CMSC 621: Project 2

Clocks, Multicast, and COMMIT

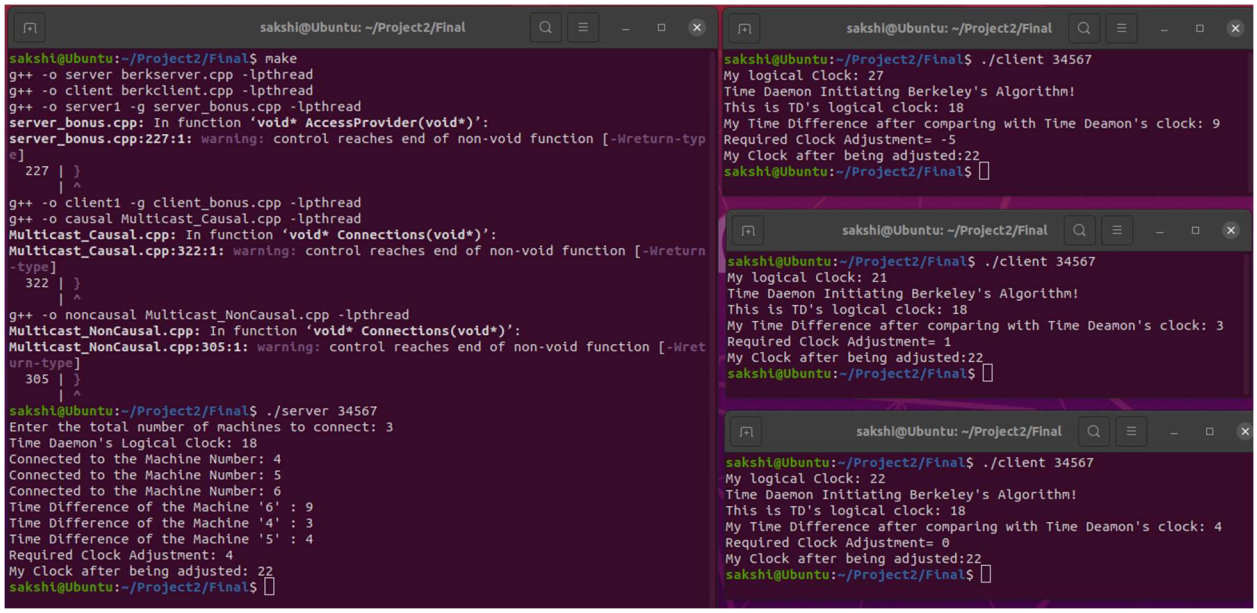
Sakshi Umesh Reddy – YO19530

Assignment 1:

- **Implementation:**

In this assignment I have implemented the Berkeley's Algorithm. The server program acts as a Time Deamon and other machines/processes are executed by executing client program. After executing the server program, we must give the number of clients to be connected. Initially a random number in range 5-30 is selected as logical clock of that machine/process. The server keeps accepting the connections till all the mentioned clients are connected and it spawns a new thread to each client for further communication. Once all the clients are connected to the server, it starts the Berkeley's Algorithm by sending it's own logical clock to all connected clients.

Each client then calculates the difference between it's logical clock and the time deamon's logical clock, then it sends the difference to the server (time deamon). The server adds all the received time differences, its own time difference and calculates the average value. So, the server calculates the time adjustment to be done by each machine/process including itself by subtracting the received time difference from the average. The value of time adjustment can be negative as well. After calculating it, the server sends the adjustment value to the respective clients and the clients add the received time adjustment to their clock to get the correct logical clock. The server adjusts its own logical clock by adding the average value to its logical clock and the algorithm terminates.



- **Issues encountered while implementing:**

I was not sure how will the server know when to start the algorithm. So, I have kept the number of clients to be connected as user input. The server waits till the number of clients connected is equal to the given number before starting the algorithm.

