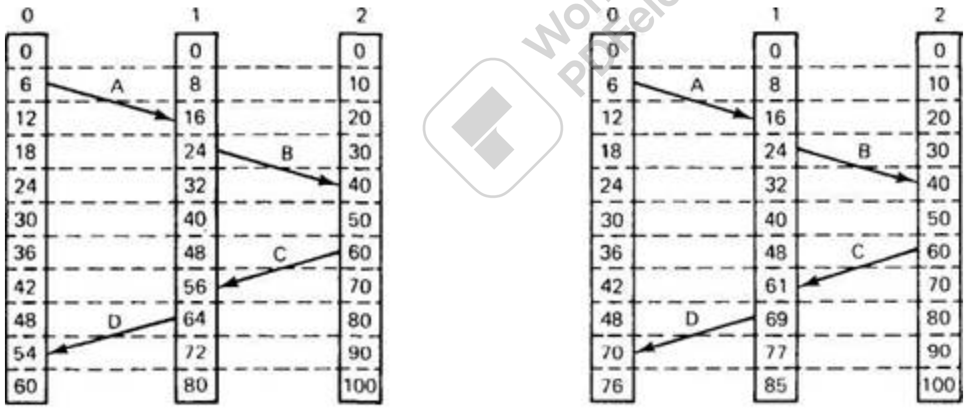
Program – 2

AIM: Implement Lamport Clock Synchronization

Introduction and Theory

The algorithm of Lamport timestamps is a simple algorithm used to determine the order of events in a distributed computer system. As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead, and conceptually provide a starting point for the more advanced vector clock method. They are named after their creator, Leslie Lamport. Distributed algorithms such as resource synchronization often depend on some method of ordering events to function. For example, consider a system with two processes and a disk. The processes send messages to each other, and also send messages to the disk requesting access. The disk grants access in the order the messages were sent.

For example process A sends a message to the disk requesting write access, and then sends a read instruction message to process B. Process B receives the message, and as a result sends its own read request message to the disk. If there is a timing delay causing the disk to receive both messages at the same time, it can determine which 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.



(a) (b)

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.

The algorithm follows some simple rules:

1. A process increments its counter before each event in that process;

Program – 2

2. When a process sends a message, it includes its counter value with the message;

3. On receiving a message, the counter of the recipient is updated, if necessary, to the greater of its current counter and the timestamp in the received message. The counter is then incremented by 1 before the message is considered received.

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;
```

Program – 2

```
    }
    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
*/
```

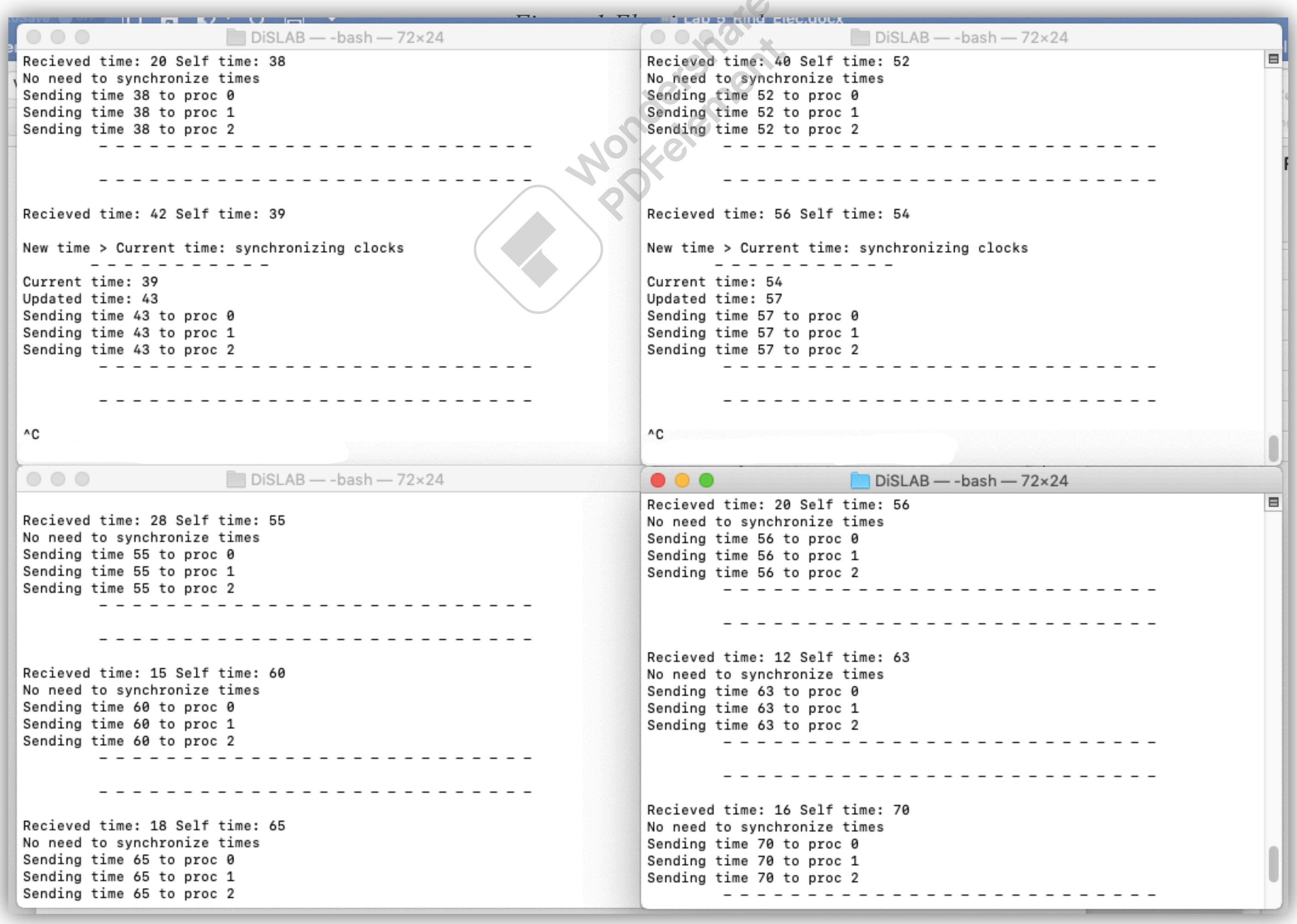
Program – 2

```
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);\
```

Program – 2

```
            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");
       }
}
```

Results and Outputs:



Program – 2

Findings and Learnings:

1. We successfully implemented Lamport Clock .