# Program – 6

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)
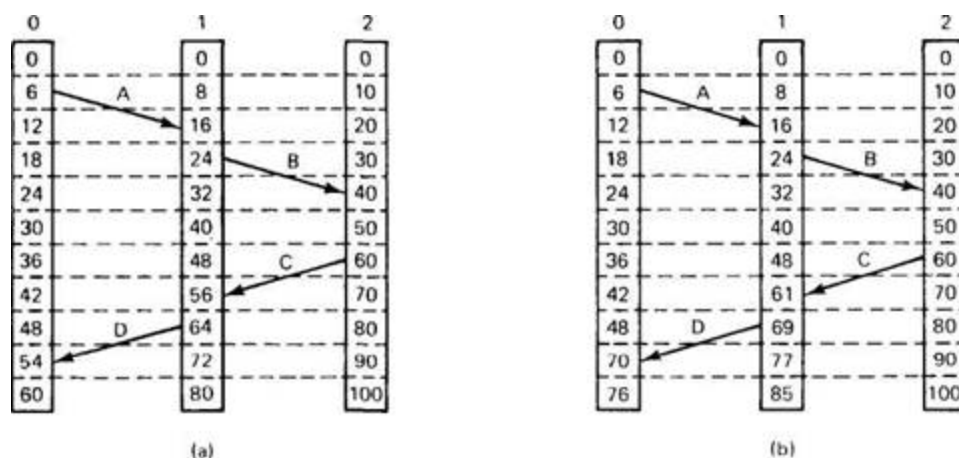
Lamport invented a simple mechanism by which the happened-before ordering can be captured numerically. A Lamport logical clock is an incrementing software counter maintained in each process.

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 – 6

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

```
1   #include <sys/socket.h>
2   #include <netinet/in.h>
3   #include <arpa/inet.h>
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <unistd.h>
7   #include <errno.h>
8   #include <string.h>
9   #include <sys/types.h>
10  #include <time.h>
11  #define MSG_CONFIRM 0
12  #define TRUE 1
13  #define FALSE 0
14  #define ML 1024
15  #define MPROC 32
16
17  /*
18          Function to create a new connection to port 'connect_to'
19          1. Creates the socket.
20          2. Binds to port.
21          3. Returns socket id
22  */
23
24  typedef struct lamport_clock{
25      int timer;
26  }lamport_clock;
27
28
29  void init(lamport_clock *clk)
30  {
31      clk->timer = 0;
32  }
33
34  void tick(lamport_clock *clk, int phase)
35  {
36      clk->timer += phase;
37  }
38
39  int str_to_int(char str[ML], int n)
40  {
41      int x = 0, i = 0, k;
42      printf("x: %d\n", x);
43      for (i = 0; i < n; i++)
44      {
45          k = atoi(str[i]);
46          x = x*10 + k;
```

Program – 6

```
47        }
48        return x;
49 }
50
51 void update_clock(lamport_clock *clk, int new_time)
52 {
53        clk->timer = new_time;
54 }
55
56 int connect_to_port(int connect_to)
57 {
58        int sock_id;
59        int opt = 1;
60        struct sockaddr_in server;
61        if ((sock_id = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
62        {
63                perror("unable to create a socket");
64                exit(EXIT_FAILURE);
65        }
66        setsockopt(sock_id, SOL_SOCKET, SO_REUSEADDR, (const void
67 *)&opt, sizeof(int));
68        memset(&server, 0, sizeof(server));
69        server.sin_family = AF_INET;
70        server.sin_addr.s_addr = INADDR_ANY;
71        server.sin_port = htons(connect_to);
72
73        if (bind(sock_id, (const struct sockaddr *)&server,
74 sizeof(server)) < 0)
75        {
76                perror("unable to bind to port");
77                exit(EXIT_FAILURE);
78        }
79        return sock_id;
80 }
81 /*
82        sends a message to port id to
83 */
84 void send_to_id(int to, int id, lamport_clock clk)
85 {
86        struct sockaddr_in cl;
87        memset(&cl, 0, sizeof(cl));
88     char message[ML];
89     sprintf(message, "%d", clk.timer);
90        cl.sin_family = AF_INET;
91        cl.sin_addr.s_addr = INADDR_ANY;
92        cl.sin_port = htons(to);
93
94        sendto(id, \
95                (const char *)message, \
96                strlen(message), \
97                MSG_CONFIRM, \
98                (const struct sockaddr *)&cl, \
99                sizeof(cl));
100 }
101 /*
102        announces completion by sending coord messages
103 */
```

