PROJECT2 REPORT

Sohil L. Shrestha and Akash Lohani

1001556964, 1001661458
February 12, 2019

I have neither given or received unauthorized assistance on this work.
Signed: Sohil L. Shrestha, Akash Lohani Date: February 12, 2019

1 Abstract

In this programming assignment, we implemented a synchronization environment based on logical clock as well as distributed locking scheme. We implemented totally ordered multicast based on Lamport Clock in Assignment1, and based on Vector clock in Assignment2. In Assignment3, we used token ring scheme to perform distributed locking system. The remainder of this report is structured as follows. We discuss related Understandings of theory in Section 2, followed by an explanation of our implementation in Section 3. We evaluate our programming by elaborating the encountered issue in Section 4. Finally, we will conclude our assignment work in Section 5.

2 What We have learnt

Lamport Clock

The programming assignment allowed us to visualize how lamport algorithm is used to achieve totally ordered events. Lamport Algorithm is stated as below:

1. Before executing an event P_i executes $C_i = C_i + 1$.
2. When process Pi sends a message m to P_j, it sets ms timestamp ts (m) equal to C_i after having executed the previous step.
3. Upon the receipt of a message m, process P_j adjusts its own local counter as C_j=maxC_j, ts (m), after which it then executes the first step and delivers the message to the application.

Lamport Clock algorithm allowed to achieve casual ordering between the processes. However to achieve total order, it can be extended as discussed below.

Totally Ordered Multicast

Totally Ordered Multicast is stated as below :

1. Apply Lamports algorithm
2. Every message is timestamped and the local counter is adjusted according to every message
3. Each update triggers a multicast to all servers
4. Each server multicasts an acknowledgement for every received update request
5. Pass the message to the application only when
   (a) The message is at the head of the queue
   (b) All acknowledgements of this message has been received
6. The above steps guarantees that the messages are in the same order at every server, assuming Message transmission is reliable

   The algorithm ensures that the events occurred at each process in the same order in all processes. The delivery restrictions enables the ordering of the messages. Totally ordered multi-cast takes a lot of network bandwidth as messages as well as acknowledge is multi-casted to other processes.

Vector Clock

Vector Clock allows us to implement easier totally ordered synchronization between devices. The property of Vector Clock VC[j] can be described as below.

1. $VC_i[j]$ is the number of events that have occured so far at Pi.
2. If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j.

Based on above-mentioned properties, Vector Clock behaves as follows.

1. Before sending a message, P_i executes,

$$VC_i[i] = VC_i[i] + 1 \tag{1}$$

2. When process P_i sends a message m to P_j, it set m's (vector) timestamp ts (m) to VC_i after having executed the previous step.
3. When node P_j receives a message from node P_i with ts(m), it delays delivery until;:

$$ts(m)[i] = VC_i[i] + 1 ts(m)[k] <= VC_j[k] for any k <> i \tag{2}$$

4. Upon the receipt of a message m, process P_j adjusts its own vector by setting

$$VC_j[k] = maxVC_j[k], ts(m)[k] \tag{3}$$

   for each k and delivers the message to the application.

This step will allow us to simply obtain the feature of totally order multicast.

Token Ring Algorithm
This is one of the algorithm in distriibuted locking scheme.

1. Pass the token along the ring based distributed system and only the process that has the token has the right to enter the critical region.

2. If the process that has the token wishes to enter the critical region, it used the token to open the shared file and when it finishes it will pass the token to the next process.

3. If the process don't want to enter the critical region, it just pass the token to the next process.

Token ring Algorithm is simple to deploy, however it does not work if tokens are lost. Furthermore it is difficult to detect whether the system has lost the token or not.

3 Implementation

Assignment1
Each process has its own (1) communication thread and (2) sender thread which will run infinitely.
(1)Communication Thread will keep listening to a port . Once a message/packet is received, it will distinguish if it is an event message (which is seeded with "MSG") or if it an acknowledgement (seeded with "ACK"). An event message is put into a priority queue which is adjusted based on the timestamp of the message and if the timestamp of both the message is same, it is adjusted based on the process id (pid) .

