process;
2. When a process sends a message, it includes its counter value with the message;
3. On receiving a message, the receiver sets its counter greater than or equal to its own counter (already incremented due to rule 1) and greater than the message counter, before it considers the message received.

In a Pseudocode format, the algorithm for sending:

```
time = time+1;
time_stamp = time;
send(Message, time_stamp);
```

The algorithm for receiving a message:

```
(message, time_stamp) = receive();
time = max(time_stamp, time)+1;
```

Program:

```
#include<stdio.h>
#define MAX 20
int d=1;

struct event
{
      int tm;
        int pno;
        int eno;
};
struct process
{
      int ne;
        struct event e[MAX];
}p[MAX];
main()
{
      int i,j,pn,dno;
        int before_proc,after_proc,before_event,after_event;
        int dependency[4][MAX];
```

```
        clrscr();
        printf("\nenter the number of processes");
        scanf("%d",&pn);
        for(i=0;i<pn;i++)
        {
           printf("enter the number of events in process %d ",i);
                scanf("%d",&p[i].ne);
                for(j=0;j<p[i].ne;j++)
                 p[i].e[j].tm=j+d;
        }
        printf("\nscenario:\n");
        for(i=0;i<pn;i++)
        {
           printf("\nP%d",i);
           for(j=0;j<p[i].ne;j++)
                 printf("\te%d%d",i,p[i].e[j].tm-1);
        }
        printf("\nhow many dependencies exist?");
        scanf("%d",&dno);
        printf("\nenter the values for dependency matrix:\nprocess
no\tevent no\t->\tprocess no\tevent no\n");
        for(i=0;i<dno;i++)
           for(j=0;j<4;j++)
                 scanf("%d",&dependency[i][j]);
       /* printf("\ndependency matrix");
        for(i=0;i<dno;i++)
        {
           for(j=0;j<4;j++)
                 printf("\t%d",dependency[i][j]);
                printf("\n");
        }  */
        for(i=0;i<dno;i++)
        {
           before_proc=dependency[i][0];
                before_event=dependency[i][1];
                after_proc=dependency[i][2];
                after_event=dependency[i][3];
                //printf("\n%d",before_proc);
                if(p[after_proc].e[after_event].tm <
(p[before_proc].e[before_event].tm+d))
                {

     p[after_proc].e[after_event].tm=p[before_proc].e[before_event].tm
+d;
                        for(j=after_event+1;j<p[after_proc].ne;j++)
                             p[after_proc].e[j].tm=p[after_proc].e[j-
1].tm+d;
                }
        }
        printf("\nthe values for the timestamps are:");
        for(i=0;i<pn;i++)
        {
           printf("\nP%d",i);
           for(j=0;j<p[i].ne;j++)
                 printf("\t%d",p[i].e[j].tm);
        }
        getch();
        return 0;
```

```
}
```

Output:-

```
enter the number of processes:-3
enter the number of events in process 1:-5
enter the number of events in process 1:-3
enter the number of events in process 1:-5
scenario:

P0      e00     e01     e02     e03     e04
P1      e10     e11     e12
P2      e20     e21     e22     e23     e24
how many dependencies exist?4
enter the values for dependency matrix:
process no      event no        ->      process no      event no
  0              0                       1                1
  1              1                       2                1
  2              4                       1                2
  1              2                       0                4
dependency matrix
        0       0       1       1
        1       1       2       1
        2       4       1       2
        1       2       0       4
The values for the timestamps are:
P0      1       2       3       4       8
P1      1       2       7
P2      1       3       4       5       6
```

Result:

Thus, we have executed a program to implement Lamport clock.

Practical No. 3

Aim: Write a program to simulate the functioning of Vector clock.

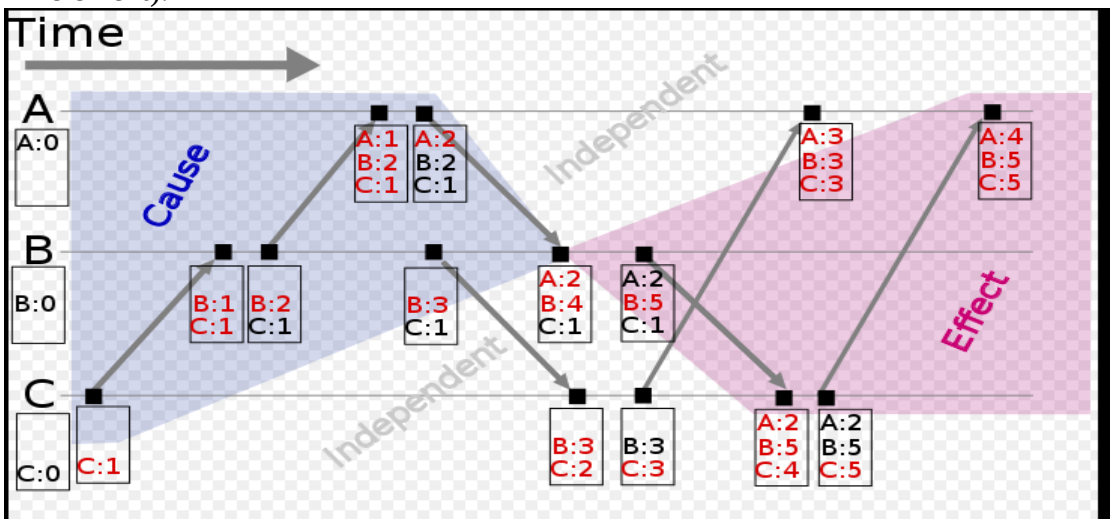
Theory:

A vector clock is an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations. Just as in Lamport timestamps, interprocess messages contain the state of the sending process's logical clock. A vector clock of a system of N processes is an array/vector of N logical clocks, one clock per process; a local "smallest possible values" copy of the global clock-array is kept in each process, with the following rules for clock updates:

Events in the blue region are the causes leading to event B4, whereas those in the red region are the effects of event B4

- Initially all clocks are zero.
- Each time a process experiences an internal event, it increments its own logical clock in the vector by one.

- Each time a process prepares to send a message, it sends its entire vector along with the message being sent.
- Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).



