Distributed Systems

Practical File (CO-405)



Submitted By: SATYAM BANSAL (2K16/CO/277)

**TABLE OF CONTENTS**

|  |  |  |  |
| --- | --- | --- | --- |
| S. No. | Program | Date | Signature |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# PROGRAM-1

### **AIM:** Implement concurrent day-time client-server application.

**THEORY**:

There are two major transport layer protocols to communicate between hosts : TCP and UDP. In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive. Datagrams upon arrival contain the address of sender which the server uses to send data to the correct client.

**Socket:**

A socket is a combination of IP address and port on one system. On each system a socket exists for a process interacting with the socket on other system over the network. A combination of local socket and the socket at the remote system is also known a ‘Four tuple’ or ‘4-tuple’. Each connection between two processes running at different systems can be uniquely identified through their 4-tuple.

**Function Descriptions socket()**

Creates an UN-named socket inside the kernel and returns an integer known as socket descriptor. This function takes domain/family as its first argument. For Internet family of ipv4 addresses we use AF\_INET. The second argument ‘SOCK\_STREAM’ specifies that the transport layer protocol that we want should be reliable i.e. It should have acknowledgement techniques. The third argument is generally left zero to let the kernel decide the default protocol to use for this connection. For connection oriented reliable connections, the default protocol used is TCP.

**bind()**

Assigns the details specified in the structure ‘serv\_addr’ to the socket created in the step above. The details include, the family/domain, the interface to listen on(in case the system has multiple interfaces to network) and the port on which the server will wait for the client requests to come.

**listen()**

With second argument as ’10’ specifies maximum number of client connections that server will queue for this listening socket. After the call to listen(), this socket becomes a fully functional listening socket.

**accept()**

The server is put to sleep and when for an incoming client request, the threeway TCP handshake is complete, the function accept () wakes up and returns the socket descriptor representing the client socket. Accept() is run in an infinite loop so that the server is always running and the delay or sleep of 1 sec ensures that this server does not eat up all your CPU processing. As soon as server gets a request from client, it prepares the date and time and writes on the client socket through the descriptor returned by accept().

**ALGORITHM:**

server()

* create UDP socket
* Bind socket to address
* wait for datagram from client
* process and reply to client request
* repeat while server is active

client()

* create UDP socket
* send request to server
* wait for datagram from server
* process and reply from server
* close socket and exit

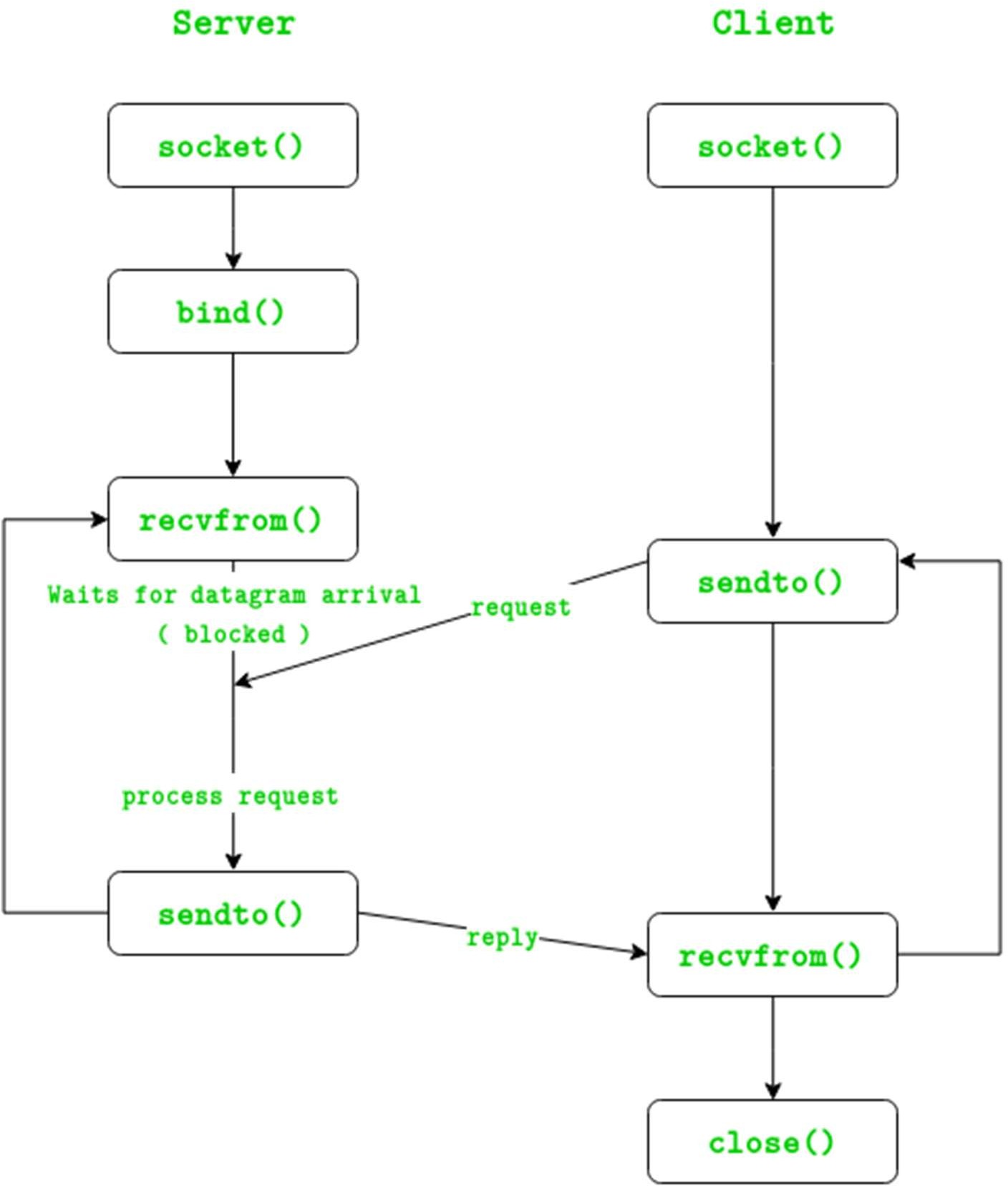


FIG: UDP Server-Client

**SERVER CODE**

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

int main()

{

struct sockaddr\_in sa; // Socket address data structure

int sockfd, coontfd; // Source and destination addresses char str[1025]; // Buffer to hold the out-going stream time\_t tick; // System time data structure

sockfd = socket(AF\_INET, SOCK\_STREAM, 0); // New socket created

// Checking for valid socket if (sockfd < 0)

{

printf("Error in creating socket\n"); exit(0);

}

else

{

printf("Socket Created\n");

}

// Clearing and assigning type and address to the socket printf("Socket created\n");

bzero(&sa, sizeof(sa));

memset(str, '0', sizeof(str)); // clearing the buffer sa.sin\_family = AF\_INET;

sa.sin\_port = htons(5600); sa.sin\_addr.s\_addr = htonl(INADDR\_ANY);

// binding and verifying the socket to address

if (bind(sockfd, (struct sockaddr\*)&sa, sizeof(sa))<0)

{

printf("Bind Error\n");

}

else

printf("Binded\n");

// starts the server with a max client queue size set as 10 listen(sockfd, 10);

// server run

while(1)

{

coontfd = accept(sockfd, (struct sockaddr\*)NULL ,NULL); // Accept a request from client printf("Accepted\n");

tick = time(NULL);

snprintf(str, sizeof(str), "%.24s\r\n", ctime(&tick)); // read sys time and write to buffer printf("sent\n");

printf("%s\n", str);

write(coontfd, str, strlen(str)); // send buffer to client

}

close(sockfd); // close the socket return 0;

}

## CLIENT CODE

#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()

{

struct sockaddr\_in sa; // Socket address data structure

int n, sockfd; // read and source

char buff[1025]; // buffer to store the read stream

sockfd = socket(PF\_INET, SOCK\_STREAM, 0); // New socket created

// Checking for valid socket if (sockfd < 0)

{ printf("Error in creation\n"); exit(0);

}

else

printf("Socket created\n");

// Clearing and assigning type and address to the socket bzero(&sa, sizeof(sa));

sa.sin\_family = AF\_INET; sa.sin\_port = htons(5600);

// establishing and verifying the connection

if (connect(sockfd, (struct sockaddr\_in\*)&sa, sizeof(sa)) < 0)

{ printf("Connection failed\n"); exit(0);

}

else

printf("Connection made\n");

// Reading and priting data from the server after verification if ( n = read(sockfd, buff, sizeof(buff)) < 0)

{ printf("Read Error\n"); exit(0);

}

else

{

printf("Read message: %s\n", buff);

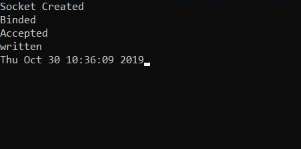
printf("%s\n", buff); printf("Done with connection, exiting\n");

}

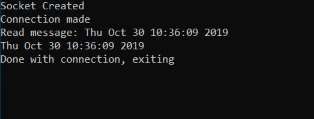
close(sockfd); // Closing the socket return 0;

}

## SERVER OUTPUT:



**CLIENT OUTPUT:**



**FINDING AND LEARNING:**

1. We successfully implemented a date-time client-server.
2. UDP is a connectionless protocol where the server waits for a request from a client to become active. each connection is treated as a new one.
3. On a local system i.e. within the same computer, the loop back address should be used as the argument to the client.
4. The connect procedure follows the Three way handshake process to establish the connection.