(2)Sender thread is responsible for issuing messages as well as multi-casting acknowledge. It will issue a message, then updates its local clock based on lamport algorithm, multi-cast the message to all other processes (including itself). When the previously issued message is delivered, the process will issue a new message.

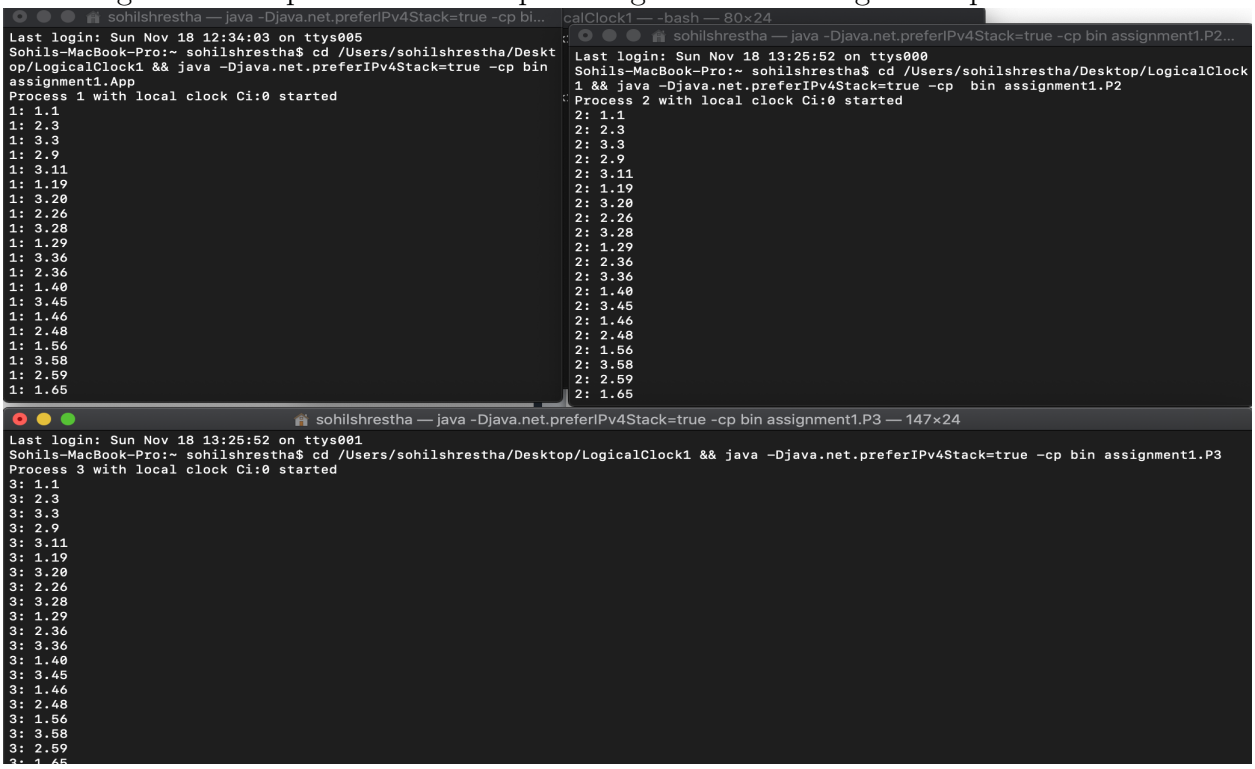
Before issuing a message, it will updates its logical clock and then binds the messages with the logical clock and then multicast to all other processes
To test the program, we used 3 processes which is executed at the same time and allow it issue and multi-cast messages to each other. The implementation makes use of multicast api to ease the multi-casting messages as well as acknowledgement to all other processes. We have eliminated the use of sorting thread with priority queue which basically allows

each process to maintain a queue based on the priority(first based on the time stamp of the messages and then based on process id).
When messages are delivered,we print the messages as CurrentPID:PID.EventID. Message is delivered to applications only when

- It is at head of queue
- It has been acknowledged by all involved processes
- P_i sends an acknowledgement to P_j if
    1. P_i has not made an update request
    2. P_is identifier is greater than P_js identifier
    3. P_is update has been processed;

The details of how to run the code is in the ReadMe file . We use three processes to ensure that the events happening in different processes have the same order(totally ordered). With the shell script, all three processes first spawns it communication thread that listens top the port, sleeps for a second and then spawns its sender thread to issue messages. We wait for the sender thread to spawn because it helps eliminate the warm up period where every processes don't have their listening thread (or communication thread) up and ready.