Program:

```
#include<stdio.h>
#define MAX 20
int d=1;

struct event
{
      int tm[MAX];
        //int pno;
        int eno;
};
struct process
{
      int ne;
        struct event e[MAX];
}p[MAX];
main()
{
      int i,j,k,pn,dno;
        int before_proc,after_proc,before_event,after_event;
        int dependency[4][MAX];
        clrscr();
        printf("\nenter the number of processes");
        scanf("%d",&pn);
        for(i=0;i<pn;i++)
        {
            printf("enter the number of events in process %d ",i);
                scanf("%d",&p[i].ne);
                for(j=0;j<p[i].ne;j++)
                {
                        p[i].e[j].eno=j;
                 for(k=0;k<pn;k++)
                            p[i].e[j].tm[k]=0;
```

```
                p[i].e[j].tm[i]=j+d;
               }
        }
       /* printf("\ntime stamps are:");
         for(i=0;i<pn;i++)
        {
           printf("\nP%d",i);
           for(j=0;j<p[i].ne;j++)
                {
                 printf("\t[");
                 for(k=0;k<pn;k++)
                       printf(" %d ",p[i].e[j].tm[k]);
                        printf("]\t");
                }
        } */
        printf("\nscenario:\n");
        for(i=0;i<pn;i++)
        {
           printf("\nP%d",i);
           for(j=0;j<p[i].ne;j++)
                 printf("\te%d%d",i,p[i].e[j].eno);
        }
        printf("\nhow many dependencies exist?");
        scanf("%d",&dno);
        printf("\nenter the values for dependency matrix:\nprocess
no\tevent no\t->\tprocess no\tevent no\n");
        for(i=0;i<dno;i++)
           for(j=0;j<4;j++)
                 scanf("%d",&dependency[i][j]);
       /* printf("\ndependency matrix");
        for(i=0;i<dno;i++)
        {
           for(j=0;j<4;j++)
                 printf("\t%d",dependency[i][j]);
                printf("\n");
        }  */
        for(i=0;i<dno;i++)
        {
           before_proc=dependency[i][0];
                before_event=dependency[i][1];
                after_proc=dependency[i][2];
                after_event=dependency[i][3];
                //printf("\n%d",before_proc);
                for(k=0;k<pn;k++)
                if(p[after_proc].e[after_event].tm[k] <
(p[before_proc].e[before_event].tm[k]))
                {

      p[after_proc].e[after_event].tm[k]=p[before_proc].e[before_event]
.tm[k];
                        for(j=after_event+1;j<p[after_proc].ne;j++)
                             p[after_proc].e[j].tm[k]=p[after_proc].e[j-
1].tm[k];
                }
        }
        printf("\nthe values for the timestamps are:");
        for(i=0;i<pn;i++)
        {
```

```
            printf("\nP%d",i);
            for(j=0;j<p[i].ne;j++)
                 {
                  printf("\t[");
                  for(k=0;k<pn;k++)
                        printf(" %d ",p[i].e[j].tm[k]);
                         printf("]");
                 }
        }
        getch();
        return 0;
}
```

Output:-

```
enter the number of processes:-3
enter the number of events in process 1:-5
enter the number of events in process 1:-3
enter the number of events in process 1:-5

scenario:

P0      e00     e01     e02     e03     e04
P1      e10     e11     e12
P2      e20     e21     e22     e23     e24

how many dependencies exist?4

enter the values for dependency matrix:
process no      event no        ->      process no      event no
  0              0                       1                1
  1              1                       2                1
  2              4                       1                2
  1              2                       0                4

dependency matrix
        0       0       1       1
        1       1       2       1
        2       4       1       2
        1       2       0       4

the values for the timestamps are:
P0      [ 1  0  0 ] [ 2  0  0 ] [ 3  0  0 ] [ 4  0  0 ] [ 8  8  7 ]
P1      [ 0  1  0 ] [ 2  2  0 ] [ 7  7  6 ]
P2      [ 0  0  1 ] [ 3  3  2 ] [ 4  4  3 ] [ 5  5  4 ] [ 6  6  5 ]
```

Result:

Thus, we have executed a program to simulate the functioning of Vector clock.

Practical No. 4

Aim: Write a program to implement Ricart Agrawala mutual exclusion algorithm.

Theory:

Ricart & Agrawala put forth a fully distributed mutual exclusion algorithm in 1981. It requires the following:

- total ordering of all events in a system (e.g. Lamport's timestamp algorithm with unique timestamps)
- messages are reliable (the delivery of every message is acknowledged).

When a process wants to get a lock for a resource, it:

1. Composes a message containing *{ message identifier(machine ID, process ID), name of the resource, timestamp)*.
2. Sends a *request* message to all other processes in the group (using reliable messaging).
3. Wait until *everyone* in the group has given permission.
4. Access the resource; it now has the lock.

When a process receives a request message, it may be in one of three states:

Case 1
: The receiver is not interested in the resource, send a reply (*OK*) to the sender.

Case 2
: The receiver is in the resource. Do not reply but add the request to a local queue of requests.

Case 3
: The receiver also wants the lock on the resource and has sent its request. In this case, the receiver compares the timestamp in the received message with the one in the message that it has sent out. The earliest timestamp wins. If the receiver is the loser, it sends a reply (*OK*) to sender. If the receiver has the earlier timestamp, then it is the winner and does not reply (and will get an *OK* from the other process). Instead, it adds the request to its queue and will send the *OK* only when it is done with its use of the resource.

Algorithm

Requesting Site

- Sends a message to all sites. This message includes the site's name, and the current timestamp of the system according to its logical clock (*which is assumed to be synchronized with the other sites*)

Receiving Site

- Upon reception of a request message, immediately sending a timestamped *reply* message if and only if:
    - the receiving process is not currently interested in the critical section OR
    - the receiving process has a lower priority (*usually this means having a later timestamp)*
- Otherwise, the receiving process will defer the reply message. This means that a reply will be sent only after the receiving process has finished using the critical section itself.