- **Learnings:**

By implementing the Berkeley's Algorithm, I got to implement the pthread correctly this time. Also I learned how the algorithm is implemented, how the calculation is implemented.

Assignment 2:

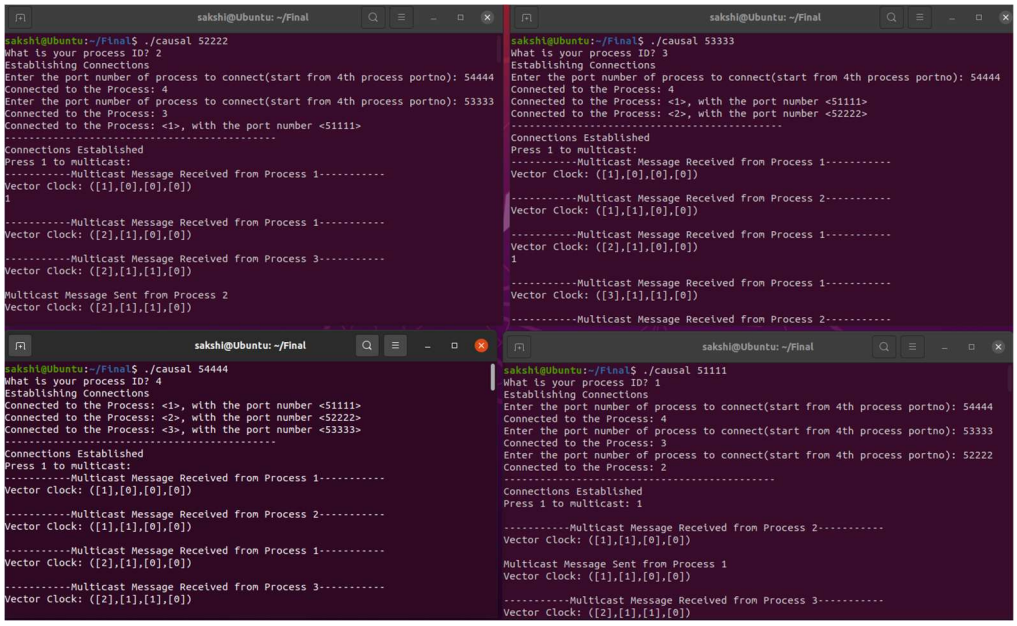
- **Implementation:**

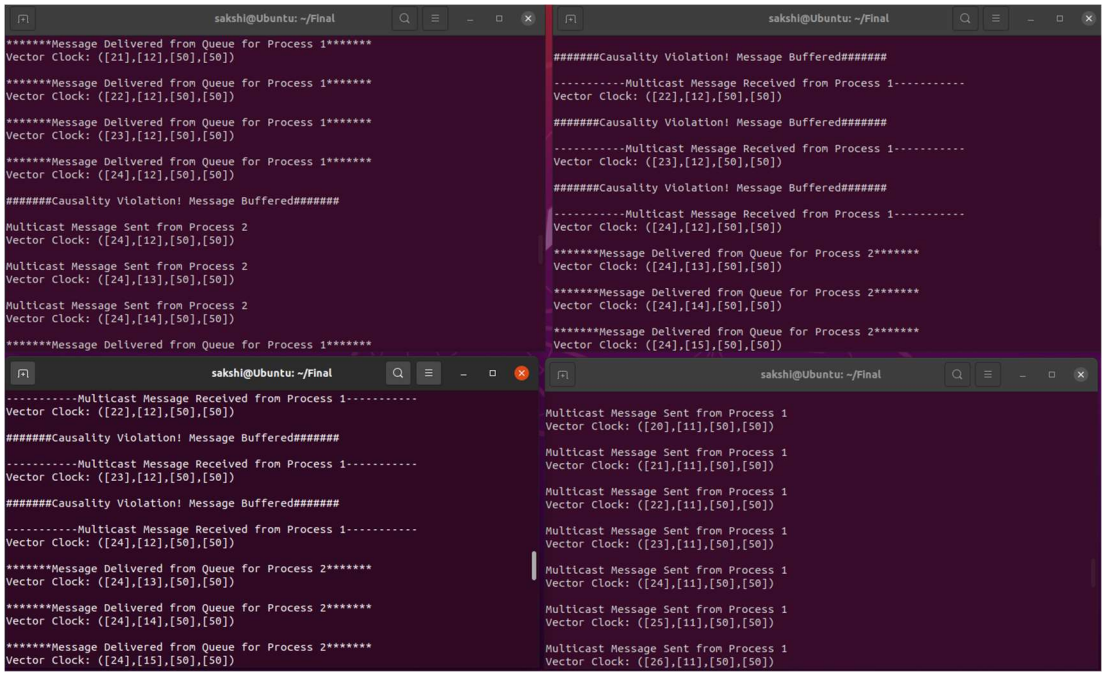
All the processes are connected to each other, and every process maintains a vector clock having the logical clock value for all the processes. Each process is associated with an element of an array of a structure that has all relevant variables to store information regarding each process. To establish the connection, some specific order must be maintained while providing input to the program. These details are mentioned in README file.

