AOS Project 2

Name: Intekhab Naser **ID: ZC11577**

Project Report

Project Fundamentals:

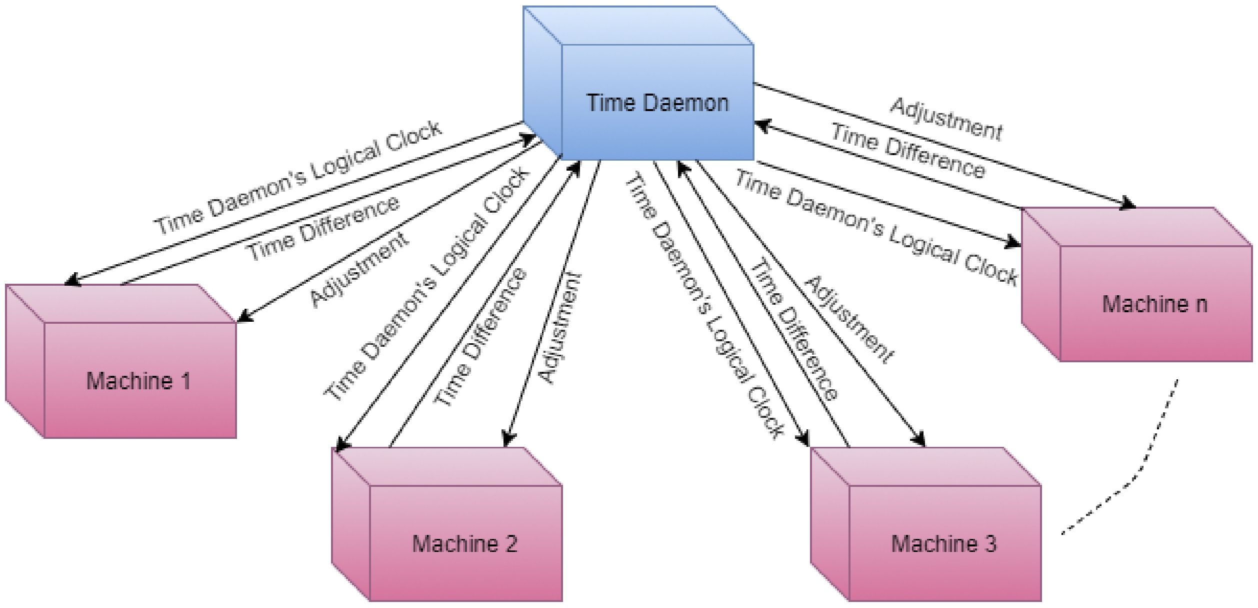
The programs in the project are based on the following concepts:

CLOCKS, MULTICAST, AND COMMIT

Assignment - 1:

In this Assignment, I used Client Server Architecture to implement the Berkeley's Algorithm.

1) **Design:**



2) **Implementation:**

The Time Daemon is a server and all the other machines/processes are clients. The server keeps accepting the connections till all desired machines are connected to it. For each client connection, server (Time Daemon) spawns a new thread for communicating with respective clients. All the machines, including the server, initialize their logical clocks randomly in the range of 5 to 30 integer values.