Critical Section:

- Requesting site enters its critical section only after receiving all reply messages.
- Upon exiting the critical section, the site sends all deferred reply messages.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
struct process{
 int timestamp,cs;
}p[20];
void main(){
int  i,j,pn,cs_no=0,small,index;
clrscr();
printf("\nNo of processes : ");
scanf("%d",&pn);
for(i=0;i<pn;i++)
      {
       printf("\nEnter timestamp for Process %d : ",i+1);
       scanf("%d",&p[i].timestamp);
       printf("\nProcess %d want to participate (enter 1 for yes or 0
for no): ",i+1);
       scanf("%d",&p[i].cs);
      }
for(i=0;i<pn;i++)
      {
       if(p[i].cs == 1)
            cs_no++;
      }
      printf("\nNo of processes participating cs : %d",cs_no);
for(i=0;i<cs_no;i++)
      {
       small=999;
       for(j=0;j<pn;j++)
            {
              if(p[j].timestamp<small && p[j].cs!=0)
                  {
                   small=p[j].timestamp;
                   index=j;
                  }
            }
             //  printf("\nSmall : %d",small);
            p[index].cs=0;
            sleep(2);
            printf("\n\nProcess %d is executing cs ... Timestamp =
%d",index+1,p[index].timestamp);
            sleep(2);
            printf("\nProcess %d is exiting from cs...",index+1);
      }
      getch();
}
```

Output:-

```
No of processes : 3
Enter timestamp for Process 1 : 5
Process 1 want to participate (enter 1 for yes or 0 for no): 1
Enter timestamp for Process 2 : 3
Process 2 want to participate (enter 1 for yes or 0 for no): 1
Enter timestamp for Process 3 : 4
Process 3 want to participate (enter 1 for yes or 0 for no): 1
No of processes participating cs : 3
Process 2 is executing cs ... Timestamp = 3
```

```
Process 2 is exiting from cs...
Process 3 is executing cs ... Timestamp = 4
Process 3 is exiting from cs...
Process 1 is executing cs ... Timestamp = 5
Process 1 is exiting from cs...
```

Result:

Thus, we have executed a program to implement Ricart Agrawala algorithm.

Practical No. 5

Aim:- Write a program to implement Lamport Non-Token based algorithm.

Theory:-

Lamport was the first to give a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme. Let $\mathbf{R_i}$ be the ***request set*** of site *Si* , i.e. the set of sites from which S_i needs permission when it wants to enter CS. In Lamport's algorithm, $\forall i : 1 \leq i \leq N$ :: **Ri**= {*S1, S2,...,SN*}. Every site *Si* keeps a queue, ***request_queuei***, which contains mutual exclusion requests ordered by their timestamps. This algorithm requires messages to be delivered in the FIFO order between every pair of sites.

The Algorithm

Requesting the critical section.

1. When a site *Si* wants to enter the CS, it sends REQUEST(*tsi*, *i*) message to all the sites in its request set **Ri** and places the request on *request_queuei* (*tsi* is the timestamp of the request).

2. When a site *Sj* receives the REQUEST(*tsi*, *i*) message from site *Si*, it returns a timestamped REPLY message to *Si* and places site *S'is* request on r*equest_queuej*.

Executing the critical section.

1. Site *Si* enters the CS when the two following conditions hold**:**

a) [L1:] *Si* has received a message with timestamp larger than *(tsi, i*) from all other sites.

b) [L2:] *S'is* request is at the top *request_queuei.*

Releasing the critical section.

1. Site *Si*, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.
2. When a site *Sj* receives a RELEASE message from site *Si*, it removes *S'i*s request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter CS. The algorithm executes CS requests in the increasing order of timestamps.

Program:-

```
#include<stdio.h>
//#include<conio.h>
struct process
{
struct message
{
  int tstamp;
  int pid;
}m[20];
int nreply;
};
int main()
{
  int totn,n,i,j,k,tempid,temptstamp;
  int a,b;
  struct process p[50];

  printf("enter the total no of processes\n");
  scanf("%d",&totn);
  printf("enter the no of processes competing to enter critical section\n");
  scanf("%d",&n);
  j=0;
  for(i=0;i<n;i++)
  {
    printf("enter the process id (a no less than %d)\n",totn);
    scanf("%d",&tempid);
    printf("enter the timestamp of that process\n");
    scanf("%d",&temptstamp);
```

```
  for(k=0;k<totn;k++)
  {
    p[k].m[j].pid=tempid;
    p[k].m[j].tstamp=temptstamp;
    p[k].nreply=totn;
  }
  j++;
 }
 for(k=0;k<totn;k++)
 {
 for(i=0;i<n-1;i++)
 {
  for(j=i+1;j<n;j++)
  {
    if(p[k].m[i].tstamp>p[k].m[j].tstamp)
    {
          tempid=p[k].m[i].pid;
          temptstamp=p[k].m[i].tstamp;
          p[k].m[i].pid=p[k].m[j].pid;
          p[k].m[i].tstamp=p[k].m[j].tstamp;
          p[k].m[j].pid=tempid;
          p[k].m[j].tstamp=temptstamp;
    }
  }
 }
 }
 for(i=0;i<totn;i++)
 {
  printf("process %d with %d REPLY messages  ",i,p[i].nreply-1);
  for(j=0;j<n;j++)
  {
   printf("(%d,%d)<--",p[i].m[j].pid,p[i].m[j].tstamp);
  }
  printf("\n");
 }
 while(n>0)
 {
 for(i=0;i<totn;i++)
 {
  if(p[i].m[0].pid==i)
  {
   printf("Since process %d is at the top of the request queue ,it enters the critical
section\n",i);
    printf("\nUpon exiting the critical section ");
    printf("process %d sends a RELEASE message to all the other processes\n",i);
    for(a=0;a<totn;a++)
    {
```

```
    for(j=0;j<n-1;j++)
    {
          p[a].m[j].pid=p[a].m[j+1].pid;
          p[a].m[j].tstamp=p[a].m[j+1].tstamp;