**PROGRAM-2**

**AIM**: Implement Lamport Clock Synchronization

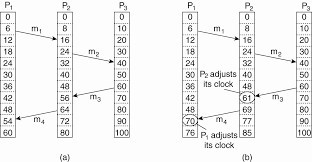
**THEORY**

One of the main properties of a distributed system is that it **lacks a global clock**. All the processes have their own local clock, but due to clock skew and clock drift they have no direct way to know if their clock is in check with the local clocks of the other processes in the system, this problem is sometimes referred to as the **problem of clock synchronization**.

Solutions to this problem consist of using a central time server (Cristian’s Algorithm) or a mechanism called a **logical clock**. The problem with a central time server is that its error depends on the round- trip time of the message from process to time server and back.

Logical clocks are based on capturing chronological and causal relationships of processes and ordering events based on these relationships. The first implementation, the Lamport timestamps, was proposed by **Leslie Lamport** in 1978 and still forms the foundation of almost all logical clocks.

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 message happened-before the other: ( A A happens-before B B if one can get from A A to B B 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 above-mentioned vector clock is a generalization of the idea into the context of an arbitrary number of parallel, independent processes.

**Lamport Timestamps algorithm**

A Lamport logical clock is an incrementing 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 (causality).

The algorithm of Lamport Timestamps can be captured in a few rules:

* + All the process counters start with value 0.
  + A process increments its counter for each event (internal event, message sending, message receiving) in that process.
  + When a process sends a message, it includes its (incremented) counter value with the message.
  + On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one.

Looking at these rules, we can see the algorithm will create a minimum overhead, since the counter consists of just one integer value and the messaging piggybacks on inter-process messages.

**CODE**

#include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <errno.h> #include <string.h> #include <sys/types.h> #include <time.h> #define MSG\_CONFIRM 0

#define TRUE 1

#define FALSE 0

#define ML 1024

#define MPROC 32

/\*

Function to create a new connection to port 'connect\_to'

1. Creates the socket.
2. Binds to port.
3. Returns socket id

\*/

typedef struct lamport\_clock{ int timer;

}lamport\_clock;

void init(lamport\_clock \*clk)

{

clk->timer = 0;

}

void tick(lamport\_clock \*clk, int phase)

{

clk->timer += phase;

}

int str\_to\_int(char str[ML], int n)

{

int x = 0, i = 0, k; printf("x: %d\n", x); for (i = 0; i < n; i++)

{

k = atoi(str[i]); x = x\*10 + k;

}

return x;

}

void update\_clock(lamport\_clock \*clk, int new\_time)

{

clk->timer = new\_time;

}

int connect\_to\_port(int connect\_to)

{

int sock\_id; int opt = 1;

struct sockaddr\_in server;

if ((sock\_id = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0)

{

perror("unable to create a socket"); exit(EXIT\_FAILURE);

}

setsockopt(sock\_id, SOL\_SOCKET, SO\_REUSEADDR, (const void \*)&opt, sizeof(int)); memset(&server, 0, sizeof(server));

server.sin\_family = AF\_INET; server.sin\_addr.s\_addr = INADDR\_ANY; server.sin\_port = htons(connect\_to);

if (bind(sock\_id, (const struct sockaddr \*)&server, sizeof(server)) < 0)

{

perror("unable to bind to port"); exit(EXIT\_FAILURE);

}

return sock\_id;

}

/\*

sends a message to port id to

\*/

void send\_to\_id(int to, int id, lamport\_clock clk)

{

struct sockaddr\_in cl; memset(&cl, 0, sizeof(cl));

char message[ML]; sprintf(message, "%d", clk.timer);

cl.sin\_family = AF\_INET; cl.sin\_addr.s\_addr = INADDR\_ANY; cl.sin\_port = htons(to);

sendto(id, \

(const char \*)message, \ strlen(message), \ MSG\_CONFIRM, \

(const struct sockaddr \*)&cl, \ sizeof(cl));

}

/\*

announces completion by sending coord messages

\*/

int main(int argc, char\* argv[])

{

// 0. Initialize variables int self = atoi(argv[1]);

int n\_proc = atoi(argv[2]); int phase = atoi(argv[3]);

int procs[MPROC]; int sock\_id;

int new\_time;

int itr, len, n, start\_at;

char buff[ML], message[ML]; struct sockaddr\_in from;

lamport\_clock self\_clock;

for (itr = 0; itr < n\_proc; itr += 1) procs[itr] = atoi(argv[4 + itr]);

start\_at = atoi(argv[4 + n\_proc]) == 1? TRUE : FALSE; init(&self\_clock);

tick(&self\_clock, phase);

// 1. Create socket

printf("creating a node at %d %d \n", self, start\_at); sock\_id = connect\_to\_port(self);

// getchar();

// 2. check is process is initiator

if (start\_at == TRUE)

{

printf("Proc %d is starting comms \n", self); for (itr = 0; itr < n\_proc; itr++)

{

printf("Sending to proc: %d", itr); send\_to\_id(procs[itr], sock\_id, self\_clock);

}

}

// 3. if not the initiator wait for someone else while(TRUE)

{

printf("\t \n\n");

sleep(1); tick(&self\_clock, phase);

memset(&from, 0, sizeof(from));

n = recvfrom(sock\_id, (char \*)buff, ML, MSG\_WAITALL, (struct sockaddr \*)&from, &len); buff[n] = '\0';

printf("Recieved time: %s Self time: %d\n", buff, self\_clock.timer); new\_time = atoi(buff);

// printf("Recieved time: %s %d\n", buff, new\_time);

if (new\_time > self\_clock.timer)

{

printf("\nNew time > Current time: synchronizing clocks\n\t \n");

printf("Current time: %d\n", self\_clock.timer);\ printf("Updated time: %d\n", new\_time + 1); update\_clock(&self\_clock, new\_time + 1);

}

else

{

printf("No need to synchronize times\n");

}

for (itr = 0; itr < n\_proc; itr++)

{

printf("Sending time %d to proc %d\n", self\_clock.timer, itr); send\_to\_id(procs[itr], sock\_id, self\_clock);

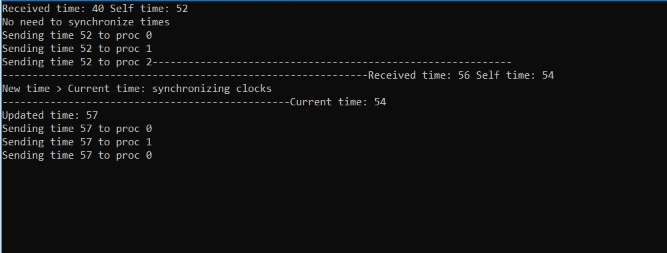
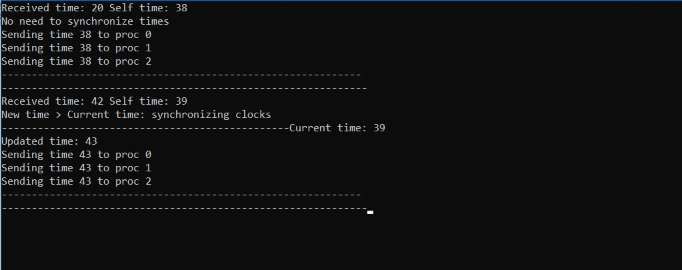
}