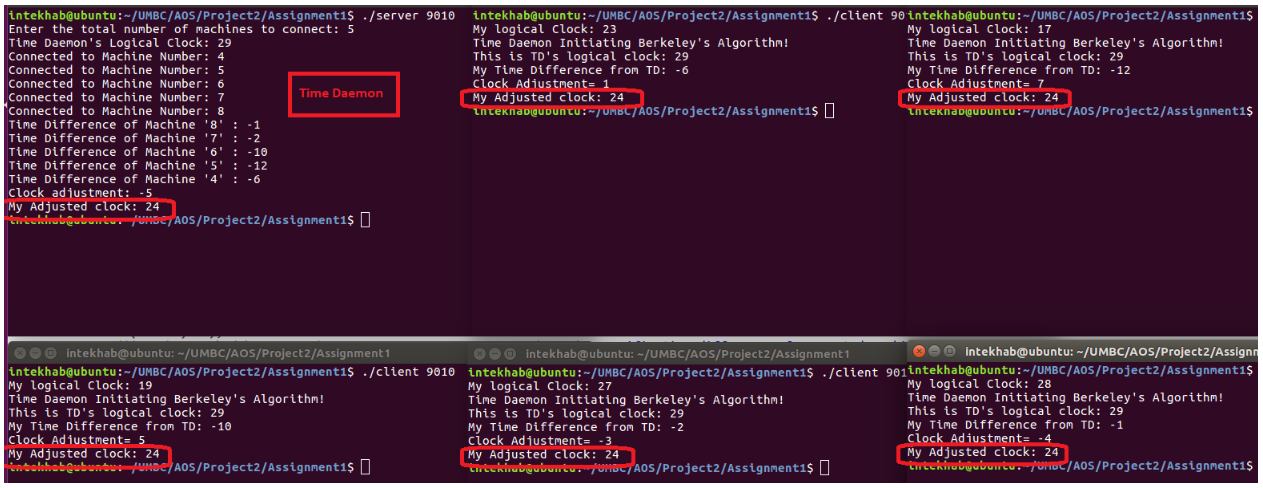
When all clients are connected, Time Daemon sends its logical clock to all the connected clients.

Each client then calculates the difference of their logical clock, minus the Time Daemon's logical clock and sends this time difference back to server. The server adds the time differences of all the connected clients and its own time difference (which is always zero) and calculates the average value.

Corresponding to each client, server calculates the time adjustment as: (average – time difference) and sends this time adjustment back to the client. All the clients collect their respective adjustment values and add it to their logical clocks, while the Time Daemon adds the average value to its logical clock and the algorithm terminates.

All the machines achieve consistent logical clocks.

A quick overview of how the algorithm is implemented may be understood better by the following screenshot:



```
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$ ./server 9010
Enter the total number of machines to connect: 5
Time Daemon's Logical Clock: 29
Connected to Machine Number: 4
Connected to Machine Number: 5
Connected to Machine Number: 6
Connected to Machine Number: 7
Connected to Machine Number: 8
Time Difference of Machine '8' : -1
Time Difference of Machine '7' : -2
Time Difference of Machine '6' : -10
Time Difference of Machine '5' : -12
Time Difference of Machine '4' : -6
Clock adjustment: -5
My Adjusted clock: 24
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$
```

```
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$ ./client 90
My logical Clock: 23
Time Daemon Initiating Berkeley's Algorithm!
This is TD's logical clock: 29
My Time Difference from TD: -6
Clock Adjustment= 1
My Adjusted clock: 24
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$
```

```
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$
My logical Clock: 17
Time Daemon Initiating Berkeley's Algorithm!
This is TD's logical clock: 29
My Time Difference from TD: -12
Clock Adjustment= 7
My Adjusted clock: 24
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$
```

```
intekhab@ubuntu: ~/UMBC/AOS/Project2/Assignment1
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$ ./client 9010
My logical Clock: 19
Time Daemon Initiating Berkeley's Algorithm!
This is TD's logical clock: 29
My Time Difference from TD: -10
Clock Adjustment= 5
My Adjusted clock: 24
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$
```

```
intekhab@ubuntu: ~/UMBC/AOS/Project2/Assignment1
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$ ./client 901
My logical Clock: 27
Time Daemon Initiating Berkeley's Algorithm!
This is TD's logical clock: 29
My Time Difference from TD: -2
Clock Adjustment= -3
My Adjusted clock: 24
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$
```

```
intekhab@ubuntu: ~/UMBC/AOS/Project2/Assignn
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$
My logical Clock: 28
Time Daemon Initiating Berkeley's Algorithm!
This is TD's logical clock: 29
My Time Difference from TD: -1
Clock Adjustment= -4
My Adjusted clock: 24
intekhab@ubuntu:~/UMBC/AOS/Project2/Assignment1$
```

3) Learnings:

By implementing this algorithm, I learned the simple process of Berkeley's algorithm of how time synchronization is achieved in a distributed system.

