Cloud Computing Assignment

By Thadicherla Hrishith(20CS02002)

Mallela Sathvik(20CS01061)

P.R.R Sumiran Adithya(20CS01014)

Project Statement:

The project requires implementation of Lamport's Mutual exclusion using logical clock. The project has to be implemented via sockets because you need to communicate between 3 devices. The critical section can be assumed to be access to a file on any one of the 3 machines. There need to be two threads per device. One thread shall create local event and send event; the send event should only be for mutual exclusion request. The other thread need to act as port listener and hence, receiver thread. The only catch here is to synchronise between sender and receiver thread to update the logical clock. The implementation needs to be done only in C++ or Java.

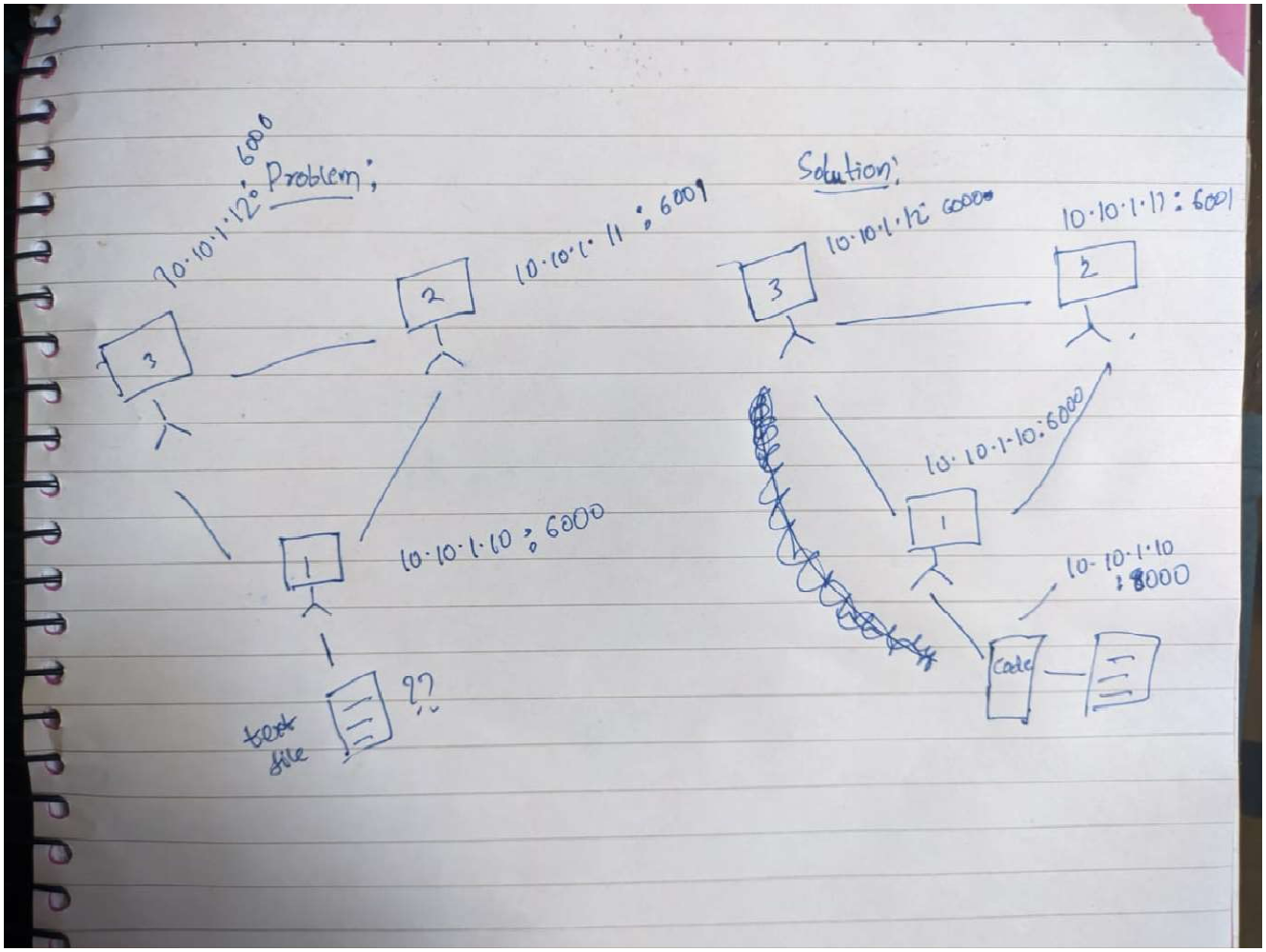
Approach to Solution:

The first part of this problem is to create the sender and listener threads. My approach to this was that don't need to keep the sender thread always on. We can just create the sender thread whenever are are making the request and just send it to the 2 other clients. However the listener thread needs to be active all the time. So we activate it when we first take in the ip and port.

The second part of this problem is how do we access the file hosted on one system from another system.

One approach is to ask the process whose system is hosting it to edit it, but it seems wrong because we are giving one process on the system direct access to the data of the other process in the system. Also, the file access needs to be fair, we can understand why the code to access the file shouldn't be in the main code.

Other approach is creating another process on this same system which has edit access to this file, attach a socket to it and then let all the systems access this process through mutual exclusion. The access is only possible since we attached a socket to it.



File called CriticalSection.java is the code which has access to the file.

Next issue is synchronisation of clocks , queues and replies.

The way we can do that is by usage of locks. This way we can make sure that only one thread sender or listener is accessing these sections at the same time.