Program – 6

```
104  int main(int argc, char* argv[])
105  {
106          // 0. Initialize variables
107      int self = atoi(argv[1]);
108          int n_proc = atoi(argv[2]);
109      int phase = atoi(argv[3]);
110          int procs[MPROC];
111          int sock_id;
112      int new_time;
113          int itr, len, n, start_at;
114          char buff[ML], message[ML];
115          struct sockaddr_in from;
116      lamport_clock self_clock;
117
118          for (itr = 0; itr < n_proc; itr += 1)
119                  procs[itr] = atoi(argv[4 + itr]);
120
121          start_at = atoi(argv[4 + n_proc]) == 1? TRUE : FALSE;
122      init(&self_clock);
123      tick(&self_clock, phase);
124          // 1. Create socket
125          printf("creating a node at %d %d \n", self, start_at);
126          sock_id = connect_to_port(self);
127          // getchar();
128          // 2. check is process is initiator
129      if (start_at ==  TRUE)
130      {
131          printf("Proc %d is starting comms \n", self);
132          for (itr = 0; itr < n_proc; itr++)
133          {
134              printf("Sending to proc: %d", itr);
135              send_to_id(procs[itr], sock_id, self_clock);
136          }
137      }
138          // 3. if not the initiator wait for someone else
139          while(TRUE)
140          {
141          printf("\t - - - - - - - - - - - - - - - - - - - - - - - -
142  -\n\n");
143          sleep(1);
144          tick(&self_clock, phase);
145
146                  memset(&from, 0, sizeof(from));
147                  n = recvfrom(sock_id, (char *)buff, ML, MSG_WAITALL,
148  (struct sockaddr *)&from, &len);
149
150                  buff[n] = '\0';
151
152                  printf("Recieved time: %s Self time: %d\n", buff,
153  self_clock.timer);
154          new_time = atoi(buff);
155          // printf("Recieved time: %s %d\n", buff, new_time);
156          if (new_time > self_clock.timer)
157          {
158              printf("\nNew time > Current time: synchronizing
159  clocks\n\t- - - - - - - - - -\n");
160              printf("Current time: %d\n", self_clock.timer);\
```

Program – 6

```
161              printf("Updated time: %d\n", new_time + 1);
162              update_clock(&self_clock, new_time + 1);
163          }
164          else
165          {
166              printf("No need to synchronize times\n");
167          }
168          for (itr = 0; itr < n_proc; itr++)
169          {
170              printf("Sending time %d to proc %d\n", self_clock.timer,
171  itr);
172              send_to_id(procs[itr], sock_id, self_clock);
173          }
174          printf("\t - - - - - - - - - - - - - - - - - - - - - - - -
175  -\n\n");
176          }
177  }
178
```

Results and Outputs:

```
●○○            DiSLAB — -bash — 72×24
Recieved time: 20 Self time: 38
No need to synchronize times
Sending time 38 to proc 0
Sending time 38 to proc 1
Sending time 38 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - - - -

          - - - - - - - - - - - - - - - - - - - - - - - - - -

Recieved time: 42 Self time: 39

New time > Current time: synchronizing clocks
          - - - - - - - - - - -
Current time: 39
Updated time: 43
Sending time 43 to proc 0
Sending time 43 to proc 1
Sending time 43 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - -

          - - - - - - - - - - - - - - - - - - - - - - - -

^C
```

```
●○○            DiSLAB — -bash — 72×24
Recieved time: 40 Self time: 52
No need to synchronize times
Sending time 52 to proc 0
Sending time 52 to proc 1
Sending time 52 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - - - -

          - - - - - - - - - - - - - - - - - - - - - - - - - -

Recieved time: 56 Self time: 54

New time > Current time: synchronizing clocks
          - - - - - - - - - - -
Current time: 54
Updated time: 57
Sending time 57 to proc 0
Sending time 57 to proc 1
Sending time 57 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - -

          - - - - - - - - - - - - - - - - - - - - - - - -

^C
```

```
●○○            DiSLAB — -bash — 72×24

Recieved time: 28 Self time: 55
No need to synchronize times
Sending time 55 to proc 0
Sending time 55 to proc 1
Sending time 55 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - -

          - - - - - - - - - - - - - - - - - - - - - - - -

Recieved time: 15 Self time: 60
No need to synchronize times
Sending time 60 to proc 0
Sending time 60 to proc 1
Sending time 60 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - -

          - - - - - - - - - - - - - - - - - - - - - - - -

Recieved time: 18 Self time: 65
No need to synchronize times
Sending time 65 to proc 0
Sending time 65 to proc 1
Sending time 65 to proc 2
```

```
●●●            DiSLAB — -bash — 72×24
Recieved time: 20 Self time: 56
No need to synchronize times
Sending time 56 to proc 0
Sending time 56 to proc 1
Sending time 56 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - - -

          - - - - - - - - - - - - - - - - - - - - - - - - -

Recieved time: 12 Self time: 63
No need to synchronize times
Sending time 63 to proc 0
Sending time 63 to proc 1
Sending time 63 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - - -

          - - - - - - - - - - - - - - - - - - - - - - - - -

Recieved time: 16 Self time: 70
No need to synchronize times
Sending time 70 to proc 0
Sending time 70 to proc 1
Sending time 70 to proc 2
          - - - - - - - - - - - - - - - - - - - - - - - - -
```

Program – 6

Findings and Learnings:

1. We successfully implemented Lamport Clock .