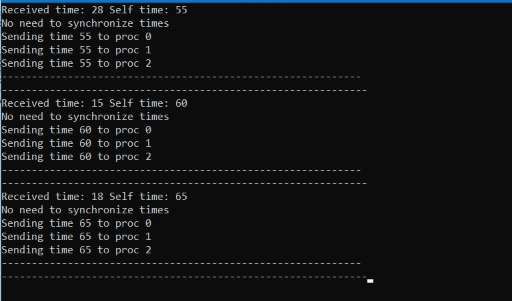
printf("\t \n\n");

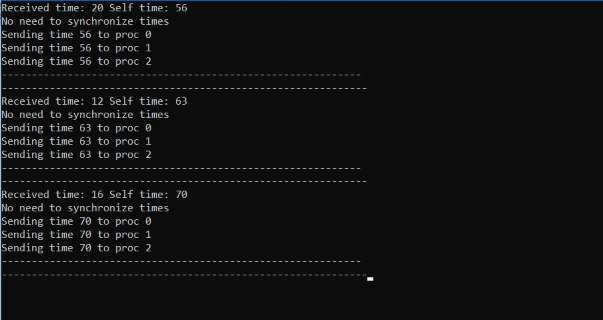
}

}

## OUTPUT







**FINDING AND LEARNING**

1. We successfully implemented Lamport Clock .
2. One of the shortcomings of Lamport clocks is rooted in the fact that they only partially order events (as opposed to total order).
3. **Partial order** indicates that not every pair of events need be comparable.
4. If two events can’t be compared, we call these events **concurrent**.
5. The problem with Lamport clocks is that they can’t tell if events are concurrent or not.
6. This problem is solved by Vector Clocks.

**PROGRAM-3**

**AIM**: Implement Mutual Exclusion using Centralized Algorithm.

**THEORY:**

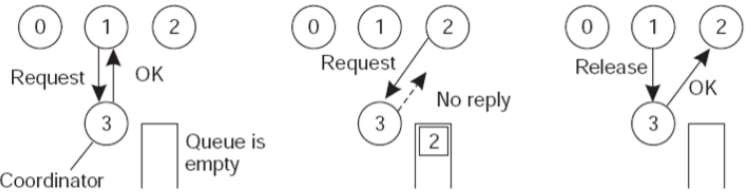
**Mutual exclusion** is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.

**Mutual exclusion in single computer system Vs. distributed system:**

In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.

In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.

In centralized algorithm one process is elected as the coordinator which may be the machine. 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. When another process asks for permission to enter the same critical region. Now the coordinator knows that a different process is already in the critical region, so it cannot grant permission. The coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply or it could send a reply ‘permission denied.’ When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access. The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. When it sees the grant, it can enter the critical region.



* **Advantages**
  + Algorithm guarantees mutual exclusion by letting one process at a time into each critical region.
  + It is also fair as requests are granted in the order in which they are received.
  + No process ever waits forever so no starvation.
  + Easy to implement so it requires only three messages per use of a critical region (request, grant, release).
  + Used for more general resource allocation rather than just managing critical regions.
* **Disadvantages**
  + The coordinator is a single point of failure, the entire system may go down if it crashes.
  + If processes normally block after making a request, they cannot distinguish a dead coordinator from ‘‘permission denied’’ since no message comes back.
  + In a large system a single coordinator can become a performance bottleneck.

**CONTROLLER CODE**

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

#define TRUE 1

#define FALSE 0

typedef struct resources

{

int A; int B; int C; int D;

} resources;

int main()

{

resources R, temp;

R.A = 1;

R.B = 2;

R.C = 3;

R.D = 4; FILE \*fle;

fle = fopen("shared\_mem.txt", "w"); fwrite(&R, sizeof(R), 1, fle); fclose(fle);

struct sockaddr\_in sa; // Socket address data structure int opt = TRUE, addrlen;

char buff[256]; // Buffer to hold the out-going stream int rec, i, sd, activity, new\_sock, sended;

int max\_sd; int flag = 0;

sockfd = socket(AF\_INET, SOCK\_STREAM, 0); // New socket created

// Checking for valid socket

memset(clients, 0, sizeof(clients));

fd\_set readfds; if (sockfd < 0)

{

}

else

{

}

printf("Error in creating socket\n"); exit(0);

printf("Socket Created\n");

if (setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, (char \*)&opt, sizeof(opt)) < 0)

{

printf("error\n");

}

// Clearing and assigning type and address to the socket printf("Socket created\n");

bzero(&sa, sizeof(sa)); sa.sin\_family = AF\_INET; sa.sin\_port = htons(8888);

sa.sin\_addr.s\_addr = htonl(INADDR\_ANY);

// binding and verifying the socket to address

if (bind(sockfd, (struct sockaddr \*)&sa, sizeof(sa)) < 0)

{

}

else

printf("Bind Error\n");

printf("Binded\n");

// starts the server with a max client queue size set as 10 listen(sockfd, 10);

addrlen = sizeof(sa);

// server run while (TRUE)

{

// Clearing socket set FD\_ZERO(&readfds);

FD\_SET(sockfd, &readfds); max\_sd = sockfd;

for (i = 0; i < 50; i++)

{

if (sd > 0)

FD\_SET(sd, &readfds); if (sd > max\_sd)

max\_sd = sd;

}

activity = select(max\_sd + 1, &readfds, NULL, NULL, NULL); if (activity < 0)

printf("Select error\n"); if (FD\_ISSET(sockfd, &readfds))

{

if ((new\_sock = accept(sockfd, (struct sockaddr \*)NULL, NULL)) < 0) perror("accept");

else

{

}

printf("New connection, sock fd %d\n", new\_sock);

sended = send(new\_sock, buff, strlen(buff), 0); if (sended < 0)

perror("Send"); for (i = 0; i < 50; i++)

{

if (clients[i] == 0)

{

clients[i] = new\_sock; break;

}

}

}

for (i = 0; i < 50; i++)

{

sd = clients[i];

if (FD\_ISSET(sd, &readfds))

{

FILE \*fle;

fle = fopen("shared\_mem.txt", "r"); fread(&temp, sizeof(temp), 1, fle); fclose(fle);

rec = read(sd, buff, 256); if (rec == 0)

{

getpeername(sd, (struct sockaddr \*)&sa,

(socklen\_t \*)&sa);

printf("%d has disconnected unexpectedly with ip %s and port %d\n", sd, inet\_ntoa(sa.sin\_addr), ntohs(sa.sin\_port));

printf("recovering data\n"); FILE \*fle;

fle = fopen("shared\_mem.txt", "w+"); fwrite(&temp, sizeof(temp), 1, fle); fclose(fle);

close(sd); clients[i] = 0;

}

else

{

buff[rec] = '\0';

printf("recieved %s from %d\n", buff, sd);

if (strcmp(buff, "PING") == 0 && flag == 1)

{

printf("Read buffer = %s, from %d and send NACK\n", buff, sd); sended = write(sd, "NACK", 4);

}

else if (strcmp(buff, "PING") == 0 && flag == 0)

{

printf("Read Buffer = %s, from %d\n", buff, sd); flag = 1;

sended = write(sd, "PONG", 4);

}

}

else if (strcmp(buff, "DONE") == 0)

{

printf("Lock freed\n"); flag = 0;

FILE \*fle;

fle = fopen("shared\_mem.txt", "r"); fread(&temp, sizeof(temp), 1, fle);

printf("Read %d, %d, %d, %d from %d\n", temp.A, temp.B, temp.C, temp.D, sd);

fclose(fle); clients[i] = 0; close(sd); break;

}

}

}

}

}

close(sockfd); // close the socket return 0;

}

## CLIENT CODE

#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<unistd.h>

typedef struct resources

{

int A; char B; int C; char D;

}resources;

int main()

{

struct sockaddr\_in sa; // Socket address data structure resources R;

int n, sockfd; // read and source

char buff[1025], obuff[256]; // buffer to store the read stream int snded, rec;

sockfd = socket(PF\_INET, SOCK\_STREAM, 0); // New socket created

// Checking for valid socket if (sockfd < 0)

{

printf("Error in creation\n"); exit(0);

}

else

printf("Socket created\n");

// Clearing and assigning type and address to the socket bzero(&sa, sizeof(sa));

sa.sin\_family = AF\_INET; sa.sin\_port = htons(8888);

