DELHI TECHNOLOGICAL UNIVERSITY

DEPT. OF COMPUTER SCIENCE

DISTRIBUTED SYSTEMS LAB (CO - 407)

Submitted To:-

Ms. Shivani Pandey

Submitted By:-

Umang Ahuja

A4 - G3

(2K16/CO/337)

4th Year

Index

S. No.	Name of Practical	Dated	Signature
1.			
2.			
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			

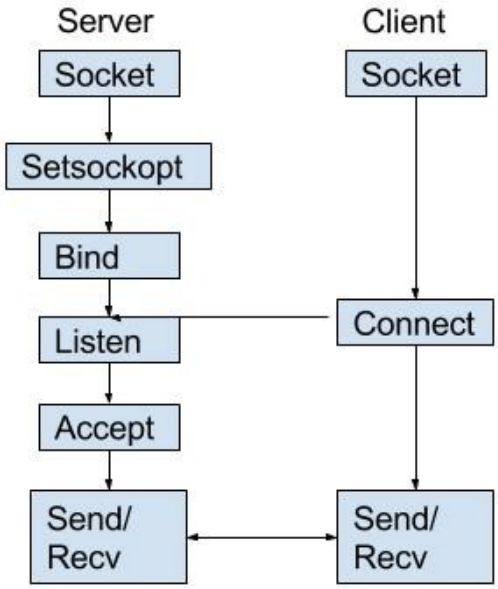
Experiment - 1

Aim

To implement concurrent day-time client-server application.

Description

A client/server application is a piece of software that runs on a client computer and makes requests to a remote server. The client-server model is a distributed communication framework of network processes among service requestors, clients and service providers. The client-server connection is established through a network or the Internet.



Code

Server.c

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
#include <time.h>
```

```
int main()
{
   time_t clock;
   char dataSending[1025];
   int clintListn = 0, clintConnt = 0;
   struct sockaddr_in ipOfServer;
   clintListn = socket(AF_INET, SOCK_STREAM, 0);
   memset(&ipOfServer, '0', sizeof(ipOfServer));
   memset(dataSending, '0', sizeof(dataSending));
   ipOfServer.sin_family = AF_INET;
   ipOfServer.sin_addr.s_addr = htonl(INADDR_ANY);
   ipOfServer.sin_port = htons(2017); //  port number of running server
   bind(clintListn, (struct sockaddr *)&ipOfServer, sizeof(ipOfServer));
   listen(clintListn, 20);

   while (1)
   {
      clintConnt = accept(clintListn, (struct sockaddr *)NULL, NULL);
      printf("New client connected\n");

      clock = time(NULL);
      snprintf(dataSending, sizeof(dataSending), "%.24s\r\n", ctime(&clock));
      write(clintConnt, dataSending, strlen(dataSending));
      printf("Current time sent to client\n");
      close(clintConnt);
      sleep(1);
   }

   return 0;
}
```

Client.c

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
```

```
#include <arpa/inet.h>

int main()
{
   int CreateSocket = 0, n = 0;
   char dataReceived[1024];
   struct sockaddr_in ipOfServer;

   memset(dataReceived, '0', sizeof(dataReceived));

   if ((CreateSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
      printf("Socket not created \n");
      return 1;
   }

   ipOfServer.sin_family = AF_INET;
   ipOfServer.sin_port = htons(2017);
   ipOfServer.sin_addr.s_addr = inet_addr("127.0.0.1");

   if (connect(CreateSocket, (struct sockaddr *)&ipOfServer, sizeof(ipOfServer)) < 0)
   {
      printf("Connection failed due to port and ip problems\n");
      return 1;
   }

   while ((n = read(CreateSocket, dataReceived, sizeof(dataReceived) - 1)) > 0)
   {
      dataReceived[n] = 0;
      printf("\nCurrent time received from server\n");
      if (fputs(dataReceived, stdout) == EOF)
      {
         printf("\nStandard output error");
      }

      printf("\n");
   }

   if (n < 0)
   {
      printf("Standard input error \n");
   }
```

```
    return 0;
}
```

Output

Discussion

One client-server model drawback is having too many client requests underrun a server and lead to improper functioning or total shutdown. Hackers often use such tactics to terminate specific organizational services through distributed denial-of-service (DDoS) attacks.

Finding and Learnings

In this experiment we learnt how to obtain date and time from server by sending a request to server from client machine.

