CSE 5306: Distributed Systems
Project 2 Report

Hamza Reza Pavel (1001741797)
Manish Munikar (1001826846)

November 7, 2020

I have neither given nor received unauthorized assistance on this work.

Signed: Hamza Reza Pavel, Manish Munikar Date: October 10, 2020

Assignment 1: Totally Ordered Multicasting

In this assignment, we implemented the totally ordered multicast using Lamport's algorithm. We implemented the algorithm and simulated the output in python.

For communication between processes, we used UDP socket-based communication for simplicity. Our main function spawns N number of processes, each with a different process id or PID and port number. Each process has a port number defined by (port offset + process id). Each process runs two threads: default thread for sending events, and a communication thread for receiving the events and sending acknowledgments from/to other processes. When an event occurs, the process increments its logical clock and multicasts the event to all processes in the group. We have added a delay to simulate the occurrence of events. When a process receives an event, it increments its logical clock and puts the event in a buffer queue. The communication thread's response depends on the message it receives:

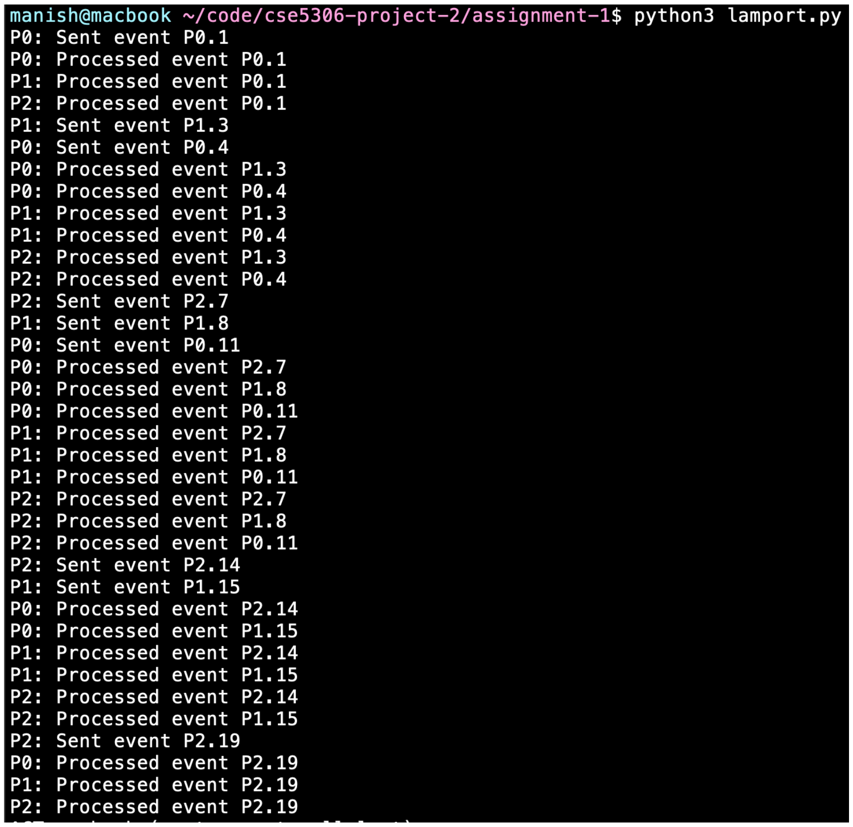
Receiving an event:

1. It updates its logical clock by comparing it with the timestamp in the received event.
2. It puts the event in a priority queue, which is a min-heap with event's timestamp as the comparison key.
3. It broadcasts the acknowledgment for the event at the front of the queue if not already sent.