Once the connection is established, each process can send multicast message to all other processes. I have hardcoded the number of multicast messages a process sends to all other processes, i.e. 50 times with its vector clock. To resolve the causality issues, I had to create non-causality by adding random time before receive operation and send operation.

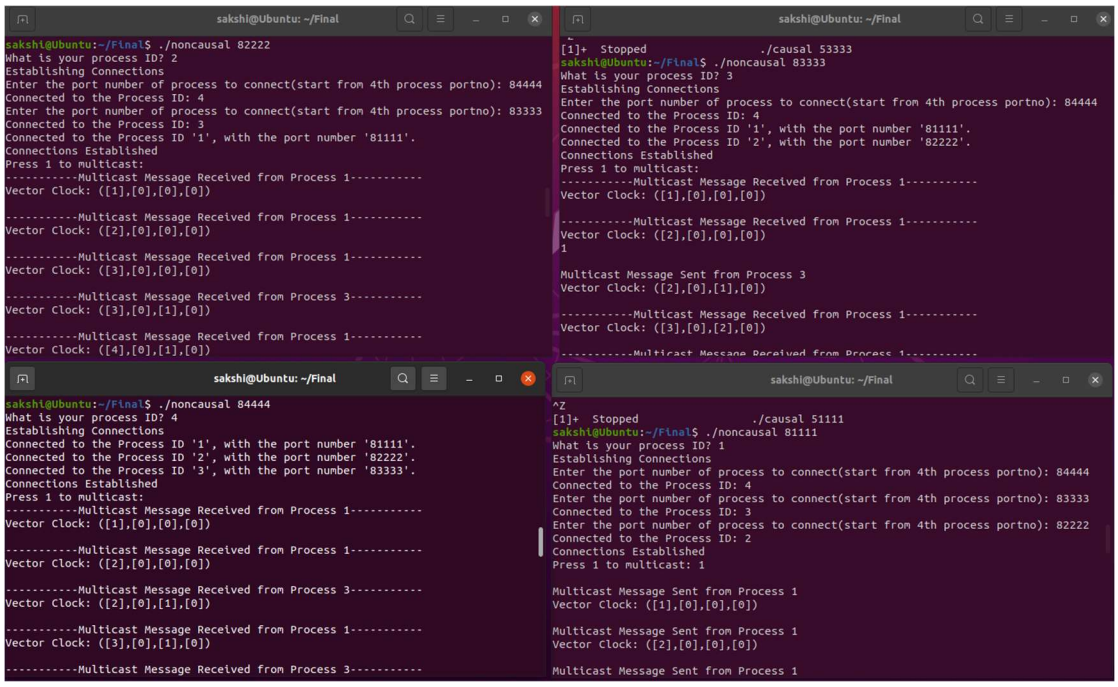
Two programs are written, one for causal ordering and another for non-causal ordering.