Advantages of Client-server networks:

1. **Centralized**: Centralized back-up is possible in client-server networks, i.e., all the data is stored in a server.
2. **Security**: These networks are more secure as all the shared resources are centrally administered.
3. **Performance**: The use of the dedicated server increases the speed of sharing resources. This increases the performance of the overall system.
4. **Scalability**: We can increase the number of clients and servers separately, i.e., the new element can be added, or we can add a new node in a network at any time.

Disadvantages of Client-Server network:

1. Traffic Congestion is a big problem in Client/Server networks. When a large number of clients send requests to the same server may cause the problem of Traffic congestion.
2. It does not have a robustness of a network, i.e., when the server is down, then the client requests cannot be met.

Experiment - 2

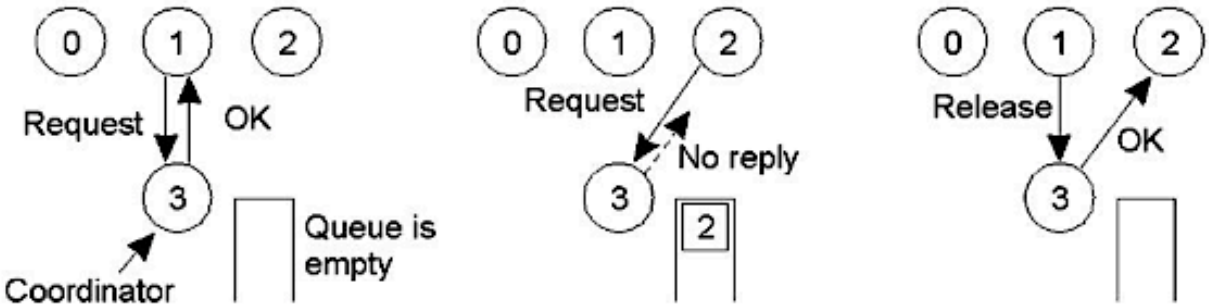
Aim

To implement Mutual Exclusion using centralized algorithm.

Description

To achieve mutual exclusion in distributed systems, the simplest way is to use a centralized algorithm. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address). Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission. When the reply arrives, the requesting process enters the critical region.

Now suppose that another process, asks for permission to enter the same critical region, the coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. Once process 1 exits critical section, it sends a release message to coordinator and other requests are processed by coordinator to allocate critical section to another requesting process.



a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

Code

Server.java

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

```
public class Server implements Runnable {
   Socket socket = null;
   static ServerSocket ss;
   Server(Socket newSocket) {
      this.socket = newSocket;
   }

   public static void main(String args[]) throws IOException {
      ss = new ServerSocket(7000);
      System.out.println("Server Started");
      while (true) {
         Socket s = ss.accept();
         Server es = new Server(s);
         Thread t = new Thread(es);
         t.start();
      }
   }

   public void run() {
      try {
         BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream())
);
         while (true) {
            System.out.println(in.readLine());
         }
      } catch (Exception e) {
      }
   }
}
```

Client1.java

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

public class Client1 {
   public static void main(String args[]) throws IOException {
      Socket s = new Socket("localhost", 7000);
      PrintStream out = new PrintStream(s.getOutputStream());
      ServerSocket ss = new ServerSocket(7001);
      Socket s1 = ss.accept();
      BufferedReader in1 = new BufferedReader(new InputStreamReader(s1.getInputStream()));
      PrintStream out1 = new PrintStream(s1.getOutputStream());
```