We will create a timeStamp Lock, PriorityqueueLock and replyLock and surround it whenever we need to update timeStamp or insert into Priorityqueue or reset or increment the replies.

That's pretty much the end of our issues outside the logic of the algorithm . All we need to do is see the issues inside Lamport's Mutual Exclusion.

The Algorithm as in the text book Advanced Operating Systems by Shivratri.

Requesting the critical section:

1. When a site Si wants to enter the CS, it sends a REQUEST(ts{i}, i) message to all the sites in its request set R_{i} and places the request on request queue. ((ts{i}, i) is the timestamp of the request.)

2. When a site S_{j} receives the REQUEST(ts{i}, i) message from site S_{i} it returns a timestamped REPLY message S_{i} and places site S_{i}' s request on request queues

Executing the critical section: Site S, enters the CS when the two following conditions hold.

[L1:] S_{i} has received a message with timestamp larger than (ts_{i}, i) from all other sites.

[L2:] S_{i} ‘s request is at the top of request queue

Releasing the critical section:

3. Site S_{i} upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.

4. When a site S_{j} receives a RELEASE message from site S_{i} it removes S_{i}’ s request from its request queue.

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

The messages are send in the format :

TypeofMessage_Timestamp@Ip:Port

From this we can easily decode the message by using split in java.

So the listener thread receives all kinds of messages REQUEST, REPLY and RESPONSE.

Sender thread can create the Event and send requests.

The main issue we have after synchronising everything with locks is

Accessing the critical section:

The way we implement that is as follows:

1. If we are on the top of queue then it is straight forward. We should access the critical section .So other systems with check their queues send the replies and it is done.
2. If we are not on top of queue then it becomes tricky. In that case where some other pc is accessing CS. When it sends the release message. What we need to do is pop the queue. Now you have two cases.
    1. If after popping we are on top of queue then we are not supposed to do anything. We have to wait for the one who just released the CS to send a reply and the other pc should check the queue and send the reply
    2. If after popping we are not on top of queue then we are supposed to send a reply to whoever is on the top of the queue.

After this the logic is pretty straight forward and explained in the code comments.

Here is an example of how this runs:

We are running this in localhost with ports 6000, 6001,6002

```
System running on 10.10.13.236
Enter all the port numbers of the devices:
6000
6001
6002
Welcome User 10.10.13.236:6000

---------------------------------------
 Select the option below:
 1.Event Creation
 2.Request Critical Section
 3.Current Top of Stack
 4.Event Info

2
Message received: REPL_3@10.10.13.236:6001
Entering critical section
Message received: REPL_3@10.10.13.236:6002
CS Socket Connected
Message received: REQ_4@10.10.13.236:6001
Completed CS
exiting CS
Sending REPL to 6001 10.10.13.236

---------------------------------------
 Select the option below:
 1.Event Creation
 2.Request Critical Section
 3.Current Top of Stack
 4.Event Info

Message received: REL_12@10.10.13.236:6001
```

```
System running on 10.10.13.236
Enter all the port numbers of the devices:
6001
6000
6002
Welcome User 10.10.13.236:6001

---------------------------------------
 Select the option below:
 1.Event Creation
 2.Request Critical Section
 3.Current Top of Stack
 4.Event Info

Sending REPL to 6000 10.10.13.236
Message received: REQ_1@10.10.13.236:6000
2
Message received: REL_7@10.10.13.236:6000
Message received: REPL_9@10.10.13.236:6002
Entering critical section
CS Socket Connected
Message received: REPL_8@10.10.13.236:6000
Completed CS
exiting CS

---------------------------------------
 Select the option below:
 1.Event Creation
 2.Request Critical Section
 3.Current Top of Stack
 4.Event Info
```

```
'C:\Users\thris\AppData\Local\Temp\vscodesws_1c3
\jdt.ls-java-project\bin' 'Main'
System running on 10.10.13.236
Enter all the port numbers of the devices:
6002
6000
6001
Welcome User 10.10.13.236:6002

---------------------------------------
 Select the option below:
 1.Event Creation
 2.Request Critical Section
 3.Current Top of Stack
 4.Event Info

Sending REPL to 6000 10.10.13.236
Message received: REQ_1@10.10.13.236:6000
Message received: REQ_4@10.10.13.236:6001
3
Top element in the queue is 1=10.10.13.236:6000

---------------------------------------
 Select the option below:
 1.Event Creation
 2.Request Critical Section
 3.Current Top of Stack
 4.Event Info

Sending REPL to 6001 10.10.13.236
Message received: REL_7@10.10.13.236:6000
Message received: REL_12@10.10.13.236:6001
```

The above example is of 6000 giving request first and 6001 giving immediate request next.

Here is a screenshot of timestamps of all the events that happened:

```
4
1 : 1 : sendREQ
2 : 4 : receiveREPL
3 : 5 : receiveREPL
4 : 6 : receiveREQ
5 : 7 : sendREL
6 : 8 : sendREPL
7 : 13 : receiveREL
```

```
4
1 : 2 : receiveREQ
2 : 3 : sendREPL
3 : 4 : sendREQ
4 : 8 : receiveREL
5 : 10 : receiveREPL
6 : 11 : receiveREPL
7 : 12 : sendREL
```

```
4
1 : 2 : receiveREQ
2 : 3 : sendREPL
3 : 5 : receiveREQ
4 : 8 : receiveREL
5 : 9 : sendREPL
6 : 13 : receiveREL
```