4) Issues:

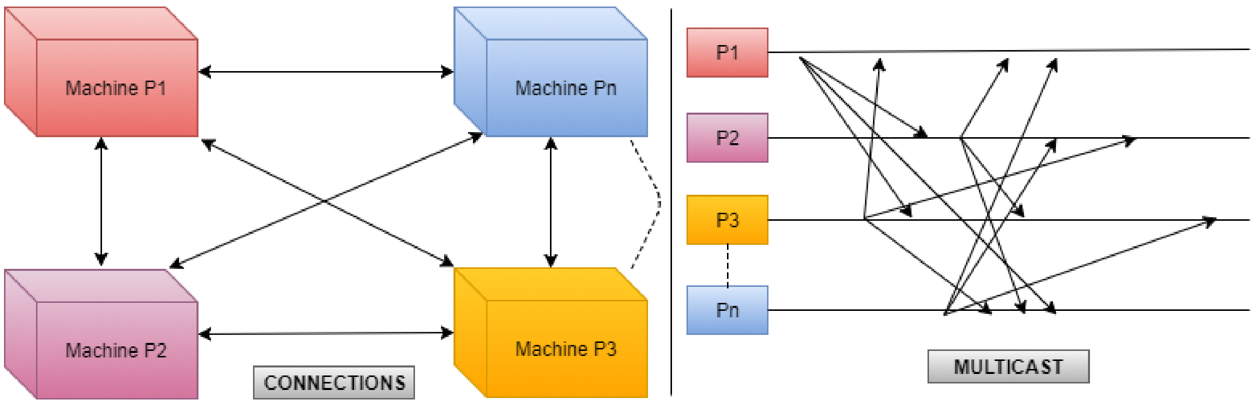
While implementing this algorithm, I was unsure of when to start the algorithm and how to calculate the average of all time differences across different clients. I solved this problem by making the server wait for all connections to complete before starting the algorithm and wait till all the clients have sent their time differences before calculating the average value.

-x-

<u>Assignment - 2</u>:

In this Assignment, I used an architecture in which all the machines/processes may behave as a client and server simultaneously to implement the Multicast message ordering mechanism.

1) Design:



2) Implementation:

All the machines are connected to each other and each process or machine knows about the total number of machines in the system. Every process maintains a vector clock in which the logical clock values of all corresponding processes is maintained.

Assumption: *It is assumed that any process in this system will send multicast message to all the other processes in the system i.e. for every process, the group for multicasting any message contains all the other processes in the system*.

All the processes multicast messages to all the other processes in the system and send their vector clocks along with the message. While sending a message, the process increments the vector clock value corresponding to its index by one and sends the vector with message.

Two versions of this system are implemented: one with causal ordering and the other is non-causal.