Receiving an acknowledgment:

1. It increments the acknowledgment count for that event.
2. If the event at the front of the queue has received all the acknowledgments, then that event is dequeued and processed (by simply printing a message in the terminal). Otherwise, it broadcasts the acknowledgment for that event if not already sent

From this assignment, we have learned the working of Lamport's algorithm and how to implement a totally ordered multicast using it. The main challenge was handling interprocess communication without blocking the processes. Also, we have implemented the assignment so that it can be tested with N number of processes. Figure 1 shows a sample of the output of the program, where we can verify that all processes process events in the same order.



```
manish@macbook ~/code/cse5306-project-2/assignment-1$ python3 lamport.py
P0: Sent event P0.1
P0: Processed event P0.1
P1: Processed event P0.1
P2: Processed event P0.1
P1: Sent event P1.3
P0: Sent event P0.4
P0: Processed event P1.3
P0: Processed event P0.4
P1: Processed event P1.3
P1: Processed event P0.4
P2: Processed event P1.3
P2: Processed event P0.4
P2: Sent event P2.7
P1: Sent event P1.8
P0: Sent event P0.11
P0: Processed event P2.7
P0: Processed event P1.8
P0: Processed event P0.11
P1: Processed event P2.7
P1: Processed event P1.8
P1: Processed event P0.11
P2: Processed event P2.7
P2: Processed event P1.8
P2: Processed event P0.11
P2: Sent event P2.14
P1: Sent event P1.15
P0: Processed event P2.14
P0: Processed event P1.15
P1: Processed event P2.14
P1: Processed event P1.15
P2: Processed event P2.14
P2: Processed event P1.15
P2: Sent event P2.19
P0: Processed event P2.19
P1: Processed event P2.19
P2: Processed event P2.19
```

Figure 1: Sample output of assignment 1 code

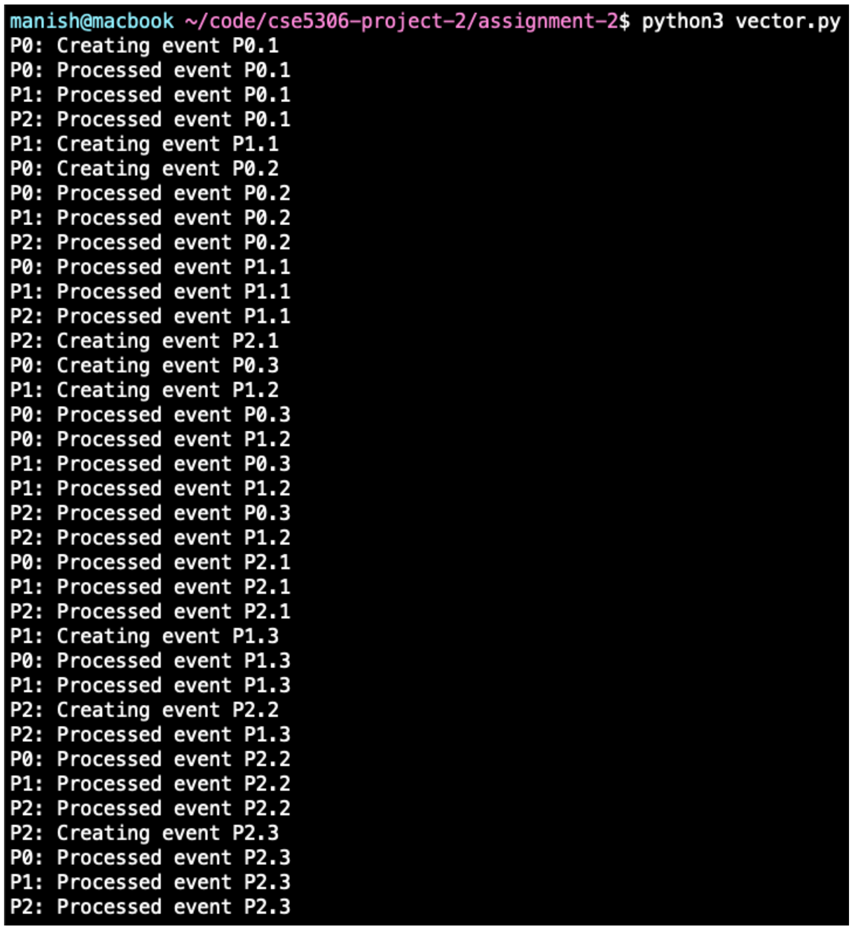
Assignment 2: Vector Clock

In this assignment, we implemented the vector clock algorithm. Same as assignment 1, we implemented the algorithm and simulated the output in python.