    }
    }
    n--;
    if(n>0)
    {
    printf("Now the request queue status in each process is\n");
    for(a=0;a<totn;a++)
    {
          printf("process %d with %d REPLY messages",a,p[a].nreply);
          for(b=0;b<n;b++)
          {
            printf("(%d,%d)<--",p[a].m[b].pid,p[a].m[b].tstamp);
          }
          printf("\n");
    }
    }
    else
     printf("Now the request queue is empty\n");
   }
  }
  }
 // getch();
}
```

Output:-

Enter the total no of processes 3
Enter the no of processes competing to enter critical section 3
Enter the process id (a no less than 3) 2
Enter the timestamp of that process 3
Enter the process id (a no less than 3) 1
Enter the timestamp of that process 4
Enter the process id (a no less than 3) 3
Enter the timestamp of that process 4

process 0 with 2 REPLY messages (2,3)<--(1,4)<--(3,4)<--
process 1 with 2 REPLY messages (2,3)<--(1,4)<--(3,4)<--
process 2 with 2 REPLY messages (2,3)<--(1,4)<--(3,4)<--
Since process 2 is at the top of the request queue ,it enters the critical section

Upon exiting the critical section process 2 sends a RELEASE message to all the other processes
Now the request queue status in each process is
process 0 with 3 REPLY messages(1,4)<--(3,4)<--
process 1 with 3 REPLY messages(1,4)<--(3,4)<--

process 2 with 3 REPLY messages(1,4)<--(3,4)<--
Since process 1 is at the top of the request queue ,it enters the critical section

Upon exiting the critical section process 1 sends a RELEASE message to all the other processes
Now the request queue status in each process is
process 0 with 3 REPLY messages(3,4)<--
process 1 with 3 REPLY messages(3,4)<--
process 2 with 3 REPLY messages(3,4)<--

Since process 3 is at the top of the request queue ,it enters the critical section

Upon exiting the critical section process 3 sends a RELEASE message to all the other processes
Now the request queue is Empty

Result:

Thus, we have executed a program to implement Lamport Non-Tokan Based algorithm.

Practical No. 6

Aim:- Write a program to implement ChandyMisraHaas mutual exclusion algorithm.

Practical No. 9

Aim:- Write a program to implement Java RMI.

Theory:-

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. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP).
2. In order to support code running in a non-JVM context, a CORBA version was later developed.

Usage of the term RMI may denote solely the programming interface or may signify both the API and JRMP, IIOP, or another implementation, whereas the term RMI-IIOP (read: RMI over IIOP) specifically denotes the RMI interface delegating most of the functionality to the supporting CORBA implementation.

The basic idea of Java RMI, the distributed garbage-collection (DGC) protocol, and much of the architecture underlying the original Sun implementation, come from the 'network objects'

feature of Modula-3. The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

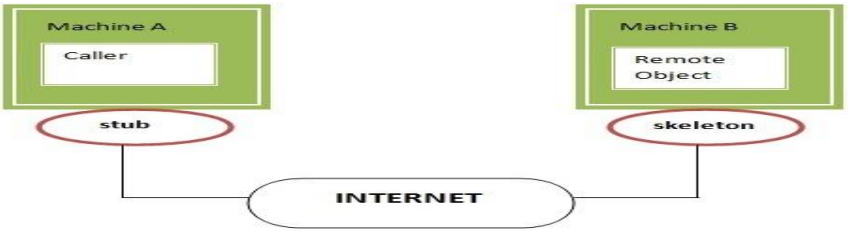
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.



Program:-

```
import java.io.*;
import java.net.*;

public class Client implements Runnable{
   static Socket clientSocket = null;
   static PrintStream os = null;
   static DataInputStream is = null;
   static BufferedReader inputLine = null;
   static boolean closed = false;


   public static void main(String[] args)
   {
    int port_number=1111;
    String host="localhost";

   try {
```

```
          clientSocket = new Socket(host, port_number);
          inputLine = new BufferedReader(new InputStreamReader(System.in));
          os = new PrintStream(clientSocket.getOutputStream());
          is = new DataInputStream(clientSocket.getInputStream());
          } catch (Exception e) {
          System.out.println("Exception occurred : "+e.getMessage());
             }

       if (clientSocket != null && os != null && is != null)
       {
         try
            {
                new Thread(new Client()).start();

                 while (!closed)
                {
                  os.println(inputLine.readLine());
                }

        os.close();
        is.close();
        clientSocket.close();

            } catch (IOException e)
               {
              System.err.println("IOException:  " + e);
               }
      }
  }

    public void run()
    {
    String responseLine;
    try
            {
       while ((responseLine = is.readLine()) != null)
                  {
                System.out.println("\n"+responseLine);
if (responseLine.equalsIgnoreCase("GLOBAL_COMMIT")==true ||
responseLine.equalsIgnoreCase("GLOBAL_ABORT")==true )
                           {
                             break;
                           }
          }
                closed=true;
    } catch (IOException e)
                  {
           System.err.println("IOException:  " + e);
          }
    }
}

import java.io.*;
import java.net.*;
import java.util.*;
```

```
public class Server
 {
  boolean closed=false,inputFromAll=false;
  List<clientThread> t;
  List<String> data;

 Server()
 {
   t= new ArrayList<clientThread>();
   data= new ArrayList<String>();
 }

  public static void main(String args[])
  {
     Socket clientSocket = null;
     ServerSocket serverSocket = null;
     int port_number=1111;

     Server ser=new Server();

     try {
       serverSocket = new ServerSocket(port_number);
       }
       catch (IOException e)
       {System.out.println(e);}

   while(!ser.closed)
            {
       try {
       clientSocket = serverSocket.accept();
                        clientThread th=new clientThread(ser,clientSocket);
                        (ser.t).add(th);
       System.out.println("\nNow Total clients are : "+(ser.t).size());
       (ser.data).add("NOT_SENT");
       th.start();
            }
            catch (IOException e) {}
   }

       try
      {
        serverSocket.close();
      }catch(Exception e1){}

}
}

 class clientThread extends Thread
 {

 DataInputStream is = null;
```

```
String line;
String destClient="";
String name;
PrintStream os = null;
Socket clientSocket = null;
String clientIdentity;
Server ser;

  public clientThread(Server ser,Socket clientSocket)
  {
    this.clientSocket=clientSocket;
    this.ser=ser;
  }