```
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str = "Token";
        while (true) {
            if (str.equalsIgnoreCase("Token")) {
                System.out.println("Do you want to send some data");
                System.out.println("Enter Yes or No");
                str = br.readLine();
                if (str.equalsIgnoreCase("Yes")) {
                    System.out.println("Enter the data");
                    str = br.readLine();
                    out.println(str);
                }
                out1.println("Token");
            }
            System.out.println("Waiting for Token");
            str = in1.readLine();
        }
    }
}
```

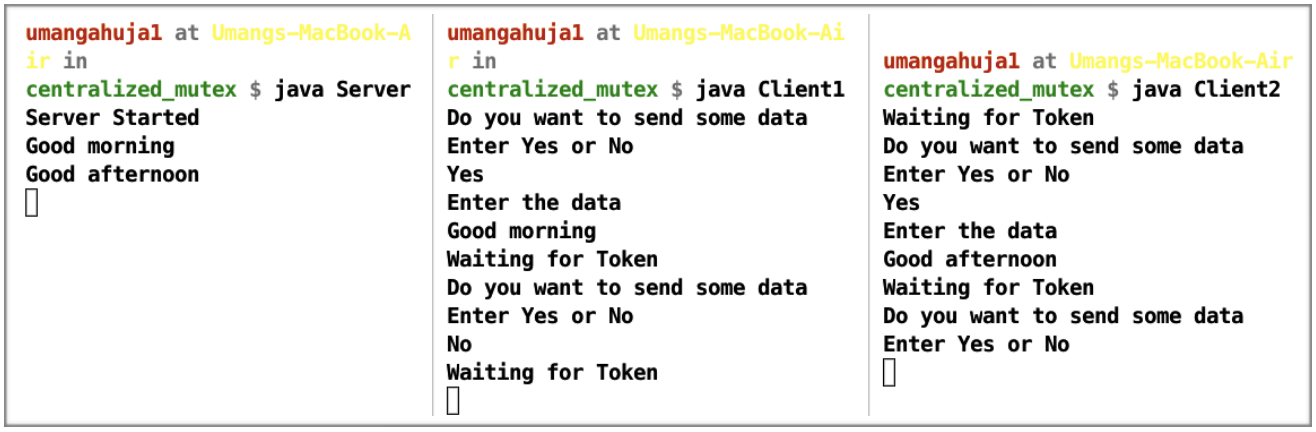
Client2.java

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

public class Client2 {
    public static void main(String args[]) throws IOException {
        Socket s = new Socket("localhost", 7000);
        PrintStream out = new PrintStream(s.getOutputStream());
        Socket s2 = new Socket("localhost", 7001);
        BufferedReader in2 = new BufferedReader(new InputStreamReader(s2.getInputStream()));
        PrintStream out2 = new PrintStream(s2.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        while (true) {
            System.out.println("Waiting for Token");
            str = in2.readLine();
            if (str.equalsIgnoreCase("Token")) {
                System.out.println("Do you want to send some data");
                System.out.println("Enter Yes or No");
                str = br.readLine();
                if (str.equalsIgnoreCase("Yes")) {
                    System.out.println("Enter the data");
```

```
                str = br.readLine();
                out.println(str);
            }
            out2.println("Token");
        }
    }
  }
}
```

Output



Discussion

Though the algorithm is simpler and easier to implement, the centralized approach also has its shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

Finding and Learnings

In this experiment we learnt how to implement centralized algorithm for obtaining mutual exclusion. It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time into each critical region. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release). It can also be used for more general resource allocation rather than just managing critical regions.

Experiment - 3

Aim

To implement Mutual Exclusion using token based algorithm.

Description

In token based algorithm to achieve mutual exclusion, the process possessing a token is granted permission to enter into the critical section. At a particular point of time, only one token is available to enter into the critical section which ensures that mutual exclusion is achieved and only one process can enter critical section at a time. The Suzuki–Kasami algorithm is a token-based algorithm for achieving mutual exclusion in distributed systems.