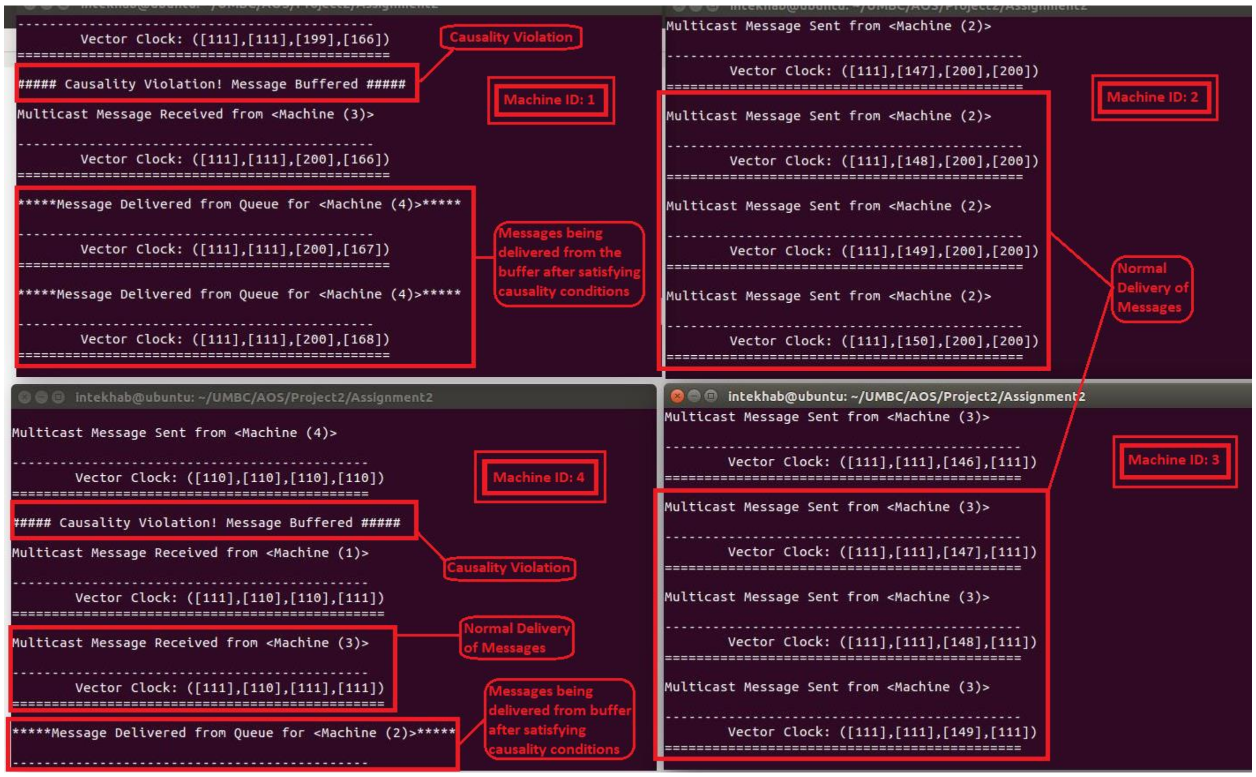
Comparison of Causal and Non-Causal Ordering:

In causal ordering, on receiving a message from a process, a causality check is done on message. If the message satisfies the conditions of causality, it is delivered to the application, otherwise it is buffered till it satisfies those conditions.

When a correct message is received, then the buffer is check for any messages who may now satisfy causality. **Thus, all the messages are delivered in the correct causal sequence**.

A quick overview of how the mechanism is implemented may be understood better by the following screenshot:

Causal Ordering Mechanism:


Causality Violation
Machine ID: 1
Messages being delivered from the buffer after satisfying causality conditions
Machine ID: 2
Normal Delivery of Messages
Machine ID: 4
Causality Violation
Normal Delivery of Messages
Messages being delivered from buffer after satisfying causality conditions
Machine ID: 3

```
        -----------------------------------------
        Vector Clock: ([111],[111],[199],[166])
=============================================
##### Causality Violation! Message Buffered #####
Multicast Message Received from <Machine (3)>

        -----------------------------------------
        Vector Clock: ([111],[111],[200],[166])
=============================================
*****Message Delivered from Queue for <Machine (4)>*****

        -----------------------------------------
        Vector Clock: ([111],[111],[200],[167])
=============================================
*****Message Delivered from Queue for <Machine (4)>*****

        -----------------------------------------
        Vector Clock: ([111],[111],[200],[168])
=============================================
```