  public void run()
  {
  try
          {
       is = new DataInputStream(clientSocket.getInputStream());
       os = new PrintStream(clientSocket.getOutputStream());
       os.println("Enter your name.");
       name = is.readLine();
              clientIdentity=name;
os.println("Welcome "+name+" to this 2 Phase Application.\nYou will receive a vote Request now...");
os.println("VOTE_REQUEST\nPlease enter COMMIT or ABORT to proceed : ");
       for(int i=0; i<(ser.t).size(); i++)
       {
          if((ser.t).get(i)!=this)
                      {
                        ((ser.t).get(i)).os.println("---A new user "+name+" entered the
Appilcation---");
                       }
               }

            while (true)
            {
               line = is.readLine();

               if(line.equalsIgnoreCase("ABORT"))
                {
                     System.out.println("\nFrom '"+clientIdentity+"' : ABORT\n\nSince
                    aborted we will not wait for inputs from other clients.");
                     System.out.println("\nAborted....");
            for(int i=0; i<(ser.t).size(); i++)
            {
                      ((ser.t).get(i)).os.println("GLOBAL_ABORT");
                      ((ser.t).get(i)).os.close();
                      ((ser.t).get(i)).is.close();
                     }
```

```
                break;
            }


          if(line.equalsIgnoreCase("COMMIT"))
         {
           System.out.println("\nFrom '"+clientIdentity+"' : COMMIT");
           if((ser.t).contains(this))
            {
              (ser.data).set((ser.t).indexOf(this), "COMMIT");
               for(int j=0;j<(ser.data).size();j++)
               {
                    if(!(((ser.data).get(j)).equalsIgnoreCase("NOT_SENT")))
                      {
                        ser.inputFromAll=true;
                        continue;
      }
        else
      {
        ser.inputFromAll=false;
                        System.out.println("\nWaiting for inputs from other clients.");
        break;
      }
       }

                if(ser.inputFromAll)
                {
                  System.out.println("\n\nCommited....");

                  for(int i=0; i<(ser.t).size(); i++)
          {
                      ((ser.t).get(i)).os.println("GLOBAL_COMMIT");
                      ((ser.t).get(i)).os.close();
                      ((ser.t).get(i)).is.close();
                    }
                  break;
                }

                }//if t.contains
              }//commit

            }//while
          ser.closed=true;
         clientSocket.close();

    }   catch(IOException e){};
  }
}

package daj;
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import daj.algorithms.*;

class AlgList {

 AlgList(int max, int def) {
  MAXNODES = max;
  DEFAULTNODES = def;
 }

 // Accessor for algorithm names.
 String getAlgName(int index)
 {
  return titles[index];
 }

 // Get number of nodes if entered interactively
 int getNodes(int parm) {
  if (node != 0) return node;
  else return parm;
 }

 // Sentinel search for the algorithm code
 //   or obtain algorithm from GUI is no code entered.
 int getAlg(String s) {
  if (s.equals("")) {
     Interactive frame = new Interactive();
     frame.setup();
     waitObject.waitOK();  // Wait for reply from frame event handler.
     frame.dispose();
     return select;
  }
  int index = algs.length-1;
  algs[0] = s;
  while (! s.equals(algs[index])) index--;
  if (index == 0) badParam();
  return index;
 }

 // Display message and exit if bad algorithm code.
 private void badParam() {
  System.err.println("Usage: java daj alg [num] or " +
    "java -jar daj.jar alg [num], where");
  System.out.println("  num is the number of nodes 2.." +
    MAXNODES + " default " + DEFAULTNODES);
  System.out.println("  alg is the algorithm:");
  for (int i = 1; i < algs.length; i++)
    System.out.println("    "+algs[i]+" - "+titles[i]);
```

```
}

// Inner class to create JFrame for GUI

class Interactive extends JFrame {

void setup() {
   setTitle(Screen.VERSION + Screen.VERSIONNUMBER);
   if (Screen.isApplication) setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setSize(450,130);
   setLocationRelativeTo(null);
   setLayout(new BorderLayout());

   final JComboBox algBox = new JComboBox();
   algBox.setEditable(false);
   for (int i = 1; i < algs.length; i++)
      algBox.addItem(titles[i]);
   algBox.setSelectedIndex(DEFAULTALG);
   algBox.setBorder(BorderFactory.createEtchedBorder());
   algBox.setMaximumRowCount(MAXROWS);
   add(algBox, BorderLayout.CENTER);

   final JComboBox numBox = new JComboBox();
   numBox.setEditable(false);
   for (int i = 1; i <= MAXNODES; i++)
      numBox.addItem("  " + i + "  ");
   numBox.setSelectedIndex(DEFAULTNODES-1);
   numBox.setBorder(BorderFactory.createEtchedBorder());
   add(numBox, BorderLayout.EAST);

   // Label for title of panel.
   JLabel label = new JLabel(CHOOSE, JLabel.CENTER);
   label.setBorder(BorderFactory.createEtchedBorder());
   add(label, BorderLayout.NORTH);

   // Panel for buttons.
   JPanel buttonPanel = new JPanel();
   JButton OKButton = new JButton(START);
   JButton exitButton = new JButton(EXIT);
   JButton aboutButton = new JButton(ABOUT);
   OKButton.setMnemonic(START.charAt(1));
   exitButton.setMnemonic(EXIT.charAt(1));
   aboutButton.setMnemonic(ABOUT.charAt(0));
   buttonPanel.add(OKButton);
   buttonPanel.add(exitButton);
   buttonPanel.add(aboutButton);
   buttonPanel.setBorder(BorderFactory.createEtchedBorder());
   add(buttonPanel, BorderLayout.SOUTH);

   KeyListener kl = new KeyListener() {
      //public boolean isFocusable() { return true; }
```

```
        public void keyReleased(KeyEvent evt) {}
        public void keyPressed(KeyEvent evt) {}
        public void keyTyped(KeyEvent evt) {
          char c = evt.getKeyChar();
          if ((int) c == KeyEvent.VK_ENTER) {
              select = algBox.getSelectedIndex() + 1;
              node = numBox.getSelectedIndex() + 1;
              waitObject.signalOK();
          }
          else if ((int) c == KeyEvent.VK_ESCAPE)
             System.exit(0);
        }
    };
    algBox.addKeyListener(kl);
    numBox.addKeyListener(kl);