// establishing and verifying the connection

if (connect(sockfd, (struct sockaddr\_in\*)&sa, sizeof(sa)) < 0)

{

printf("Connection failed\n"); exit(0);

}

else

printf("Connection made\n");

while (1)

{ snded = write(sockfd, "PING", 5); if (snded > -1)

printf("SENT PING\n"); rec = read(sockfd, obuff, 256); obuff[rec] = '\0';

if (strcmp(obuff, "PONG") == 0)

{

usleep(750); FILE \*f;

f = fopen("shared\_mem.txt", "r"); fread(&R, sizeof(R), 1, f); fclose(f);

printf("read %d, %d, %d, %d from server\n", R.A, R.B, R.C, R.D );

R.A += 1;

R.B += 1;

R.C += 1;

R.D += 1;

f = fopen("shared\_mem.txt", "w"); fwrite(&R, sizeof(R), 1, f); fclose(f);

printf("Got access to CS\n");

snded = write(sockfd, "DONE", 4); printf("Freeing Lock\n");

break;

}

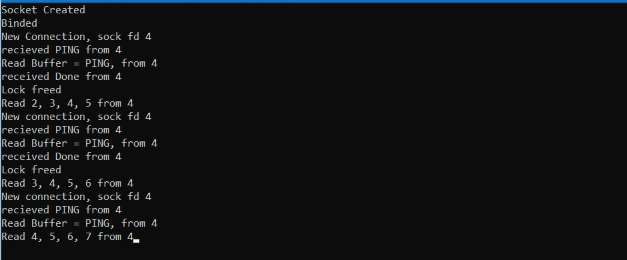
}

close(sockfd); // Closing the socket return 0;

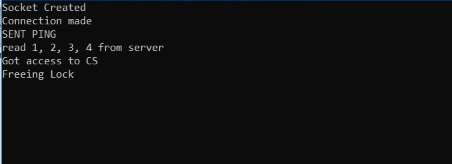
}

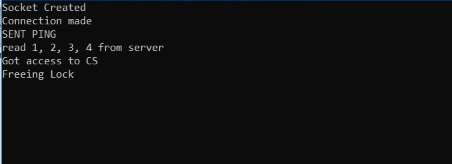
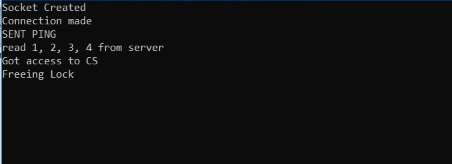
## OUTPUT

**CONTROLLER OUTPUT**



**CLIENT OUTPUTS**





**FINDING AND LEARNING**

1. We successfully implemented Centralized Mutual Exclusion.
2. When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access. The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic, or block later. Either way, when it sees the grant, it can enter the critical region.
3. 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).
4. The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release).
5. It can also be used for more general resource allocation rather than just managing critical regions.
6. The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down.
7. 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.
8. In a large system, a single coordinator can become a performance bottleneck. e-reading.club

**PROGRAM-4**

**AIM**: Implement Bully Election Algorithm

**THEORY**

**Election Algorithms**

Election algorithms choose a process from group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of coordinator should be restarted. Election algorithm assumes that every active process in the system has a unique priority number. The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number. Then, this number is sent to every active process in the distributed system.

**The Bully Election Process**

* 1. P sends a message to the coordinator.
  2. If coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.
  3. Now process P sends election message to every process with high priority number.
  4. It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.
  5. Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
  6. However, if an answer is received within time T from any other process Q,
     1. Process P again waits for time interval T’ to receive another message from Q that it has been elected as coordinator.
     2. If Q doesn’t respond within time interval T’ then it is assumed to have failed and algorithm is restarted.
* **Advantages**
  + The advantages of Bully algorithm are that this algorithm is a distributed method with simple implementation.
  + Only the processes with higher priority number respect to the priority number of process that detects the crash coordinator will be involved in election
* **Disadvantages**
  + A large number of messages are sent, this can overload the system.
  + There may be cases in very large systems that multiple coordinators get elected.

**CODE**

#include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <errno.h> #include <string.h> #include <sys/types.h> #include <time.h> #define MSG\_CONFIRM 0

#define TRUE 1

#define FALSE 0

#define ML 1024

#define MPROC 32

/\*

Function to create a new connection to port 'connect\_to'

1. Creates the socket.
2. Binds to port.
3. Returns socket id

\*/

int connect\_to\_port(int connect\_to)

{

int sock\_id; int opt = 1;

struct sockaddr\_in server;

if ((sock\_id = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0)

{

perror("unable to create a socket"); exit(EXIT\_FAILURE);

}

setsockopt(sock\_id, SOL\_SOCKET, SO\_REUSEADDR, (const void \*)&opt, sizeof(int)); memset(&server, 0, sizeof(server));

server.sin\_family = AF\_INET; server.sin\_addr.s\_addr = INADDR\_ANY; server.sin\_port = htons(connect\_to);

if (bind(sock\_id, (const struct sockaddr \*)&server, sizeof(server)) < 0)

{

perror("unable to bind to port"); exit(EXIT\_FAILURE);

}

return sock\_id;

}

/\*

sends a message to port id to

\*/

void send\_to\_id(int to, int id, char message[ML])

{

struct sockaddr\_in cl; memset(&cl, 0, sizeof(cl));

cl.sin\_family = AF\_INET; cl.sin\_addr.s\_addr = INADDR\_ANY; cl.sin\_port = htons(to);

sendto(id, \

(const char \*)message, \ strlen(message), \ MSG\_CONFIRM, \

(const struct sockaddr \*)&cl, \ sizeof(cl));

}

/\*

starts the election, returns 1 if it wins the round

\*/

int election(int id, int \*procs, int num\_procs, int self)

{

int itr;

char message[ML]; strcpy(message, "ELECTION");

int is\_new\_coord = 1; // assume you are the winner until you lose

for (itr = 0; itr < num\_procs; itr += 1)

{

if (procs[itr] > self)

{

printf("sending election to: %d\n", procs[itr]); send\_to\_id(procs[itr], id, message);

is\_new\_coord = 0; // a proc with id > self exists thus cannot be coord

}

}

return is\_new\_coord;

}

/\*

announces completion by sending coord messages

\*/

void announce\_completion(int id, int \*procs, int num\_procs, int self)

{

int itr;

char message[ML]; strcpy(message, "COORDINATOR");

for (itr = 0; itr < num\_procs; itr += 1) if (procs[itr] != self)

send\_to\_id(procs[itr], id, message);

}

int main(int argc, char\* argv[])

{

// 0. Initialize variables int self = atoi(argv[1]);

int n\_proc = atoi(argv[2]); int procs[MPROC];

int sock\_id, bully\_id; int itr, len, n, start\_at;

char buff[ML], message[ML]; struct sockaddr\_in from;

for (itr = 0; itr < n\_proc; itr += 1) procs[itr] = atoi(argv[3 + itr]);

start\_at = atoi(argv[3 + n\_proc]) == 1? TRUE : FALSE;

// 1. Create socket

printf("creating a node at %d %d \n", self, start\_at); sock\_id = connect\_to\_port(self);

// getchar();

// 2. check is process is initiator

if (start\_at == TRUE)

{

election(sock\_id, procs, n\_proc, self);

}

// 3. if not the initiator wait for someone else

while(TRUE)

{

memset(&from, 0, sizeof(from));

n = recvfrom(sock\_id, (char \*)buff, ML, MSG\_WAITALL, (struct sockaddr \*)&from, &len); buff[n] = '\0';

printf("Recieved messed: %s\n", buff);

if (!strcmp(buff, "ELECTION"))

{

strcpy(message, "E-ACK"); // send election ack sendto(sock\_id,

(const char \*)message, strlen(message),

MSG\_CONFIRM,

(const struct sockaddr \*)&from, sizeof(from));

if (election(sock\_id, procs, n\_proc, self))

{

announce\_completion(sock\_id, procs, n\_proc, self); printf("ANNOUNCING SELF AS NEW COORD\n");

}

}

else if (!strcmp(buff, "E-ACK"))