Following is the snapshot of the output we get after running three processes at a time:



In this way, we were able to achieve totally ordered multi-casting.

Assignment2
In this assignment, we implemented synchronization based on Vector Clock. Each process has its own (1) Receiving/Adding-Buffer thread and (2) Choosing/Modifying thread (3) Sending thread $\times 2$. Each thread work as follows.

(1) Receiving/Adding-Buffer Thread
It will create UDP Socket and binds to the port. It keeps listening until it receives the data from other process. It converts the received byte data into Integer vector and adds to the global list. Note that, while writing the vector into the list it performs locking in order not to crash the list which is also referred by another thread. We tackled this problem by using boolean variable. For example, while writing value into the list it turns the boolean variable into "true" which enables another process to understand the list is "in use".

(2) Choosing/Modifying Thread
First of all, it will choose the vector from the global list based on the following conditions.

$$ts(m)[i] = VC_i[i] + 1 \tag{4}$$
$$ts(m)[k] <= VC_j[k] for any k <> i \tag{5}$$

If it fullfills the above-mentioned condition, it will pick the vector and proceeds the modification of the local vector based on the following rule.

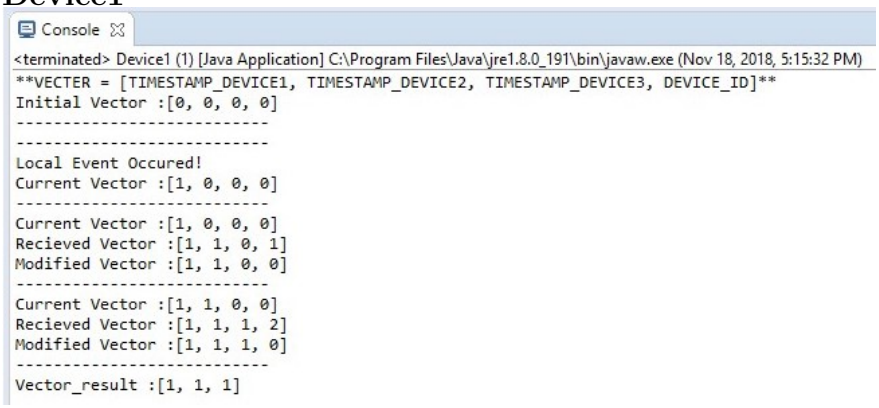
$$VC_j[k] = max VC_j[k], ts(m)[k] \tag{6}$$

If it won't fullfill the above-mention condition, it will suspend the procedure and leave the vector in the list. We performed same locking scheme used in Receiveing/Adding-Buffer thread.

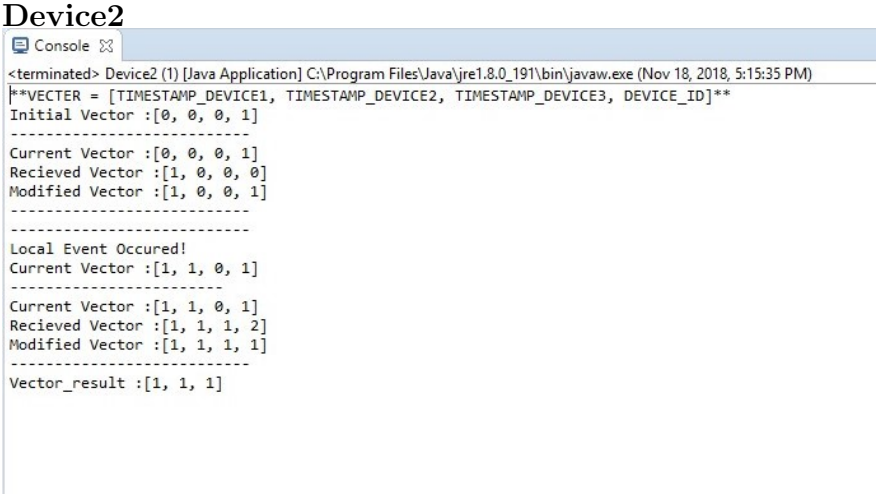
(3) Sending Thread
It will increment the vector component by 1 and multicast the vector to different process using UDP connection. In addition to that, in order to identify the sender of the vector we also append "Device ID" as the fourth element of the vector. Note that, we can also identify sender by creating different receiving port for each sender.