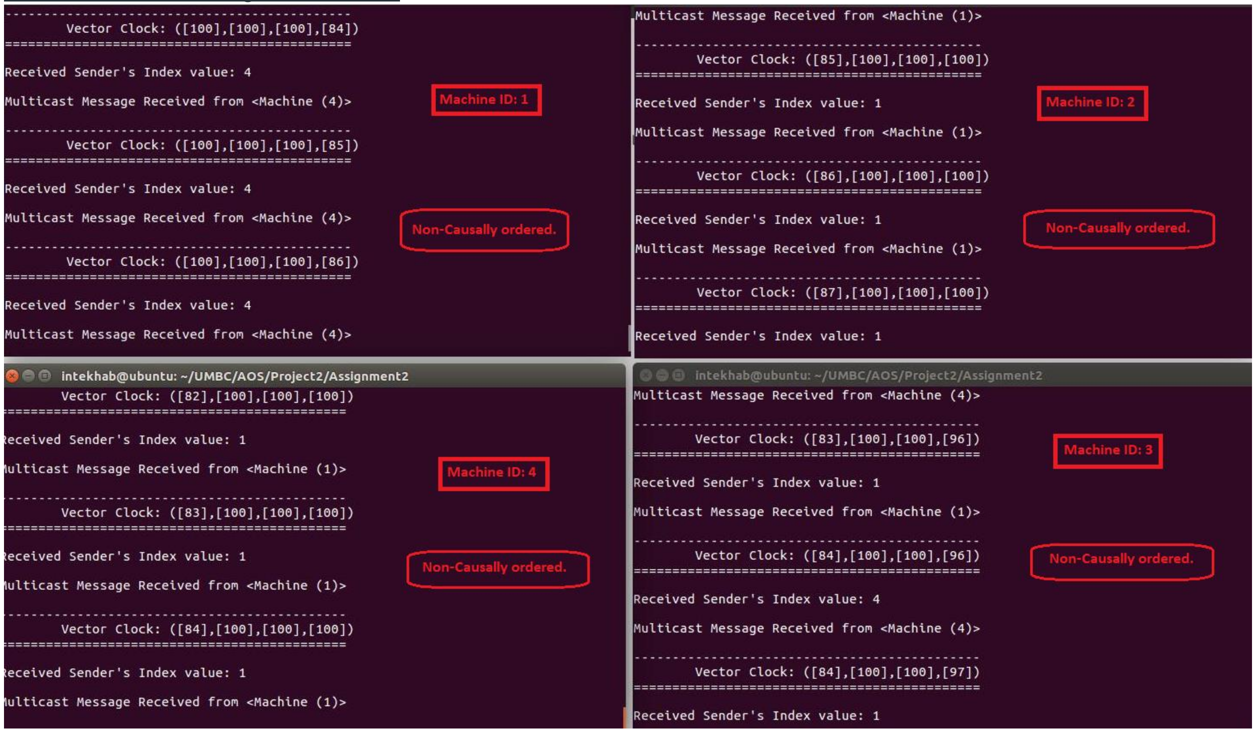
```
Multicast Message Sent from <Machine (2)>

        -----------------------------------------
        Vector Clock: ([111],[147],[200],[200])
=============================================
Multicast Message Sent from <Machine (2)>

        -----------------------------------------
        Vector Clock: ([111],[148],[200],[200])
=============================================
Multicast Message Sent from <Machine (2)>

        -----------------------------------------
        Vector Clock: ([111],[149],[200],[200])
=============================================
Multicast Message Sent from <Machine (2)>

        -----------------------------------------
        Vector Clock: ([111],[150],[200],[200])
=============================================
```

```
intekhab@ubuntu: ~/UMBC/AOS/Project2/Assignment2
Multicast Message Sent from <Machine (4)>

        -----------------------------------------
        Vector Clock: ([110],[110],[110],[110])
=============================================
##### Causality Violation! Message Buffered #####
Multicast Message Received from <Machine (1)>

        -----------------------------------------
        Vector Clock: ([111],[110],[110],[111])
=============================================
Multicast Message Received from <Machine (3)>

        -----------------------------------------
        Vector Clock: ([111],[110],[111],[111])
=============================================
*****Message Delivered from Queue for <Machine (2)>*****
        -----------------------------------------
```

```
intekhab@ubuntu: ~/UMBC/AOS/Project2/Assignment2
Multicast Message Sent from <Machine (3)>

        -----------------------------------------
        Vector Clock: ([111],[111],[146],[111])
=============================================
Multicast Message Sent from <Machine (3)>

        -----------------------------------------
        Vector Clock: ([111],[111],[147],[111])
=============================================
Multicast Message Sent from <Machine (3)>

        -----------------------------------------
        Vector Clock: ([111],[111],[148],[111])
=============================================
Multicast Message Sent from <Machine (3)>

        -----------------------------------------
        Vector Clock: ([111],[111],[149],[111])
=============================================
```