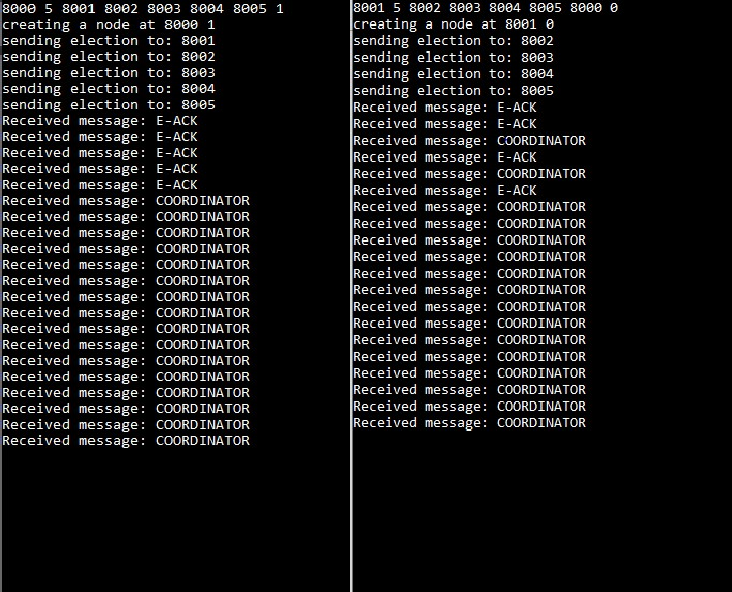
continue; // nothing do, your job is done else if (!strcmp(buff, "COORDINATOR"))

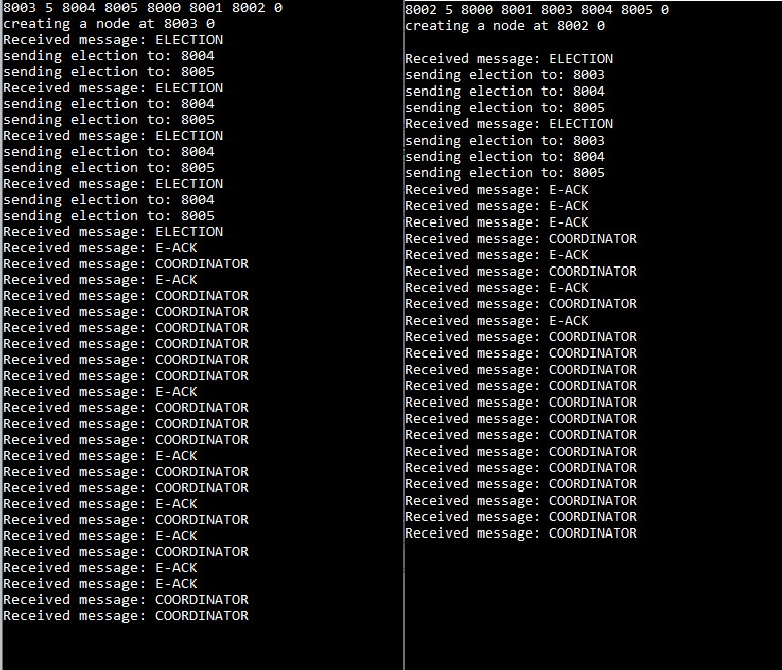
bully\_id = from.sin\_port;

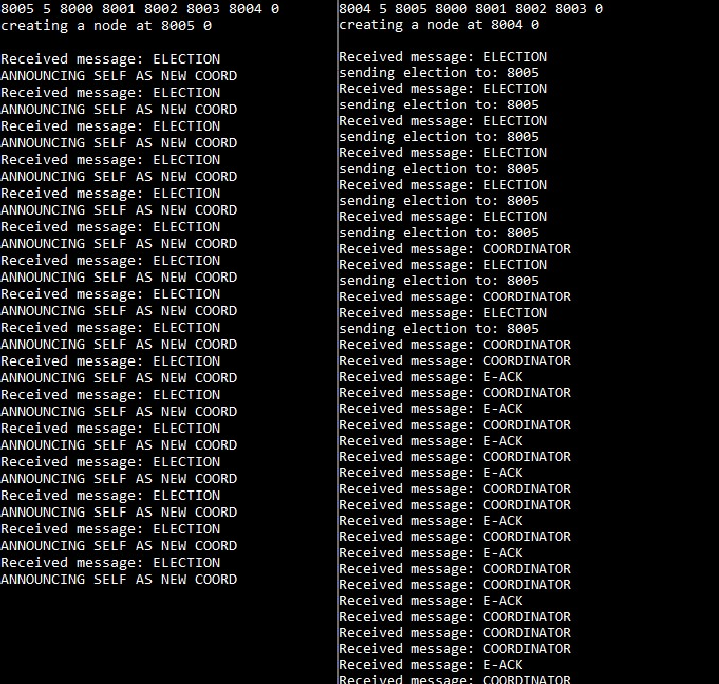
}

}

## OUTPUT







**FINDING AND LEARNING**

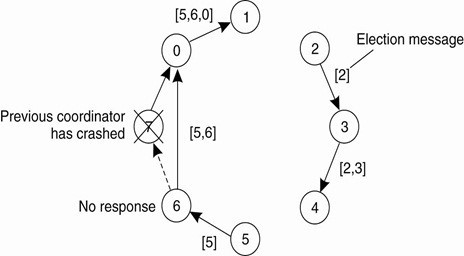
1. We successfully implemented Bully-Election Algorithm.
2. The **bully algorithm** is a method for dynamically electing a coordinator or leader from a group of distributed computer processes.
3. The process with the highest process ID number from amongst the non-failed processes is selected as the coordinator.
4. 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.

# PROGRAM-5

### **AIM**: Implement Ring Election Algorithm

**THEORY**

Another election algorithm is based on the use of a ring, but without a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message. Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.



**CODE**

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

#include <sys/types.h> #include <time.h> #define MSG\_CONFIRM 0

#define TRUE 1

#define FALSE 0

#define ML 1024

#define MPROC 32

char \* get\_slice(char str[], int s, int e)

{

char ans[ML]; int i;

for (i = s; i < e && i <strlen(str); i++)

{

printf("Read %c", str[s+i]); ans[i] = str[s+i];

}

ans[i] = '\0'; return ans;

}

char \* get\_max( char str[ML], int n)

{

char \*temp;

temp = strndup(str, 4); printf("%s\n", temp); char \*t2;

int i;

for(i = 4; i<n; i += 4)

{

if (i > n) break;

t2 = strndup(str+i, 4); printf("%s\n", t2);

if (strcmp(t2, temp)){

temp = t2;

printf("big %s\n", temp);

}

}

temp[4] = '\0'; printf("max: %s\n", temp); return temp;

}

void append\_message(char str[], int id)

{

char temp[6]; sprintf(temp, "%d ", id); strcat(str, temp);

}

/\*

Function to create a new connection to port 'connect\_to'

* 1. Creates the socket.
  2. Binds to port.
  3. Returns socket id

\*/

int connect\_to\_port(int connect\_to)

{

int sock\_id;

int opt = 1;

struct sockaddr\_in server;

if ((sock\_id = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0)

{

perror("unable to create a socket"); exit(EXIT\_FAILURE);

}

setsockopt(sock\_id, SOL\_SOCKET, SO\_REUSEADDR, (const void \*)&opt, sizeof(int)); memset(&server, 0, sizeof(server));

server.sin\_family = AF\_INET; server.sin\_addr.s\_addr = INADDR\_ANY; server.sin\_port = htons(connect\_to);

if (bind(sock\_id, (const struct sockaddr \*)&server, sizeof(server)) < 0)

{

perror("unable to bind to port"); exit(EXIT\_FAILURE);

}

return sock\_id;

}

/\*

sends a message to port id to

\*/

void send\_to\_id(int to, int id, char message[ML])

{

struct sockaddr\_in cl; memset(&cl, 0, sizeof(cl));

cl.sin\_family = AF\_INET;

cl.sin\_addr.s\_addr = INADDR\_ANY; cl.sin\_port = htons(to);

int n = sendto(id, \

(const char \*)message, \ strlen(message), \ MSG\_CONFIRM, \

(const struct sockaddr \*)&cl, \ sizeof(cl));

printf("sent packet %d\n", n);

}

/\*

starts the election, returns 1 if it wins the round

\*/

void election(int id, int to, int self)

{

char message[ML]; sprintf(message, "%d", self);

printf("sending election to: %d\n", to); send\_to\_id(to, id, message);

}

/\*

announces completion by sending coord messages

\*/

void announce\_completion(int id, int \*procs, int num\_procs, int self)

{

int itr;

char message[ML]; strcpy(message, "COORDINATOR");

for (itr = 0; itr < num\_procs; itr += 1) if (procs[itr] != self)

send\_to\_id(procs[itr], id, message);