    // Listener for buttons.
    ActionListener bl = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          String cmd = e.getActionCommand();
          if (cmd.equals(START)) {
              select = algBox.getSelectedIndex() + 1;
              node = numBox.getSelectedIndex() + 1;
              waitObject.signalOK();
          }
          else if (cmd.equals(EXIT))
              System.exit(0);
          else if (cmd.equals(ABOUT)) {
              String aboutText =
"     DAJ - Distributed Algorithms in Java. Version " +
Screen.VERSIONNUMBER + ".\n\n" +
" Developed by Moti Ben-Ari.\n"+
" Additional programming by:\n"+
"   Antoine Pineau - University of Joensuu.\n"+
"   Basil Worrall, Frederick Kemp, Frank Harvie,\n"+
"   Richard McGladdery, Leoni Lubbinge, Derick Burger, Darrell Newing\n"+
"        - University of Pretoria.\n"+
"   Maor Zamsky - Givat Brenner High School.\n\n"+
"   Otto Seppälä, Ville Karavirta - Helsinki University of Technology\n"+
"Copyright 2003-6 by Mordechai (Moti) Ben-Ari.\n\n"+
" This program is free software; you can redistribute it and/or\n" +
" modify it under the terms of the GNU General Public License\n" +
" as published by the Free Software Foundation; either version 2\n" +
" of the License, or (at your option) any later version.\n\n" +
" This program is distributed in the hope that it will be useful\n" +
" but WITHOUT ANY WARRANTY; without even the implied warranty of\n" +
" MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.\n" +
" See the GNU General Public License for more details.\n\n" +
" You should have received a copy of the GNU General Public License\n" +
" along with this program; if not, write to the Free Software\n" +
" Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA \n" +
```

```
" 02111-1307, USA.";
            JOptionPane.showMessageDialog(
               null, aboutText, Screen.VERSION + Screen.VERSIONNUMBER,
               JOptionPane.INFORMATION_MESSAGE);
            }
          } // actionPerformed
    };  // anonymous class

    OKButton.addActionListener(bl);
    exitButton.addActionListener(bl);
    aboutButton.addActionListener(bl);
    validate();
    setVisible(true);
  } // setup
} // class Interactive

// Titles and codes of the algorithms.
// Ville & Otto: Added the new algorithms to titles and algs arrays.
private static final String[] titles = {
  "",
  "Berman-Garay (King) algorithm for consensus",
  "Byzantine generals algorithm for consensus (byzantine failures)",
  "Byzantine generals algorithm for consensus (crash failures)",
  "Chandy-Lamport algorithm for global snapshots",
  "Dijkstra-Scholten termination detection",
  "Flooding algorithm for consensus",
  "Lamport algorithm for mutual exclusion",
  "Ricart-Agrawala algorithm for mutual exclusion",
  "Maekawa mutual algorithm for exclusion",
  "Suzuki-Kasami algorithm for mutual exclusion",
  "Huang credit-recovery termination detection",
  "Mattern credit-recovery termination detection",
  "Carvalho-Roucairol token algorithm for mutual exclusion",
  "Neilsen-Mizuno algorithm for mutual exclusion"
};
private static String[] algs =
  {"help", "kg", "bg", "cr", "sn", "ds", "fl", "la", "ra", "ma", "sk", "hg","mt", "tk", "nm"};

// Create and return object for an algorithm.
DistAlg getAlgObject(int index, int i, DistAlg[] da, Screen s) {
  switch (index) {
     case  1: return new KG(i, da, s);
     case  2: return new BG(i, da, s);
     case  3: return new CR(i, da, s);
     case  4: return new SN(i, da, s);
     case  5: return new DS(i, da, s);
     case  6: return new FL(i, da, s);
     case  7: return new LA(i, da, s);
     case  8: return new RA(i, da, s);
     case  9: return new MA(i, da, s);
     case 10: return new SK(i, da, s);
```

```
      // Otto: Replaced the old implementation of the Huang
      case 11: return new HG(i, da, s);
      // Otto & Ville: Added these new algorithms
      case 12: return new MT(i, da, s);
      case 13: return new TK(i, da, s);
      case 14: return new NM(i, da, s);
    }
   return null; // Dummy statement for compiler
  }