On the other hand, in non-causal, the messages are delivered as they are received which may be out of any causal order. A quick overview of how the mechanism is implemented may be understood better by the following screenshot:

Non-Causal Ordering Mechanism:



```
--------------------------------------------
        Vector Clock: ([100],[100],[100],[84])
============================================

Received Sender's Index value: 4

Multicast Message Received from <Machine (4)>

--------------------------------------------
        Vector Clock: ([100],[100],[100],[85])
============================================

Received Sender's Index value: 4

Multicast Message Received from <Machine (4)>

--------------------------------------------
        Vector Clock: ([100],[100],[100],[86])
============================================

Received Sender's Index value: 4

Multicast Message Received from <Machine (4)>
```

```
Multicast Message Received from <Machine (1)>

--------------------------------------------
        Vector Clock: ([85],[100],[100],[100])
============================================

Received Sender's Index value: 1

Multicast Message Received from <Machine (1)>

--------------------------------------------
        Vector Clock: ([86],[100],[100],[100])
============================================

Received Sender's Index value: 1

Multicast Message Received from <Machine (1)>

--------------------------------------------
        Vector Clock: ([87],[100],[100],[100])
============================================

Received Sender's Index value: 1
```

```
intekhab@ubuntu: ~/UMBC/AOS/Project2/Assignment2
        Vector Clock: ([82],[100],[100],[100])
============================================

eceived Sender's Index value: 1

ulticast Message Received from <Machine (1)>

--------------------------------------------
        Vector Clock: ([83],[100],[100],[100])
============================================

eceived Sender's Index value: 1

ulticast Message Received from <Machine (1)>

--------------------------------------------
        Vector Clock: ([84],[100],[100],[100])
============================================

eceived Sender's Index value: 1

ulticast Message Received from <Machine (1)>
```

```
intekhab@ubuntu: ~/UMBC/AOS/Project2/Assignment2
Multicast Message Received from <Machine (4)>

--------------------------------------------
        Vector Clock: ([83],[100],[100],[96])
============================================

Received Sender's Index value: 1

Multicast Message Received from <Machine (1)>

--------------------------------------------
        Vector Clock: ([84],[100],[100],[96])
============================================

Received Sender's Index value: 4

Multicast Message Received from <Machine (4)>

--------------------------------------------
        Vector Clock: ([84],[100],[100],[97])
============================================

Received Sender's Index value: 1
```

3) Learnings:

By implementing this algorithm, I learned how to merge a client and server program into a single program and connect and communicate with all the other nodes in the system.

I also learned how the causality conditions ensure the correct behavior of applications where causality is desired like in most social networks.

4) Issues:

While implementing this algorithm, initially it was bit of a challenge to connect all the nodes to each other and to make every node communicate with all the other nodes. I solved this problem by making a thread for listening to connections as a server does and simultaneously sending connection requests to other nodes from the main thread as a client does.