Results on each Device are shown next page. The details of how to run the code is in the ReadMe file.

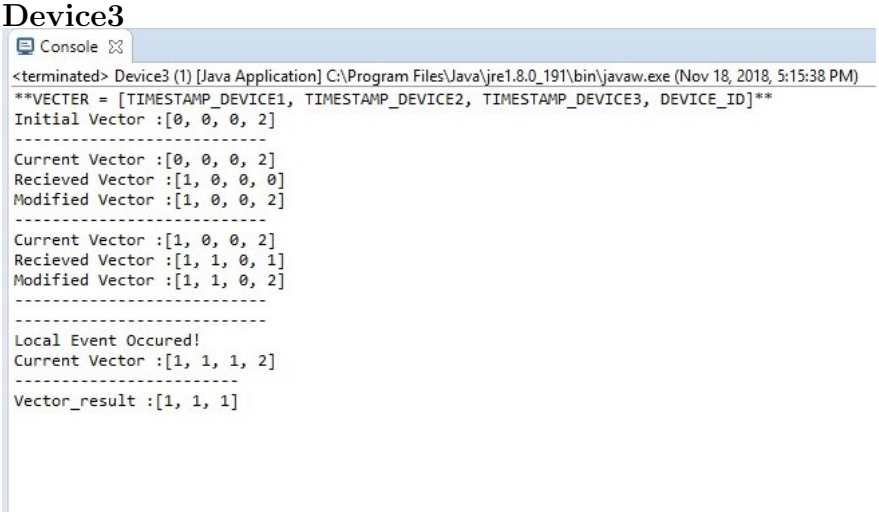
Device1



Device2



Device3



Assignment3

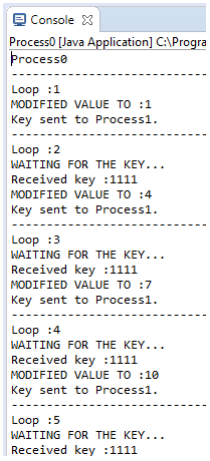
In this Assignment, we implemented token ring based distributed locking scheme. Each process has single thread which deals with three function, (1) Receiving key, (2) Reading file, (3) Sending key.

1. Receiving key
   It creats UDP Socket, binds to specific port, and keep listening to the port. If it receives the key, the following procedure is performed.

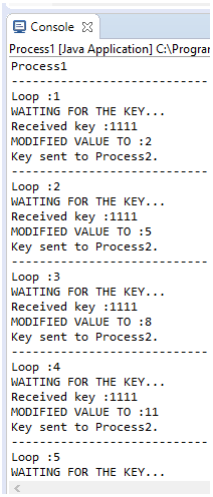
2. Reading file
   It will check whether the Process has done predefined times. If yes, it checks whether the key matches the predifined key value(1111) or not, opens the textfile(Counter.txt) and increments the written Integer value by 1. If not, it just passes the key to next process using following function. Note that Counter.txt file is created in the same package as soon as we run Process0.