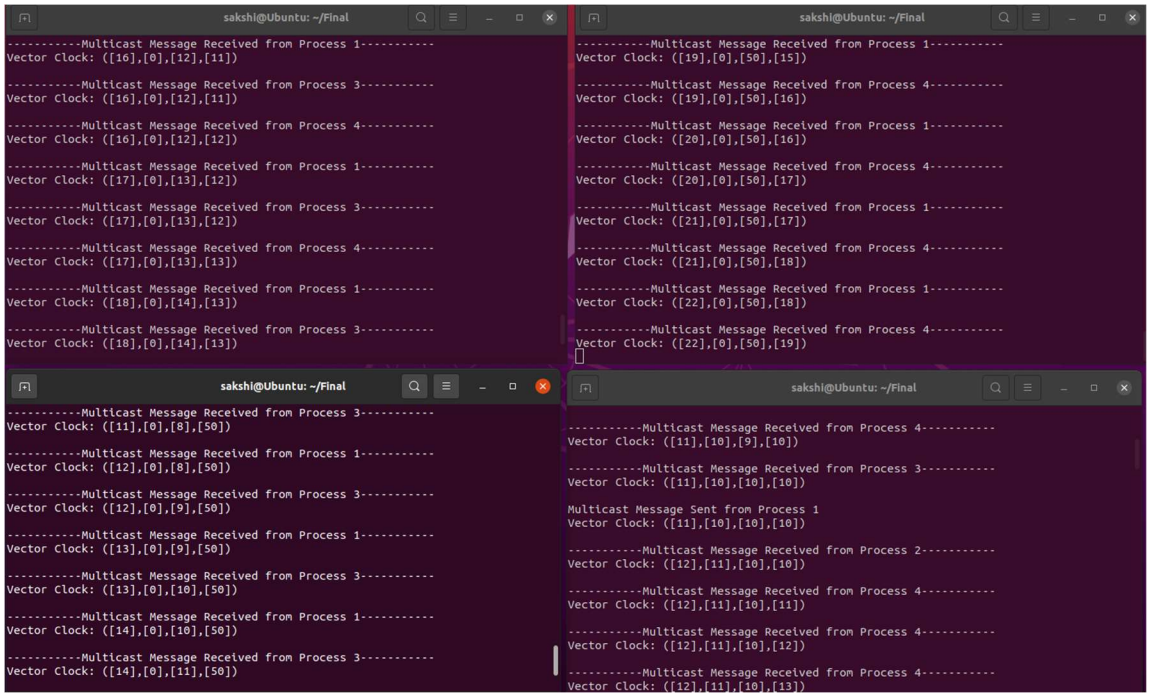
In causal ordering, when a message is received from a process, a causality check is done. If the messsge satisfies the conditions of causality, it is delivered to the application. Otherwise, it is buffered and checked after every correct message received by that process because of all the messages are received in correct causal sequence.





In non causal ordering, the messages are delivered as they are received without considering the causality of events.





- **Limitations:**

Works for only 4 processes.

Also, each process sends 50 multicast messages to other processes.

- **Issues encountered while implementing:**

This assignment was the toughest of all. It was very hard to implement the n-node communication. I was not sure how to establish the connection between all the processes. To make it little easy I have considered only 4 processes for this assignment. To establish communication between each node, both server and client code is implemented in one program. Also, by making a thread for listening to the connections as a server and simultaneously sending connection requests to other processes as a client.

Another issue was to store all the socket file descriptors for further communication. This issue was resolved by creating an array of structure with all the relevant information variables, where each array element is a process in system.