Code

```
#include <iostream>
#include <queue>
#include <vector>
#define PR 6
using namespace std;
int token_holder;

class Token
{
public:
  int id;             //Id of the site having token
  queue<int> token_q; //Token queue
  int ln[PR];         //Token Array for sequence no
  void init()         //Initializing token
  {
    id = 0;
    for (int i = 0; i < PR; i++)
    {
      ln[i] = 0;
    }
  }
} token;

//Description of each Site
class Site
{
public:
  int rn[PR];    //Site's Array for sequence no.
  bool exec;     //For checking whether site is executing
```

```
  bool isreq;    //For checking whether site is  requesting
  bool hastoken; //For checking whether site has token
  void init()    //Initializing sites
  {
    exec = 0;
    isreq = 0;
    hastoken = 0;
    for (int i = 0; i < PR; i++)
    {
      rn[i] = 0;
    }
  }
  void req(int pid, int seqno);
} site[PR];

//For a site to execute request of site pid with sequenceno seqno
void Site::req(int pid, int seqno)
{
  int i;
  rn[pid] = max(rn[pid], seqno);
  if (hastoken == 1)
  {
    if (exec == 0 && token.ln[pid] + 1 == rn[pid])
    {
      hastoken = 0;
      token_holder = pid;
    }
    else if (token.ln[pid] + 1 == rn[pid])
    {
      token.token_q.push(pid);
    }
  }
}

//Initialize
void initialize()
{
  int i;
  token.init();
  for (i = 0; i < PR; i++)
  {
    site[i].init();
  }
```

```
}

//For a site with id pid to request for C.S.
void request(int pid)
{
 int i, seqno;
 seqno = ++site[pid].rn[pid];
 //Checking whether it has already requested
 if (site[pid].isreq == 1 || site[pid].exec == 1)
 {
  printf("SITE %d is already requesting \n", pid);
  return;
 }
 site[pid].isreq = 1;
 //Checking if it has the token
 if (token_holder == pid)
 {
  site[pid].isreq = 0;
  site[pid].exec = 1;
  printf("SITE %d already has the token and it enters the critical section\n", pid);
  return;
 }

 //Sending Request
 if (token_holder != pid)
 {
  for (i = 0; i < PR; i++)
  {
   if (i != pid)
     site[i].req(pid, seqno);
  }
 }
 //Checking if it has got the token
 if (token_holder == pid)
 {
  site[pid].hastoken = 1;
  site[pid].exec = 1;
  site[pid].isreq = 0;
  printf("SITE %d gets the token and it enters the critical section\n", pid);
 }
 else
 {
```

```
    printf("SITE %d is currently executing the critical section \nSite %d has placed its request\n", tok
en_holder, pid);
  }
}

//For a site with id pid to request for C.S.
void release(int pid)
{

  if (site[pid].exec != 1)
  {
    printf("SITE %d is not currently executing the critical section \n", pid);
    return;
  }
  int i, siteid;
  token.ln[pid] = site[pid].rn[pid];
  site[pid].exec = 0;
  printf("SITE %d releases the critical section\n", pid);
  //Checking if deffred requests are there by checking token queue
  //And Passing the token if queue is non empty
  if (!token.token_q.empty())
  {
    siteid = token.token_q.front();
    token.token_q.pop();
    token.id = siteid;
    site[pid].hastoken = 0;
    token_holder = siteid;
    site[siteid].hastoken = 1;
    site[siteid].exec = 1;
    site[siteid].isreq = 0;
    printf("SITE %d gets the token and it enters the critical section\n", siteid);
    return;
  }
  printf("SITE %d still has the token\n", pid);
}

//Printing the state of the system
void print()
{
  int i, j, k = 0;
  queue<int> temp;
  printf("TOKEN STATE :\n");
  printf("TOKEN HOLDER :%d\n", token_holder);
```

```
  printf("TOKEN QUEUE: ");
  if (token.token_q.empty())
  {
   printf("EMPTY");
   j = 0;
  }
  else
  {
   j = token.token_q.size();
  }
  while (k < j)
  {
   i = token.token_q.front();
   token.token_q.pop();
   token.token_q.push(i);
   printf("%d ", i);
   k++;
  }
  printf("\n");
  printf("TOKEN SEQ NO ARRAY: ");
  for (i = 0; i < PR; i++)
   printf("%d ", token.ln[i]);
  printf("\n");

  printf("SITES SEQ NO ARRAY: \n");
  for (i = 0; i < PR; i++)
  {
   printf(" S%d :", i);
   for (j = 0; j < PR; j++)
     printf(" %d ", site[i].rn[j]);
   printf("\n");
  }
}
int main()
{
  int i, j, time, pid;
  string str;
  initialize();
  token_holder = 0;
  site[0].hastoken = 1;

  time = 0;
```

```
  cout << "THE NO OF SITES IN THE DISTRIBUTED SYSTEM ARE " << PR << endl;
  cout << "INITIAL STATE\n"
       << endl;
  print();
  printf("\n");
  while (str != "OVER")
  {
    cin >> str;
    if (str == "REQ")
    {
      cin >> pid;
      cout << "EVENT :" << str << " " << pid << endl
           << endl;
      request(pid);
      print();
      printf("\n");
    }
    else if (str == "REL")
    {
      cin >> pid;
      cout << "EVENT :" << str << " " << pid << endl
           << endl;
      release(pid);
      print();
      printf("\n");
    }
  }
}
```