3. Sending key
   It sends the key to the process next to it.

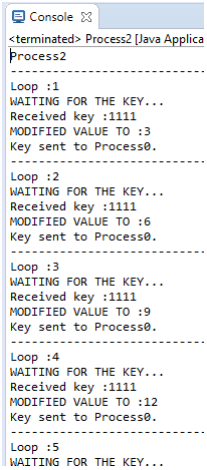
In this way, the key is passed and circulated between, Process0, Process1, Process2. When Process obtain a key, it will open the shared text file and modify the counter for predefined times. 20 times for Process0, 30 times for Process1, and 40 times for Process2. If the Process reaches its predefined iteration, it will just pass the key to the next process. So in this case, the process will circulate the key for 40 loops, so that the counter in the text file will show 90 after running the process. The detail of how to run the code is in the ReadMe file. The Result for each Process is shown below. As you can see, the key is successfully passed and the counter in the Counter.txt file is increased from 0 to 90.



Process0

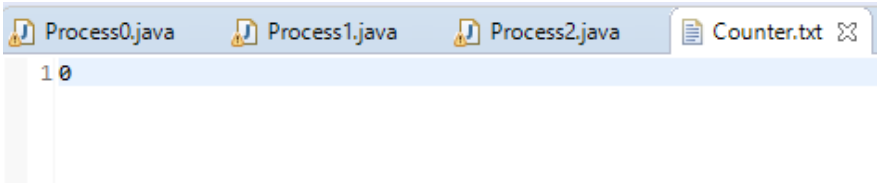


Process1

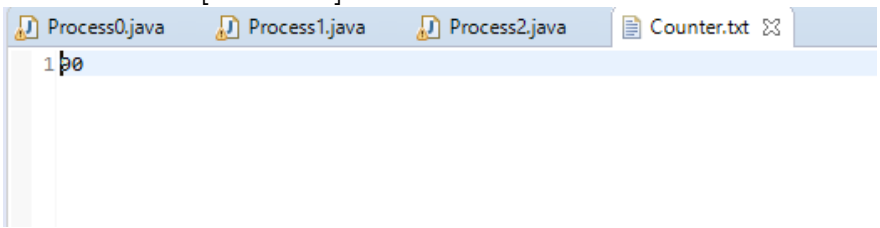


Process2

Counter.txt [BEFORE]



Counter.txt [AFTER]



4 Issues encountered

1. One issue encountered while implementing the programming assignment for total ordered multicasting was to achieve synchronization between the processes. As each processes has to be spawned such that the multi-casted messages as well as acknowledgement are not lost in the network, we were initially unable to spawn three processes at the same time. We eventually were able to tackle the issue with help of thread sleeps and shell script execution.Further details reagrding the output are explained in the implementation.

2. Creating a separate thread to sort as well as insert messages to queue made the receiving queue a shared resources. Locking the shared queue was a big challenge as it has shared between multiple classes. In assignment 1, we tackled this issue by eliminating the need of a sorting queue by using a JAVA blocking queue API. We implemented our own priority defination such that whenever a message is received, the process will insert the received messages abiding by the priority defined. Further details on priority queue are explained in implementation. In Assignment 2 and 3, we used boolean variable to check whether the shared source is used or not. For example while writing something into queue, we set the boolean variable as "true", so that other process will understand the queue is "in use" and wait until it writes and turn the value into "false".

3. While modifying the vector clock, we need to identify which vector was sent by which process. In order to tackle this problem, we appended fourth elements to vector which corresponds to Device ID.

4. While working on Assignment 3, sometimes key was lost due to some error in receiving function. Since we used UDP connection while passing the key, if there is no corresponding port while sending key from one process to another, we simply loose the key and process ends to deadlock.

5 Conclusion

Implementing the programming assignment, we learnt a lot regarding the synchronization between devices and the importance of locking scheme. We were able to implement our totally ordered multicast using Lamport's logical clock and Vector Clock. We also encoutered the serious problem while using Token ring based distributed locking system. Overally, we enhanced our coding skill as well as understanding in synchronization which is critically important while talking about distributed system.

6 References

1. https://stackoverflow.com/
2. $https://www.youtube.com/results?search_query = totally + ordered + multicast$
3. http://syllabus.cs.manchester.ac.uk/ugt/COMP28112/2011/slides09.pdf
4. http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html
5. https://piazza.com/class/jl5icerfdt97gp?cid=44