Due to similarities between assignment 1 and assignment 2, we have used assignment 1 as a base for assignment 2. That means the threads to handle communication between processes and creating and processing events are similar. Each process has two threads: main thread that generates events and a communication thread that handles all the incoming messages. Each process maintains its own local vector clock. When an event occurs, each process updates its vector clock according to the step mentioned in the algorithm and sends it to all the processes along with the message. Each process puts the received message in a buffer and processes the message if the condition mentioned in step 3 of the algorithm is fulfilled. Once the message is delivered, the process updates its vector clock accordingly.

From this assignment, we have learned the working procedures of the vector clock algorithm. Performing the steps to identify when to deliver a message was a bit challenging in this assignment then the previous one. Also, there were cases of events being missed due to starting the event sender thread before starting the event receiver thread. This issue was circumvented by starting the event receiver thread before the event sender thread and putting a certain amount of delay between them.

Figure 2 shows the a sample output of the assignment 2 code.



```
manish@macbook ~/code/cse5306-project-2/assignment-2$ python3 vector.py
P0: Creating event P0.1
P0: Processed event P0.1
P1: Processed event P0.1
P2: Processed event P0.1
P1: Creating event P1.1
P0: Creating event P0.2
P0: Processed event P0.2
P1: Processed event P0.2
P2: Processed event P0.2
P0: Processed event P1.1
P1: Processed event P1.1
P2: Processed event P1.1
P2: Creating event P2.1
P0: Creating event P0.3
P1: Creating event P1.2
P0: Processed event P0.3
P0: Processed event P1.2
P1: Processed event P0.3
P1: Processed event P1.2
P2: Processed event P0.3
P2: Processed event P1.2
P0: Processed event P2.1
P1: Processed event P2.1
P2: Processed event P2.1
P1: Creating event P1.3
P0: Processed event P1.3
P1: Processed event P1.3
P2: Creating event P2.2
P2: Processed event P1.3
P0: Processed event P2.2
P1: Processed event P2.2
P2: Processed event P2.2
P2: Creating event P2.3
P0: Processed event P2.3
P1: Processed event P2.3
P2: Processed event P2.3
```

Figure 2: Sample output of assignment 2 code

Assignment 3: Distributed Locking

In this assignment, we implemented two locking mechanisms used in distributed systems. Locking is required in distributed systems to ensure consistency by enabling mutual exclusion between different processes trying to access the same resource. We implemented the following locking algorithms.

Centralized Coordinated Locking

This is a permission-based locking algorithm where processes have to acquire permission (lock) from a centralized coordinator before accessing a shared resource. The coordinator is a server process that accepts lock acquire requests from worker processes, maintains them in a queue, and grants the lock to them one after another on a first-in-first-out (FIFO) basis.

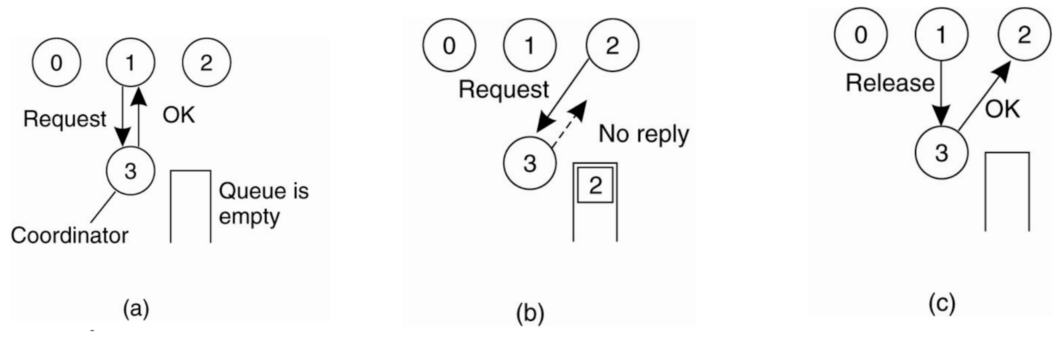


Figure 3: Process of acquiring and releasing a lock in the centralized locking algorithm.