Output

```
lab $ ./a.out
THE NO OF SITES IN THE DISTRIBUTED SYSTEM ARE 6
INITIAL STATE

TOKEN STATE :
TOKEN HOLDER :0
TOKEN QUEUE: EMPTY
TOKEN SEQ NO ARRAY: 0 0 0 0 0 0
SITES SEQ NO ARRAY:
 S0 : 0  0  0  0  0  0
 S1 : 0  0  0  0  0  0
 S2 : 0  0  0  0  0  0
 S3 : 0  0  0  0  0  0
 S4 : 0  0  0  0  0  0
 S5 : 0  0  0  0  0  0

REQ 1
EVENT :REQ 1
```

```
SITE 1 gets the token and it enters the critical section
TOKEN STATE :
TOKEN HOLDER :1
TOKEN QUEUE: EMPTY
TOKEN SEQ NO ARRAY: 0 0 0 0 0 0
SITES SEQ NO ARRAY:
 S0 : 0  1  0  0  0  0
 S1 : 0  1  0  0  0  0
 S2 : 0  1  0  0  0  0
 S3 : 0  1  0  0  0  0
 S4 : 0  1  0  0  0  0
 S5 : 0  1  0  0  0  0

REQ 2
EVENT :REQ 2
```

```
SITE 1 is currently executing the critical section
Site 2 has placed its request
TOKEN STATE :
TOKEN HOLDER :1
TOKEN QUEUE: 2
TOKEN SEQ NO ARRAY: 0 0 0 0 0 0
SITES SEQ NO ARRAY:
 S0 : 0  1  1  0  0  0
 S1 : 0  1  1  0  0  0
 S2 : 0  1  1  0  0  0
 S3 : 0  1  1  0  0  0
 S4 : 0  1  1  0  0  0
 S5 : 0  1  1  0  0  0

REL 1
EVENT :REL 1
```

```
SITE 1 releases the critical section
SITE 2 gets the token and it enters the critical section
TOKEN STATE :
TOKEN HOLDER :2
TOKEN QUEUE: EMPTY
TOKEN SEQ NO ARRAY: 0 1 0 0 0 0
SITES SEQ NO ARRAY:
 S0 : 0  1  1  0  0  0
 S1 : 0  1  1  0  0  0
 S2 : 0  1  1  0  0  0
 S3 : 0  1  1  0  0  0
 S4 : 0  1  1  0  0  0
 S5 : 0  1  1  0  0  0

REL 2
EVENT :REL 2
```

```
SITE 2 releases the critical section
SITE 2 still has the token
TOKEN STATE :
TOKEN HOLDER :2
TOKEN QUEUE: EMPTY
TOKEN SEQ NO ARRAY: 0 1 1 0 0 0
SITES SEQ NO ARRAY:
 S0 : 0  1  1  0  0  0
 S1 : 0  1  1  0  0  0
 S2 : 0  1  1  0  0  0
 S3 : 0  1  1  0  0  0
 S4 : 0  1  1  0  0  0
 S5 : 0  1  1  0  0  0

OVER
```

Discussion

If a process wants to enter its critical section and it does not have the token, it broadcasts a request message to all other processes in the system. The process that has the token, if it is not currently in a critical section, will then send the token to the requesting process. If a site receives a request message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

Finding and Learnings

In this experiment we learnt how to implement token based algorithm for obtaining mutual exclusion.

Message Complexity:

The algorithm requires 0 message invocation if the site already holds the idle token at the time of critical section request or maximum of N message per critical section execution. This N messages involves

1. (N – 1) request messages
2. 1 reply message

Drawbacks of Suzuki–Kasami Algorithm:

Non-symmetric Algorithm: A site retains the token even if it does not have requested for critical section.

Performance:

1. Synchronization delay is 0 and no message is needed if the site holds the idle token at the time of its request.
2. In case site does not holds the idle token, the maximum synchronization delay is equal to maximum message transmission time and a maximum of N message is required per critical section invocation.