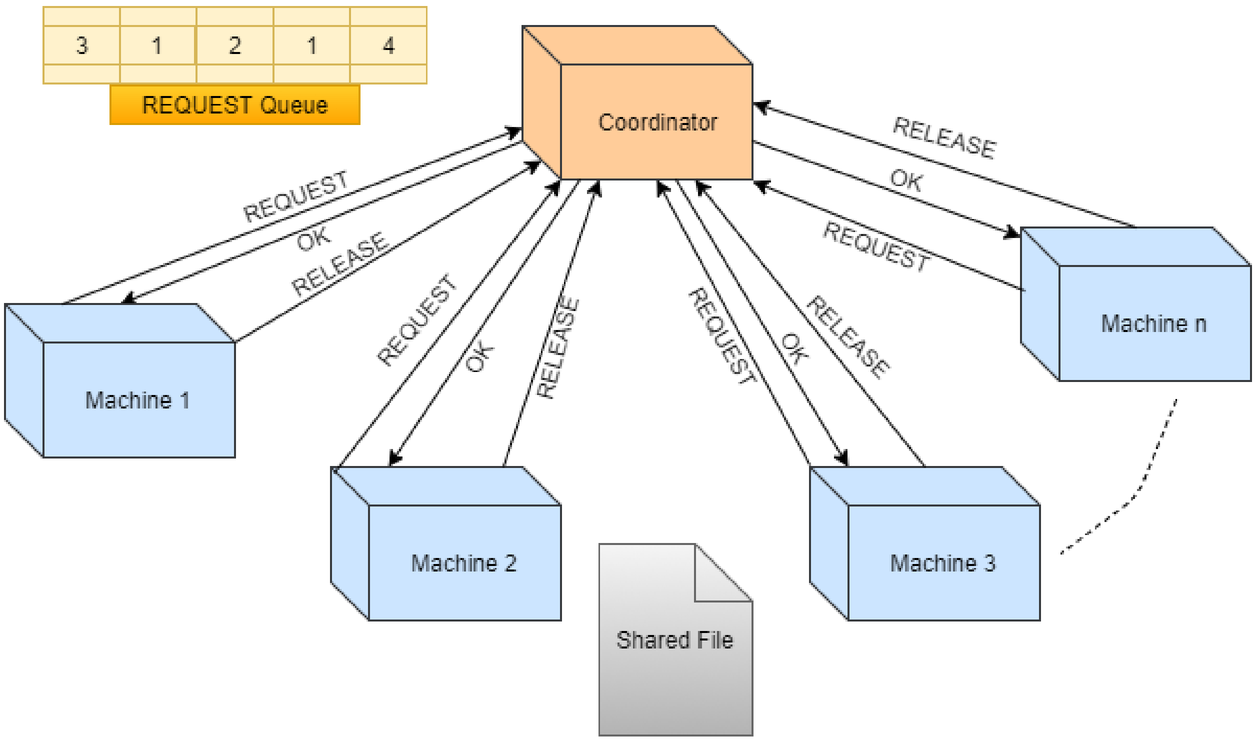
Then it was a challenge to store all the socket file descriptors to be able to communicate with desired nodes later. To solved this, I made an array of structure where each element of array signifies each node in the system and the contents of the structure include all the relevant information about that node. Once this was done, communication between nodes became easy and multicasting messages and checking messages to satisfy causality conditions was not a big problem.

Finally, another problem was simulating delayed packets due to TCP connections for deliberately violating causality condition and then rectifying it. To solve this, I added random wait timings before the send and receive operations. With this I was able to simulate some delayed packets after a lot of failed attempts.

<u>Bonus Assignment</u>:

In this Assignment, I again used Client Server Architecture to implement the Centralized Algorithm for Distributed Locking.

1) **Design:**



2) **Implementation:**

The Coordinator is a server and all the other machines/processes are clients. The server keeps accepting the connections till all desired machines are connected to it. For each client connection, server (Coordinator) spawns a new thread for communicating with respective client.

As soon as any client gets connected to server, it sends a REQUEST message to the Coordinator. The corresponding client thread on the server pushes this request message to the Queue.