Figure 3 shows the working mechanism of the centralized locking algorithm. There is one coordinator process (implemented as CoordinatorProcess class in the code) and multiple member-processes (implemented as MemberProcess class in the code). The coordinator process listens on a predefined port for lock REQUEST and RELEASE messages from member processes. Whenever a member wants to acquire the lock, it sends a REQUEST message to the coordinator and waits for an OK reply from the coordinator. If the lock is unused at the time, the coordinator grants access to the requester by replying to it with an OK message. When the member is done with the lock, it releases the lock by sending a RELEASE message to the coordinator.

If the coordinator receives a REQUEST message while the lock is being held by some other process, it will not reply immediately. Instead, it will enqueue the message to a request queue. Once the coordinator gets a RELEASE message from the current lock holder, it will send an OK message to the first member in the request queue. In this way, we can ensure that only one process accesses the shared resource at a time.

We implemented all the communications between members and coordinators using the connection-oriented reliable TCP protocol. While efficient and simple to implement, it has a fundamental drawback: the coordinator is a single point of failure. Still, this is the most commonly used mutual exclusion mechanism in popular distributed systems.

Distributed Peer-to-Peer Locking

In the distributed peer-to-peer locking algorithm, there is no centralized coordinator. A member has to send the lock request to every other member and get a positive reply from all of them to acquire the lock to a shared resource. In this algorithm, each member has to also do the work of the coordinator, i.e., each member process has two threads: one accesses and works on the shared resource, the other listens to and responds to lock request messages. Figure 4 shows how the lock is acquired in this algorithm.

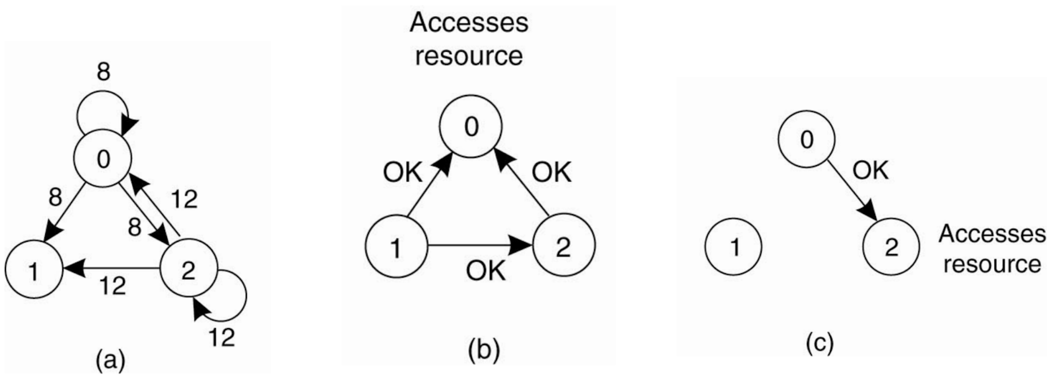


Figure 4: An example of the process of acquiring the lock in the distributed algorithm.

In our implementation, there is only one type of process (called MemberProcess in the code). Each member process spawns two threads as mentioned above: one listens on a predefined port and handles REQUEST messages and the other competes for the lock to perform an operation on a file.

Acquiring the lock involves sending a REQUEST message to all the other members and waiting for the OK reply from all of them. The REQUEST message consists of the resource id, timestamp, and member id. When a member receives a REQUEST message, what it does depends on whether it holds (or has requested for) the lock. If it's not holding and has not requested the lock, it will immediately respond with an OK message. If the current member holds the lock, it will queue the REQUEST messages and sends OK messages to them only when it wants to release the lock. Finally, if the current member has requested the lock, then it will compare the timestamp between its request and the received request. It will respond OK only if the received request has a lower timestamp than its own request.

This algorithm tries to mitigate the "single point of failure" drawback of the centralized algorithm but introduces several new problems:

- Each member has to do the work of the coordinator.
- The algorithm fails when any process fails.
- Requires more message passing (more expensive).
- Requires a notion of a global clock to order requests from multiple processes.

In our implementation, we need to have communication channels among all processes, so there needs to a lot of connections set up for message passing. We encountered many errors related to "address already in use" when creating TCP connections. Not being able to resolve them after a lot of debugging attempts, we ended up using a connectionless UDP protocol for message passing, assuming a reliable, lossless network.