  private int     select;         // Algorithm selected by GUI
  private int     node = 0;       // Number of nodes selected by GUI
  private WaitObject waitObject = new WaitObject();
                          // For synchronization with event handlers
  private static int MAXNODES;     // Maximum number of nodes
  private static int DEFAULTNODES; // Default number of nodes
  private static int DEFAULTALG = 1; // Default alg is BG
  private static final int MAXROWS = 15; // Maximum algs without scrolling
  private static final String      // GUI strings
    START = "Start",
    ABOUT = "About",
    EXIT  = "Exit",
    CHOOSE = "Choose an algorithm and the number of nodes";
}
```

Result:-

Thus, JAVA RMA for Distributed System has been studied & Executed Successfully.

Practical No. 10

Aim:- Case study: Hadoop File System.

Theory:-

The Hadoop Distributed File System (HDFS)—a subproject of the Apache Hadoop project—is a distributed, highly fault-tolerant file system designed to run on low-cost commodity hardware. HDFS provides high-throughput access to application data and is suitable for applications with large data sets. This article explores the primary features of HDFS and provides a high-level view of the HDFS architecture.

HDFS is an Apache Software Foundation project and a subproject of the Apache Hadoop project (see Resources). Hadoop is ideal for storing large amounts of data, like terabytes and petabytes, and uses HDFS as its storage system. HDFS lets you connect *nodes* (commodity personal computers) contained within clusters over which data files are distributed. You can then access and store the data files as one seamless file system. Access to data files is handled in a *streaming* manner, meaning that applications or commands are executed directly using the MapReduce processing model (again, see Resources).

HDFS is fault tolerant and provides high-throughput access to large data sets. This article explores the primary features of HDFS and provides a high-level view of the HDFS architecture.

Overview of HDFS

HDFS has many similarities with other distributed file systems, but is different in several respects. One noticeable difference is HDFS's write-once-read-many model that relaxes concurrency control requirements, simplifies data coherency, and enables high-throughput access.

Another unique attribute of HDFS is the viewpoint that it is usually better to locate processing logic near the data rather than moving the data to the application space.

HDFS rigorously restricts data writing to one writer at a time. Bytes are always appended to the end of a stream, and byte streams are guaranteed to be stored in the order written.

HDFS has many goals. Here are some of the most notable:

- Fault tolerance by detecting faults and applying quick, automatic recovery
- Data access via MapReduce streaming
- Simple and robust coherency model
- Processing logic close to the data, rather than the data close to the processing logic
- Portability across heterogeneous commodity hardware and operating systems
- Scalability to reliably store and process large amounts of data
- Economy by distributing data and processing across clusters of commodity personal computers
- Efficiency by distributing data and logic to process it in parallel on nodes where data is located
- Reliability by automatically maintaining multiple copies of data and automatically redeploying processing logic in the event of failures

HDFS architecture

HDFS is comprised of interconnected clusters of nodes where files and directories reside. An HDFS cluster consists of a single node, known as a `NameNode`, that manages the file system namespace and regulates client access to files. In addition, data nodes (`DataNodes`) store data as blocks within files.

Name nodes and data nodes

Within HDFS, a given name node manages file system namespace operations like opening, closing, and renaming files and directories. A name node also maps data blocks to data nodes, which handle read and write requests from HDFS clients. Data nodes also create, delete, and replicate data blocks according to instructions from the governing name node. Figure 1 illustrates the high-level architecture of HDFS.

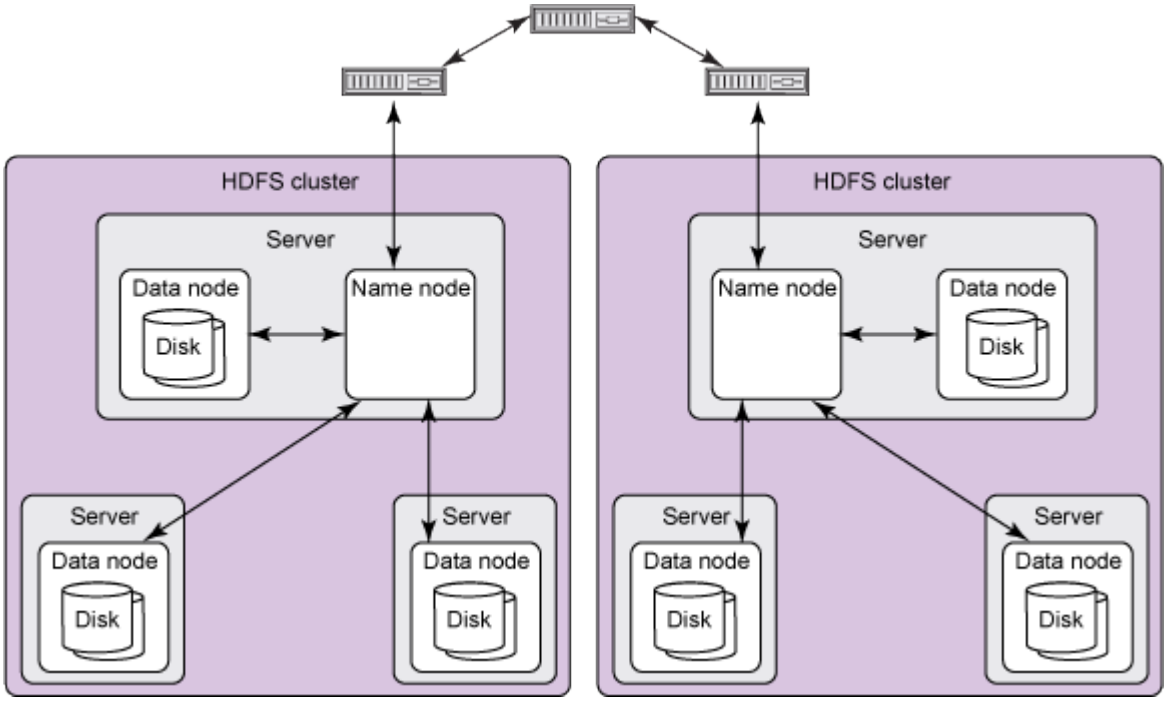


Figure 1. The HDFS architecture

As Figure 1 illustrates, each cluster contains one name node. This design facilitates a simplified model for managing each namespace and arbitrating data distribution.

Relationships between name nodes and data nodes

Name nodes and data nodes are software components designed to run in a decoupled manner on commodity machines across heterogeneous operating systems. HDFS is built using the Java programming language; therefore, any machine that supports the Java programming language can run HDFS. A typical installation cluster has a dedicated machine that runs a name node and possibly one data node. Each of the other machines in the cluster runs one data node.

Data nodes continuously loop, asking the name node for instructions. A name node can't connect directly to a data node; it simply returns values from functions invoked by a data node. Each data node maintains an open server socket so that client code or other data nodes can read or write data. The host or port for this server socket is known by the name node, which provides the information to interested clients or other data nodes. See the Communications protocols sidebar for more about communication between data nodes, name nodes, and clients.

The name node maintains and administers changes to the file system namespace.

File system namespace

HDFS supports a traditional hierarchical file organization in which a user or an application can create directories and store files inside them. The file system namespace hierarchy is similar to most other existing file systems; you can create, rename, relocate, and remove files.

Data replication

HDFS replicates file blocks for fault tolerance. An application can specify the number of replicas of a file at the time it is created, and this number can be changed any time after that. The name node makes all decisions concerning block replication.