}

int main(int argc, char\* argv[])

{

// 0. Initialize variables int self = atoi(argv[1]);

char \*sself = argv[1]; char \*coord = "9999";

int next = atoi(argv[2]); int sock\_id, win\_id;

int itr, len, n, start\_at;

char buff[ML], message[ML], \*id; struct sockaddr\_in from;

start\_at = atoi(argv[3]);

// 1. Create socket

printf("creating a node at %d %d \n", self, start\_at); sock\_id = connect\_to\_port(self);

// getchar();

// 2. check is process is initiator

if (start\_at == TRUE)

{

election(sock\_id, next, self);

}

// 3. if not the initiator wait for someone else

while(TRUE)

{

memset(&from, 0, sizeof(from));

n = recvfrom(sock\_id, (char \*)buff, ML, MSG\_WAITALL, (struct sockaddr \*)&from, &len);

buff[n] = '\0';

printf("Recieved message: %s %d\n", buff, strlen(buff)); if (buff[0] == '0')

{

id = strndup(buff+1, 4); printf("%s\n", id);

if (strcmp(sself, id) == 0)

{

printf("SET SELF = COORDINATOR\n");

strcpy(message, "COORDINATOR"); send\_to\_id(next, sock\_id, message); exit(EXIT\_SUCCESS);

}

else

{

send\_to\_id(next, sock\_id, buff);

}

}

else

{

id = strndup(buff, 4); printf("Init by: %s\n", id); if (strcmp(id, sself) == 0)

{

printf("Election polling complete\n"); coord = get\_max(buff, strlen(buff)); sprintf(message, "0%s", coord);

// printf("%s\n", coord); send\_to\_id(next, sock\_id, message);

}

else if(strcmp(buff, "COORDINATOR") == 0)

{

printf("GOT COORD ACK, EXITING ELECTION\n");

send\_to\_id(next, sock\_id, buff); exit(EXIT\_SUCCESS);

}

else

{

printf("Sending to next: %d, %s\n", next, buff); strcat(buff, sself);

// printf("message: %s\n", buff); send\_to\_id(next, sock\_id, buff);

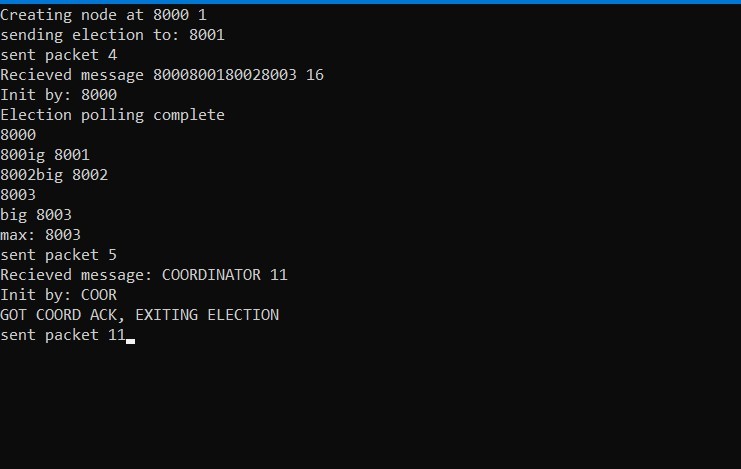
}

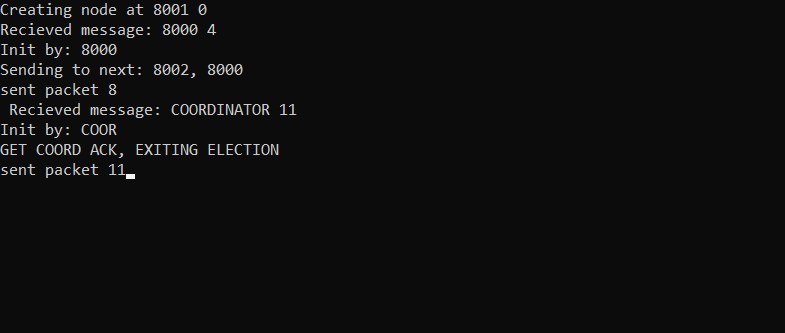
}

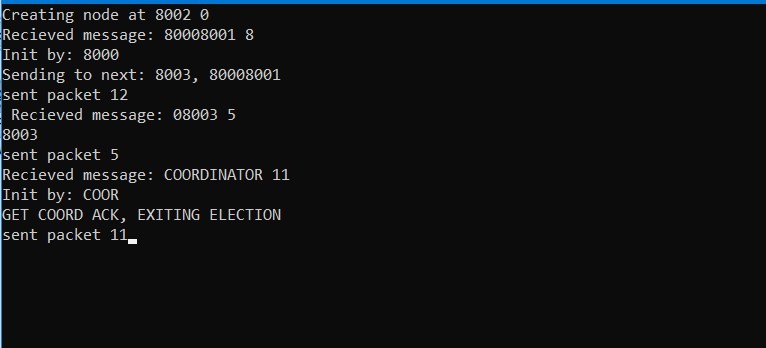
}

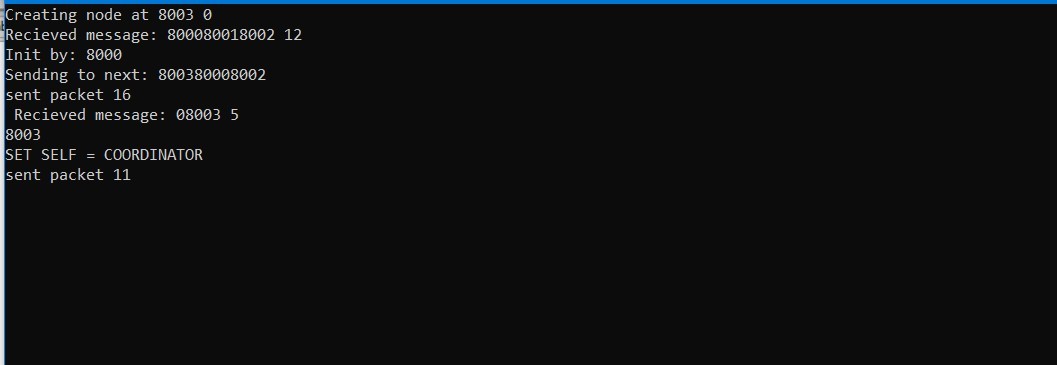
}

**OUTPUT**









**FINDING AND LEARNING**

1. We successfully implemented Ring Election Algorithm.
2. This algorithm applies to systems organized as a ring(logically or physically).
3. In this algorithm we assume that the link between the process are unidirectional and every process can message to the process on its right only.
4. Data structure that this algorithm uses is **active list**, a list that has priority number of all active processes in the system.

**PROGRAM-6**

**AIM**: To implement 2-Phase Commit client-server.

**THEORY:**

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. However, it is not resilient to all possible failure configurations, and in rare cases, manual intervention is needed to remedy an outcome. To accommodate recovery from failure (automatic in most cases) the protocol's participants use logging of the protocol's states. Log records, which are typically slow to generate but survive failures, are used by the protocol's recovery procedures. Many protocol variants exist that primarily differ in logging strategies and recovery mechanisms. Though usually intended to be used infrequently, recovery procedures compose a substantial portion of the protocol, due to many possible failure scenarios to be considered and supported by the protocol.

In a "normal execution" of any single distributed transaction (i.e., when no failure occurs, which is typically the most frequent situation), the protocol consists of two phases:

1. The *commit-request phase* (or *voting phase*), in which a *coordinator* process attempts to prepare all the transaction's participating processes (named *participants*, *cohorts*, or *workers*) to take the necessary steps for either committing or aborting the transaction and to *vote*, either "Yes": commit (if the transaction participant's local portion execution has ended properly), or "No": abort (if a problem has been detected with the local portion), and
2. The *commit phase*, in which, based on *voting* of the participants, the coordinator decides whether to commit (only if *all* have voted "Yes") or abort the transaction (otherwise), and notifies the result to all the participants. The participants then follow with the needed actions (commit or abort) with their local transactional resources (also called *recoverable resources*; e.g., database data) and their respective portions in the transaction's other output (if applicable).