- **Learnings:**

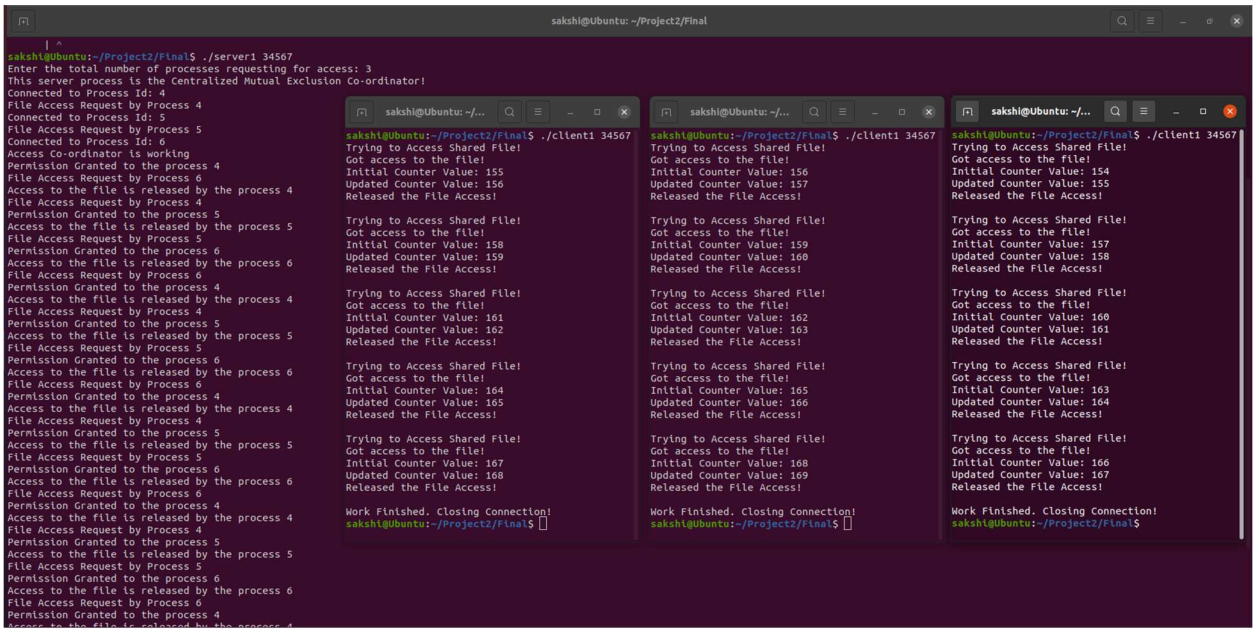
While implementing this assignment, I got to learn how to implement both client and server in a single program and to implement the n-node communication. I also learned socket commands to reuse the same socket address for multiple connections.

Assignment 3:

- **Implementation:**

Here, the server program acts as a coordinator and multiple client instances are executed to get multiple processes to access the critical section. After executing the server program, we must give the number of clients to be connected. The server keeps accepting the connections till all the mentioned clients are connected and it spawns a new thread to each client for further communication.

As soon as a client gets connected to the server, it sends REQUEST message to the coordinator. On server side, this message is added to the queue. Once all the clients are connected. The server checks if the critical section is available to provide access to requesting client processes. If the critical section is available, the server sends OK message to the first client in the queue to access the critical section file to increment and update the counter. When the client has done the required changes to the critical section file, it sends RELEASE message to the server.



Till then the next request in the queue waits on the conditional variable and when the release message is received from previous process, the lock is released, and signal is sent to the conditional variable. After receiving the signal, the waiting process wakes up and the coordinator provides access of the critical section to this process. Like this all processes/clients share the critical section to increment the counter without any deadlock.

- **Issues encountered while implementing:**

While implementing this algorithm, I was not able to figure out a way to provide access to the clients and the locking mechanism. I thought of using flags to do that but then used conditional variables to implement the same.

- **Learnings:**

By implementing this algorithm, I got to learn how to implement the conditional variables and mutex locks to avoid the deadlock.