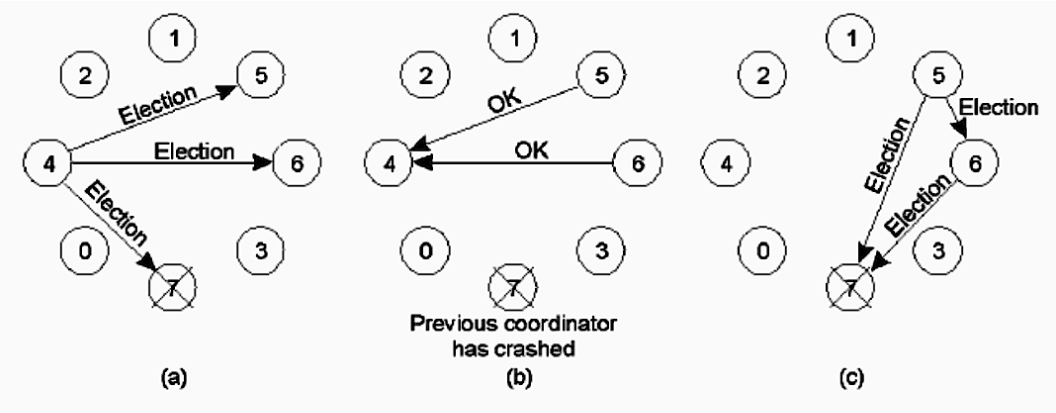
Experiment - 4

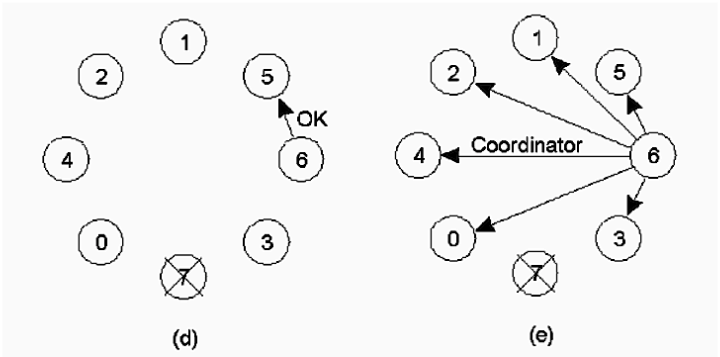
Aim

To implement Bully election algorithm.

Description

In distributed computing, the bully algorithm is a method for dynamically electing a coordinator or leader from a group of distributed computer processes. The process with the highest process ID number from amongst the non-failed processes is selected as the coordinator.





The algorithm uses the following message types:

1. Election Message: Sent to announce election.
2. Answer (Alive) Message: Responds to the Election message.
3. Coordinator (Victory) Message: Sent by winner of the election to announce victory.

When a process P recovers from failure, or the failure detector indicates that the current coordinator has failed, P performs the following actions:

1. If P has the highest process id, it sends a Victory message to all other processes and becomes the new Coordinator. Otherwise, P broadcasts an Election message to all other processes with higher process IDs than itself.
2. If P receives no Answer after sending an Election message, then it broadcasts a Victory message to all other processes and becomes the Coordinator.

3. If P receives an Answer from a process with a higher ID, it sends no further messages for this election and waits for a Victory message. (If there is no Victory message after a period of time, it restarts the process at the beginning.)
4. If P receives an Election message from another process with a lower ID it sends an Answer message back and starts the election process at the beginning, by sending an Election message to higher-numbered processes.
5. If P receives a Coordinator message, it treats the sender as the coordinator.

Code

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

class Bully {
   static int n;
   static int pro[] = new int[100];
   static int sta[] = new int[100];
   static int co;


public static void main(String args[]) throws IOException {

      System.out.print("Enter the number of process : ");
      Scanner in = new Scanner(System.in);
      n = in.nextInt();

      int i, j, k, l, m;

      for (i = 0; i < n; i++) {
         System.out.println("For process " + (i + 1) + ":");
         System.out.print("Status : ‘’);

         sta[i] = in.nextInt();
         System.out.print("Priority : ");
         pro[i] = in.nextInt();
      }

      System.out.println("Which process will initiate election?");
      int ele = in.nextInt();

      elect(ele);
      System.out.println("Final coordinator is " + co);
   }
```

```
    static void elect(int ele) {
        ele = ele - 1;
        co = ele + 1;

        for (int i = 0; i < n; i++) {
            if (pro[ele] < pro[i]) {
                System.out.println("Election message is sent from " + (ele + 1) + " to " + (i + 1));