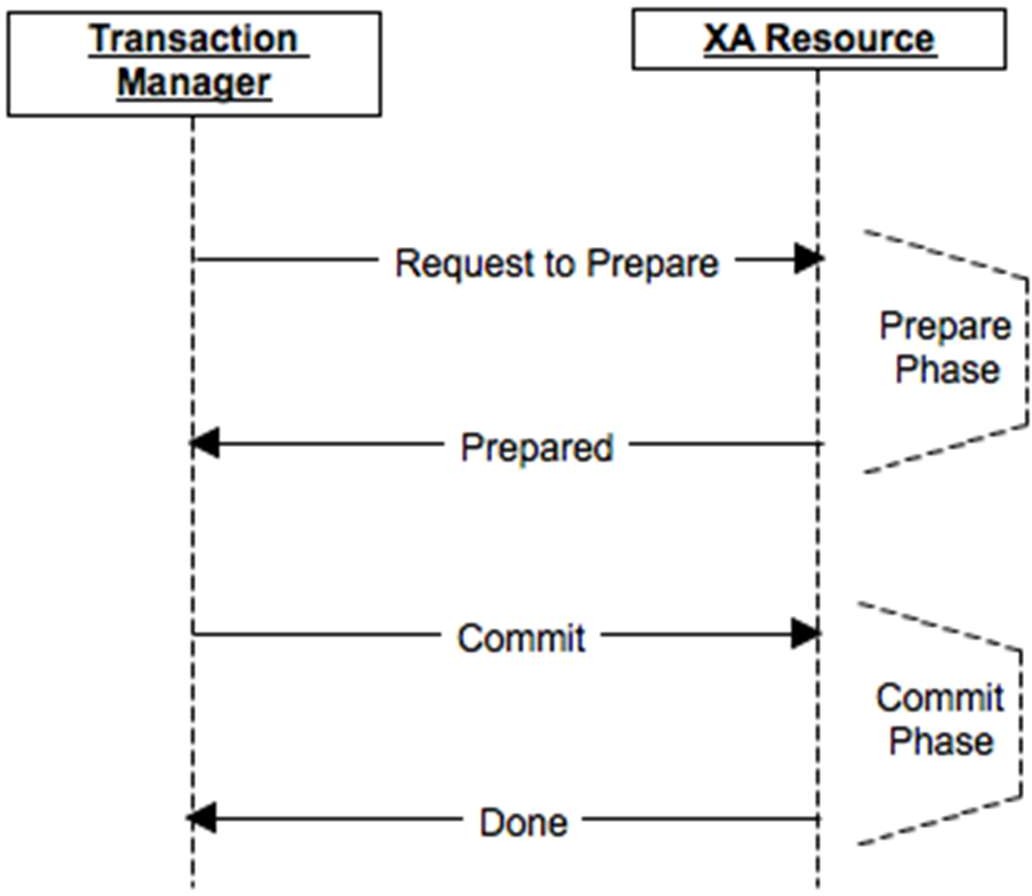


Fig : Two Phase Commit Protocol

## CLIENT CODE

#include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <errno.h> #include <string.h> #include <sys/types.h> #include <time.h> #include <string.h> #define MSG\_CONFIRM 0

#define TRUE 1

#define FALSE 0

#define ML 1024

#define MPROC 32

typedef struct wireless\_node

{

int priority; int parent;

} wireless\_node;

wireless\_node w;

int max(int a, int b)

{

return a >= b? a:b;

}

int connect\_to\_port(int connect\_to)

{

int sock\_id; int opt = 1;

struct sockaddr\_in server;

if ((sock\_id = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0)

{

perror("unable to create a socket"); exit(EXIT\_FAILURE);

}

setsockopt(sock\_id, SOL\_SOCKET, SO\_REUSEADDR, (const void \*)&opt, sizeof(int)); memset(&server, 0, sizeof(server));

server.sin\_family = AF\_INET; server.sin\_addr.s\_addr = INADDR\_ANY; server.sin\_port = htons(connect\_to);

if (bind(sock\_id, (const struct sockaddr \*)&server, sizeof(server)) < 0)

{

perror("unable to bind to port"); exit(EXIT\_FAILURE);

}

return sock\_id;

}

void send\_to\_id(int to, int from, char message[ML])

{

struct sockaddr\_in cl; memset(&cl, 0, sizeof(cl));

cl.sin\_family = AF\_INET; cl.sin\_addr.s\_addr = INADDR\_ANY; cl.sin\_port = htons(to);

sendto( from, \

(const char \*)message, \ strlen(message), \ MSG\_CONFIRM, \

(const struct sockaddr \*)&cl, \ sizeof(cl));

}

void begin\_commit(int id, int \*procs, int num\_procs)

{

int itr;

char message[ML]; sprintf(message, "%s", "SCMT"); for (itr = 0; itr < num\_procs; itr++)

{

printf("Sending begin commit to: %d\n", procs[itr]); send\_to\_id(procs[itr], id, message);

}

}

void announce\_action(int self, int \*procs, int num\_procs, char msg[ML])

{

int itr;

for (itr = 0; itr < num\_procs; itr++)

{

send\_to\_id(procs[itr], self, msg);

}

}

int main(int argc, char\* argv[])

{

int self = atoi(argv[1]); int server = atoi(argv[2]); char \*action = argv[3];

int sender, okcnt = 0, nocnt = 0, dncnt = 0; int sock\_id, coord\_id;

int itr, len, n, start, ix;

char buffer[ML], flag[ML], p\_id[ML], msg[256];

struct sockaddr\_in from;

printf("Creating node at %d\n", self);

sock\_id = connect\_to\_port(self);

while(TRUE)

{

sleep(2);

memset(&from, 0, sizeof(from));

// printf("Tring read\n");

n = recvfrom(sock\_id, (char \*)buffer, ML, MSG\_WAITALL, (struct sockaddr \*)&from, &len); buffer[n] = '\0';

printf("Recieved: %s\n", buffer);

if (strcmp(buffer, "SCMT") == 0)

{

printf("Sending %s to server\n", action); send\_to\_id(server, sock\_id, action);

}

else if (strcmp(buffer, "CDON") == 0)

{

printf("Got complete commit, committing to logs\n"); send\_to\_id(server, sock\_id, "DONE"); exit(EXIT\_FAILURE);

}

else if (strcmp(buffer, "CABT") == 0)

{

printf("Got abort commit, deleting updates\n"); send\_to\_id(server, sock\_id, "DONE"); exit(EXIT\_FAILURE);

}

// printf("Waiting\n");

}

return 0;

}

## SERVER CODE

#include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <errno.h> #include <string.h> #include <sys/types.h> #include <time.h> #include <string.h> #define MSG\_CONFIRM 0

#define TRUE 1

#define FALSE 0

#define ML 1024

#define MPROC 32

typedef struct wireless\_node

{

int priority; int parent;

} wireless\_node;

wireless\_node w;

int max(int a, int b)

{

return a >= b? a:b;

}

int connect\_to\_port(int connect\_to)

{

int sock\_id; int opt = 1;

struct sockaddr\_in server;

if ((sock\_id = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0)

{

perror("unable to create a socket"); exit(EXIT\_FAILURE);

}

setsockopt(sock\_id, SOL\_SOCKET, SO\_REUSEADDR, (const void \*)&opt, sizeof(int)); memset(&server, 0, sizeof(server));

server.sin\_family = AF\_INET; server.sin\_addr.s\_addr = INADDR\_ANY; server.sin\_port = htons(connect\_to);

if (bind(sock\_id, (const struct sockaddr \*)&server, sizeof(server)) < 0)

{

perror("unable to bind to port"); exit(EXIT\_FAILURE);

}

return sock\_id;

}

void send\_to\_id(int to, int from, char message[ML])

{

struct sockaddr\_in cl; memset(&cl, 0, sizeof(cl));

cl.sin\_family = AF\_INET; cl.sin\_addr.s\_addr = INADDR\_ANY; cl.sin\_port = htons(to);

sendto( from, \

(const char \*)message, \ strlen(message), \ MSG\_CONFIRM, \

(const struct sockaddr \*)&cl, \ sizeof(cl));

}

void begin\_commit(int id, int \*procs, int num\_procs)

{

int itr;

char message[ML]; sprintf(message, "%s", "SCMT"); for (itr = 0; itr < num\_procs; itr++)

{

printf("Sending begin commit to: %d\n", procs[itr]); send\_to\_id(procs[itr], id, message);

}

}

void announce\_action(int self, int \*procs, int num\_procs, char msg[ML])

{

int itr;

for (itr = 0; itr < num\_procs; itr++)