HDFS uses an intelligent replica placement model for reliability and performance. Optimizing replica placement makes HDFS unique from most other distributed file systems, and is facilitated by a rack-aware replica placement policy that uses network bandwidth efficiently.

Large HDFS environments typically operate across multiple installations of computers. Communication between two data nodes in different installations is typically slower than data nodes within the same installation. Therefore, the name node attempts to optimize communications between data nodes. The name node identifies the location of data nodes by their rack IDs.

Data storage reliability

One important objective of HDFS is to store data reliably, even when failures occur within name nodes, data nodes, or network partitions.

Detection is the first step HDFS takes to overcome failures. HDFS uses heartbeat messages to detect connectivity between name and data nodes.

HDFS heartbeats

Several things can cause loss of connectivity between name and data nodes. Therefore, each data node sends periodic heartbeat messages to its name node, so the latter can detect loss of connectivity if it stops receiving them. The name node marks as dead data nodes not responding to heartbeats and refrains from sending further requests to them. Data stored on a dead node is no longer available to an HDFS client from that node, which is effectively removed from the system. If the death of a node causes the replication factor of data blocks to drop below their minimum value, the name node initiates additional replication to bring the replication factor back to a normalized state. Figure 2 illustrates the HDFS process of sending heartbeat messages.

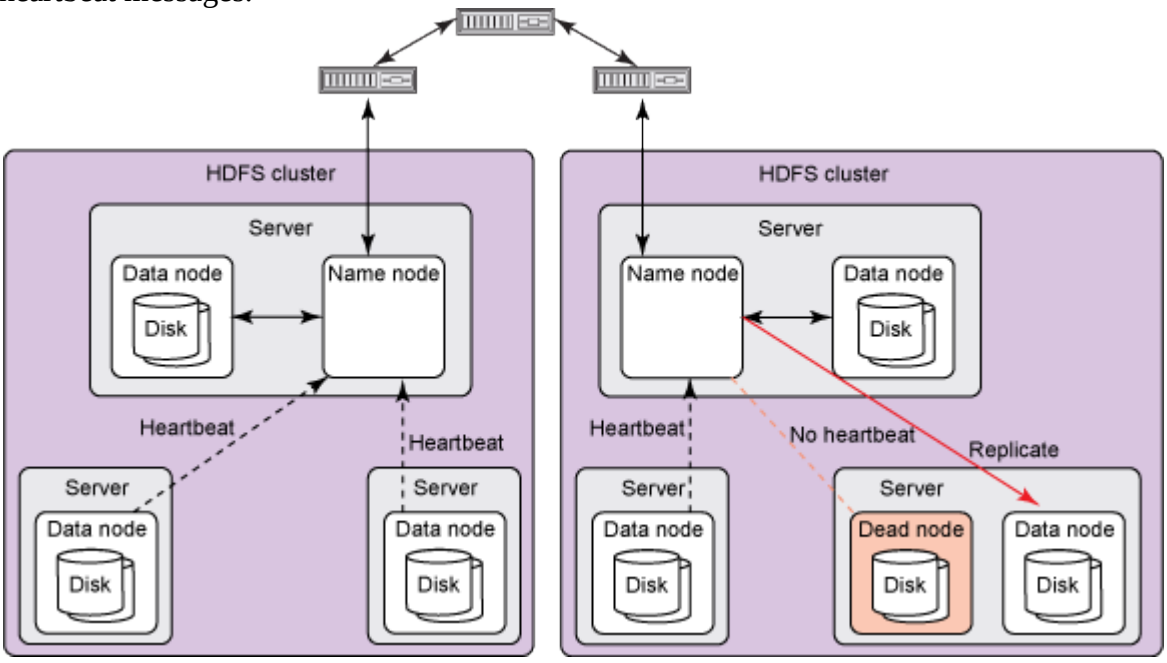


Figure 2. The HDFS heartbeat process

Data block rebalancing

HDFS data blocks might not always be placed uniformly across data nodes, meaning that the used space for one or more data nodes can be underutilized. Therefore, HDFS supports

rebalancing data blocks using various models. One model might move data blocks from one data node to another automatically if the free space on a data node falls too low. Another model might dynamically create additional replicas and rebalance other data blocks in a cluster if a sudden increase in demand for a given file occurs. HDFS also provides the `hadoop balance` command for manual rebalancing tasks.

One common reason to rebalance is the addition of new data nodes to a cluster. When placing new blocks, name nodes consider various parameters before choosing the data nodes to receive them. Some of the considerations are:

- Block-replica writing policies
- Prevention of data loss due to installation or rack failure
- Reduction of cross-installation network I/O
- Uniform data spread across data nodes in a cluster

The cluster-rebalancing feature of HDFS is just one mechanism it uses to sustain the integrity of its data. Other mechanisms are discussed next.

Data integrity

HDFS goes to great lengths to ensure the integrity of data across clusters. It uses checksum validation on the contents of HDFS files by storing computed checksums in separate, hidden files in the same namespace as the actual data. When a client retrieves file data, it can verify that the data received matches the checksum stored in the associated file.

The HDFS namespace is stored using a transaction log kept by each name node. The file system namespace, along with file block mappings and file system properties, is stored in a file called FsImage. When a name node is initialized, it reads the FsImage file along with other files, and applies the transactions and state information found in these files.

Synchronous metadata updating

A name node uses a log file known as the EditLog to persistently record every transaction that occurs to HDFS file system metadata. If the EditLog or FsImage files become corrupted, the HDFS instance to which they belong ceases to function. Therefore, a name node supports multiple copies of the FsImage and EditLog files. With multiple copies of these files in place, any change to either file propagates synchronously to all of the copies. When a name node restarts, it uses the latest consistent version of FsImage and EditLog to initialize itself.

HDFS permissions for users, files, and directories

HDFS implements a permissions model for files and directories that has a lot in common with the Portable Operating System Interface (POSIX) model; for example, every file and directory is associated with an owner and a group. The HDFS permissions model supports read (r), write (w), and execute (x). Because there is no concept of file execution within HDFS, the x permission takes on a different meaning. Simply put, the x attribute indicates permission for accessing a child directory of a given parent directory. The owner of a file or directory is the identity of the client process that created it. The group is the group of the parent directory.

Summary

Hadoop is an Apache Software Foundation distributed file system and data management project with goals for storing and managing large amounts of data. Hadoop uses a storage system called HDFS to connect commodity personal computers, known as nodes, contained within clusters over which data blocks are distributed. You can access and store the data blocks as one seamless file system using the MapReduce processing model.

HDFS shares many common features with other distributed file systems while supporting some important differences. One significant difference is HDFS's write-once-read-many model that relaxes concurrency control requirements, simplifies data coherency, and enables high-throughput access.

In order to provide an optimized data-access model, HDFS is designed to locate processing logic near the data rather than locating data near the application space.

Result:-

Thus, Hadoop distributed file system Hs been Studied.