            if (sta[i] == 1)
                    elect(i + 1);
            }

        }

    }
}
```

Output

```
lab $ java Bully
Enter the number of process : 4
For process 1:
Status : 1
Priority : 4
For process 2:
Status : 1
Priority : 3
For process 3:
Status : 1
Priority : 2
For process 4:
Status : 1
Priority : 1
Which process will initiate election?
4
Election message is sent from 4 to 1
Election message is sent from 4 to 2
Election message is sent from 2 to 1
Election message is sent from 4 to 3
Election message is sent from 3 to 1
Election message is sent from 3 to 2
Election message is sent from 2 to 1
Final coordinator is 1
```

Discussion

Assuming that the bully algorithm messages are of a fixed (known, invariant) sizes, the most number of messages are exchanged in the group when the process with the lowest id initiates an election. This process sends (N-1) Election messages, the next higher id sends (N-2) messages, and so on, resulting in O(n^2) election messages. There are also the O(n^2) Alive messages, and O(n) co-ordinator messages, thus making the overall number messages exchanged in the worst case be O(n^2).

Finding and Learnings

In this experiment we learnt how to perform bully election algorithm to select a coordinator from a group of processes. The safety property expected of leader election protocols is that every non-faulty process either elects a process Q, or elects none at all. Note that all processes that elect a leader must decide on the same process Q as the leader. The Bully algorithm satisfies this property (under the system model specified), and at no point in time is it possible for two processes in the group to have a conflicting view of who the leader is, except during an election. This is true because if it weren't, there are two processes X and Y such that both sent the Coordinator (victory) message to the group. This means X and Y must also have sent each other victory messages. But this cannot happen, since before sending the victory message, Election messages would have been exchanged between the two, and the process with a lower process id among the two would never send out victory messages. We have a contradiction, and hence our initial assumption that there are two leaders in the system at any given time is false, and that shows that the bully algorithm is safe.

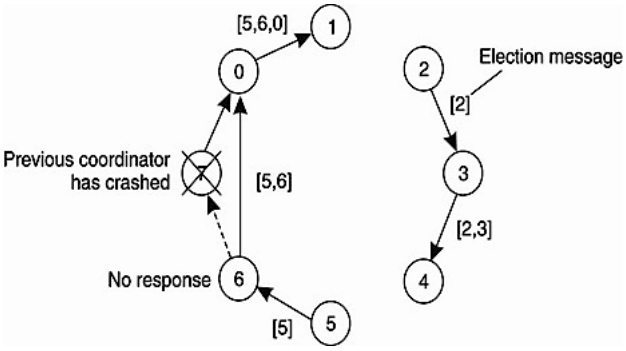
Experiment - 5

Aim

To implement ring election algorithm.

Description

Ring algorithm assumes that processes are logically ordered in some fashion, and that each process knows the order and who is coordinator. No token is involved. When a process notices that the coordinator is not responding it sends an ELECTION message with its own id to its downstream neighbor. If that neighbor doesn't respond it sends it to its neighbor's neighbor, etc. Each station that receives the ELECTION message adds its own id to the list. When the message circulates back to the originator it selects the highest id in the list and sends a COORDINATOR message announcing the new coordinator. This message circulates once and is removed by the originator.



Code

```
import java.util.Scanner;

class Process {
    public int id;
    public boolean active;
    public Process(int id) {
        this.id = id;
        active = true;
    }
}

public class Ring {
    int noOfProcesses;
    Process[] processes;
    Scanner sc;
    public Ring() {
        sc = new Scanner(System.in);
```

```
    }

    public void initialiseRing() {
        System.out.println("Enter no of processes");
        noOfProcesses = sc.nextInt();
        processes = new Process[noOfProcesses];
        for (int i = 0; i < processes.length; i++) {
            processes[i] = new Process(i);
        }
    }

    public int getMax() {
        int maxId = -99;
        int maxIdIndex = 0;
        for (int i = 0; i < processes.length; i++) {
            if (processes[i].active && processes[i].id > maxId) {
                maxId = processes[i].id;
                maxIdIndex = i;

            }
        }
        return maxIdIndex;
    }

    public void performElection() {
        System.out.println("Process no " + processes[getMax()].id + " fails");
        processes[getMax()].active = false;
        System.out.println("Election Initiated by");
        int initiatorProcesss = sc.nextInt();

        int prev = initiatorProcesss;
        int next = prev + 1;

        while (true) {
            if (processes[next].active) {
                System.out.println("Process " + processes[prev].id + " pass Election(" + processes[prev].i
d + ") to "
                        + processes[next].id);
                prev = next;
            }

            next = (next + 1) % noOfProcesses;
            if (next == initiatorProcesss) {
```

```
            break;
        }
    }

    System.out.println("Process " + processes[getMax()].id + " becomes coordinator");
    int coordinator = processes[getMax()].id;
    prev = coordinator;
    next = (prev + 1) % noOfProcesses;
    while (true) {
        if (processes[next].active) {
            System.out.println("Process " + processes[prev].id + " pass Coordinator(" + coordinator
                    + ") message to process " + processes[next].id);
            prev = next;
        }
        next = (next + 1) % noOfProcesses;
        if (next == coordinator) {
            System.out.println("End Of Election ");
            break;
        }
    }
}

public static void main(String arg[]) {
    Ring r = new Ring();
    r.initialiseRing();
    r.performElection();
}
}
```

Output