{

send\_to\_id(procs[itr], self, msg);

}

}

int main(int argc, char\* argv[])

{

int self = atoi(argv[1]);

int n\_procs = atoi(argv[2]); int procs[MPROC];

int sender, okcnt = 0, nocnt = 0, dncnt = 0; int sock\_id, coord\_id;

int itr, len, n, start, ix;

char buffer[ML], flag[ML], p\_id[ML], msg[256];

struct sockaddr\_in from;

for(itr = 0; itr < n\_procs; itr += 1) procs[itr] = atoi(argv[3 + itr]);

printf("Creating node at %d\n", self); sock\_id = connect\_to\_port(self); begin\_commit(sock\_id, procs, n\_procs);

while(TRUE)

{

sleep(2);

memset(&from, 0, sizeof(from));

// printf("Tring read\n");

n = recvfrom(sock\_id, (char \*)buffer, ML, MSG\_WAITALL, (struct sockaddr \*)&from, &len); buffer[n] = '\0';

printf("Recieved: %s\n", buffer);

if (strcmp(buffer, "CMOK") == 0)

{

okcnt += 1;

}

else if (strcmp(buffer, "CMNO") == 0)

{

nocnt += 1;

}

if ((nocnt + okcnt) == n\_procs)

{

printf("Recieved replies from all clients\n"); if (okcnt == n\_procs)

{

printf("Announcing complete commit\n"); announce\_action(sock\_id, procs, n\_procs, "CDON");

}

else

{

printf("Announcing abort commit\n"); announce\_action(sock\_id, procs, n\_procs, "CABT");

}

}

if (strcmp(buffer, "DONE") == 0)

{

dncnt += 1;

printf("clients confirmed commit\n"); if (dncnt == n\_procs)

{

printf("All process announced commit action\n"); exit(EXIT\_SUCCESS);

}

}

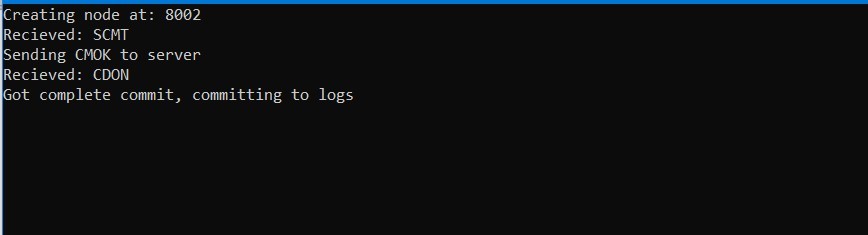
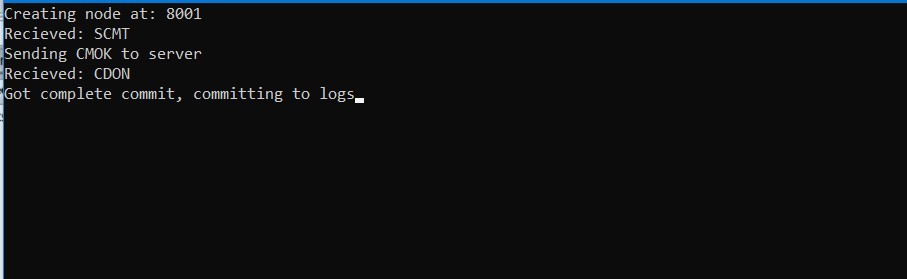
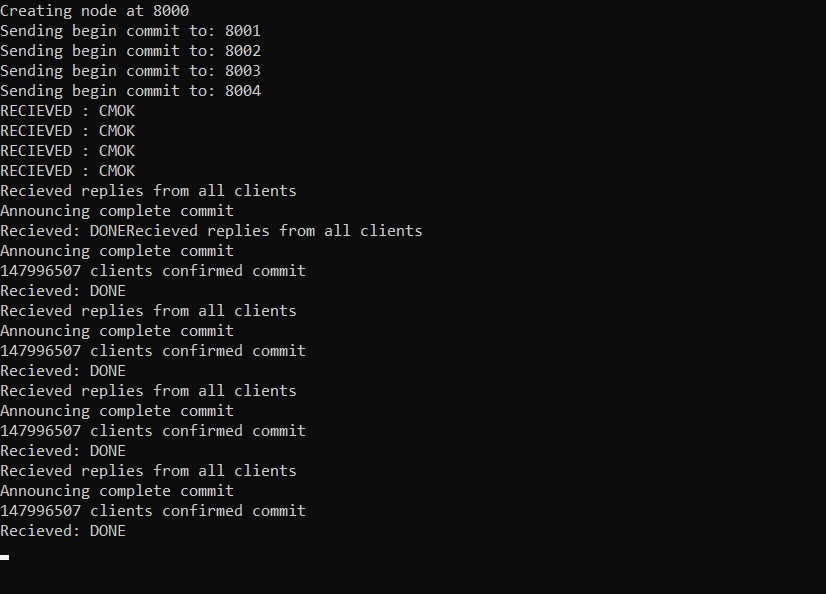
// printf("Waiting\n");

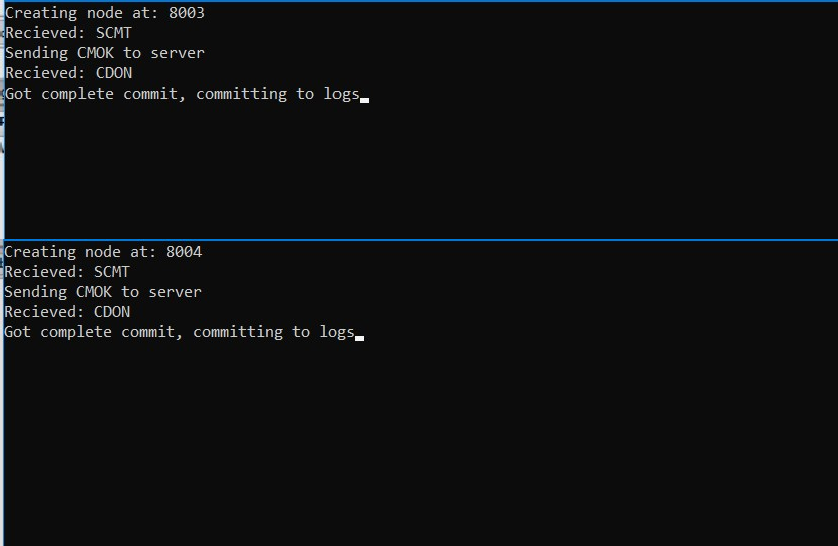
}

return 0;

}

## OUTPUT





**FINDING AND LEARNING**

* 1. We successfully implemented 2-phase commit.
  2. The greatest disadvantage of the two-phase commit protocol is that it is a blocking protocol. If the coordinator fails permanently, some participants will never resolve their transactions: After a participant has sent an **agreement** message to the coordinator, it will block until a **commit** or **rollback** is received.

# PROGRAM-7

### **AIM**: To implement 3-Phase Commit client-server.

**THEORY:**

In computer networking and databases, the three-phase commit protocol (3PC) is a distributed algorithm which lets all nodes in a distributed system agree to commit a transaction. It is a refinement of the two-phase commit protocol (2PC) which is more resilient to failures.

A two-phase commit protocol cannot dependably recover from a failure of both the coordinator and a cohort member during the Commit phase. If only the coordinator had failed, and no cohort members had received a commit message, it could safely be inferred that no commit had happened. If, however, both the coordinator and a cohort member failed, it is possible that the failed cohort member was the first to be notified, and had actually done the commit. Even if a new coordinator is selected, it cannot confidently proceed with the operation until it has received an agreement from all cohort members, and hence must block until all cohort members respond.

The three-phase commit protocol eliminates this problem by introducing the Prepared to commit state. If the coordinator fails before sending preCommit messages, the cohort will unanimously agree that the operation was aborted. The coordinator will not send out a doCommit message until all cohort members have ACKed that they are Prepared to commit. This eliminates the possibility that any cohort member actually completed the transaction before all cohort members were aware of the decision to do so (an ambiguity that necessitated indefinite blocking in the two-phase commit protocol).

**ALGORITHM CODE**

**OUTPUT**

**FINDING AND LEARNING**

1. We implemented the 3-Phase commit protocol.
2. **Disadvantage:**The protocol requires at least three round trips to complete, needing a minimum of three round trip times (RTTs). This is potentially a long latency to complete each transaction.