```
lab $ java Ring
Enter no of processes
5
Process no 4 fails
Election Initiated by
2
Process 2 pass Election(2) to 3
Process 3 pass Election(3) to 0
Process 0 pass Election(0) to 1
Process 3 becomes coordinator
Process 3 pass Coordinator(3) message to process 0
Process 0 pass Coordinator(3) message to process 1
Process 1 pass Coordinator(3) message to process 2
End Of Election
```

Discussion

If to different process discover a crash at the same time, each of these builds an ELECTION message and starts circulating it. Eventually, both messages will go all the way around, and both will convert them into COORDINATOR messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating but it wastes a little bandwidth.

Finding and Learnings

In this experiment we learnt how to perform ring election algorithm to select a coordinator from a group of processes.

Comparison Bully algorithm and Ring algorithm:

Assume n processes and one election in progress, then

1. Bully algorithm

Worst case: initiator will be node together with least expensive ID

Triggers n-2 elections with higher ranked nodes: O(n^2) msgs

Best case: immediate election: n-2 messages

2. Ring algorithm

2*(n-1) messages always

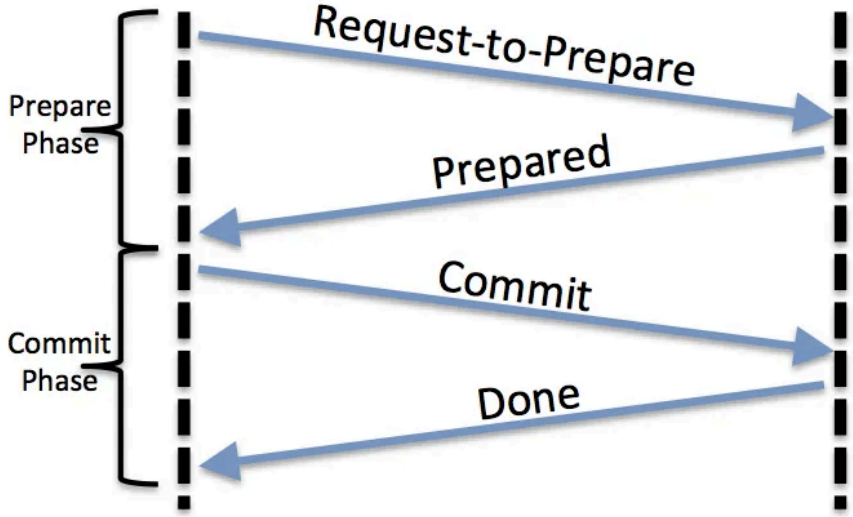
Experiment - 6

Aim

To implement 2 phase commit protocol.

Description

A two-phase commit is a standardized protocol that ensures that a database commit is implementing in the situation where a commit operation must be broken into two separate parts. In transaction processing, databases, and computer networking, the two-phase commit protocol (2PC) is a type of atomic commitment protocol (ACP). It is a distributed algorithm that coordinates all the processes that participate in a distributed atomic transaction on whether to commit or abort (roll back) the transaction (it is a specialized type of consensus protocol). The protocol achieves its goal even in many cases of temporary system failure (involving either process, network node, communication, etc. failures), and is thus widely used.



2 Phase Commit

Code

Server.java

```
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) {
 }
 ;
 }
}
```

Client.java

```
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);
        }
    }
}
```

Output

Discussion

Phase 1 - Each server that needs to commit data writes its data records to the log. If a server is unsuccessful, it responds with a failure message. If successful, the server replies with an OK message.

Phase 2 - This phase begins after all participants respond OK. Then, the coordinator sends a signal to each server with commit instructions. After committing, each writes the commit as part of its log record for reference and sends the coordinator a message that its commit has been successfully implemented. If a server fails, the coordinator sends instructions to all servers to roll back the transaction. After the servers roll back, each sends feedback that this has been completed.

Finding and Learnings

In this experiment we learnt how to two phase commit protocol works to commit a transaction by taking agreement from all the cohorts.

Disadvantages

The greatest disadvantage of the two phase commit protocol is the fact that it is a blocking protocol. A node will block while it is waiting for a message. This means that other processes competing for resource locks held by the blocked processes will have to wait for the locks to be released. A single node will continue to wait even if all other sites have failed. If the coordinator fails permanently, some cohorts will never resolve their transactions. This has the effect that resources are tied up forever.

Another disadvantage is the protocol is conservative. It is biased to the abort case rather than the complete case.