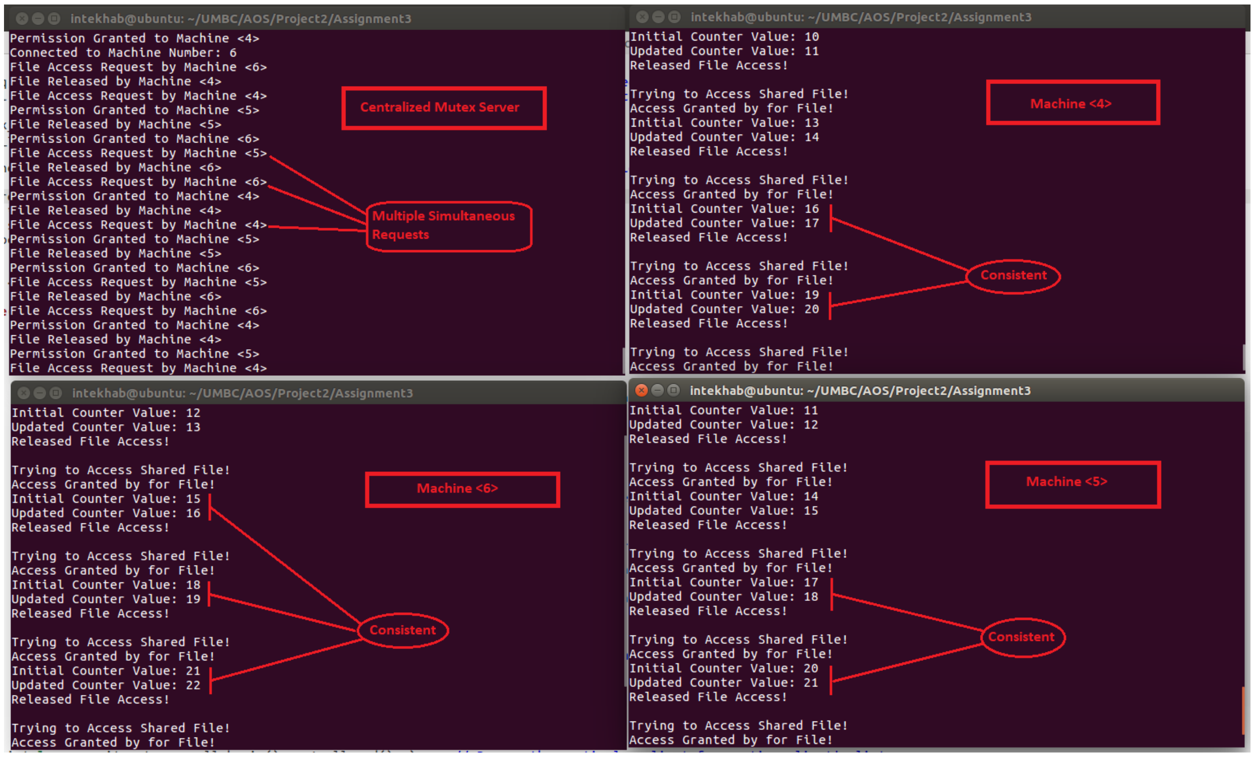
After all the clients have connected to the server, the coordinator thread is spawned.

The coordinator thread checks if the critical section is free i.e. no one is using the shared file, and grants access to the first request (the front) from the queue. This access is granted by sending an “OK” message to respective client process. On receiving the OK message, client opens the file, reads the counter, increments it by one and writes back to the counter value to the file. When this is done, client sends a RELEASE message to coordinator.

The next request from the queue waits on the conditional variable till the other process releases the shared file i.e. releases the lock and signals on the conditional variable. The waiting process wakes, and the coordinator grants it access to the shared file.

All the processes achieve consistent access to the shared file without any deadlock situation due to correct locking, unlocking, waiting and signaling.

A quick overview of how the algorithm is implemented may be understood better by the following screenshot:



```
Permission Granted to Machine <4>
Connected to Machine Number: 6
File Access Request by Machine <6>
File Released by Machine <4>
File Access Request by Machine <4>
Permission Granted to Machine <5>
File Released by Machine <5>
Permission Granted to Machine <6>
File Access Request by Machine <5>
File Released by Machine <6>
File Access Request by Machine <6>
Permission Granted to Machine <4>
File Released by Machine <4>
File Access Request by Machine <4>
Permission Granted to Machine <5>
File Released by Machine <5>
Permission Granted to Machine <6>
File Access Request by Machine <5>
File Released by Machine <6>
File Access Request by Machine <6>
Permission Granted to Machine <4>
File Released by Machine <4>
Permission Granted to Machine <5>
File Access Request by Machine <4>
```

```
Initial Counter Value: 10
Updated Counter Value: 11
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 13
Updated Counter Value: 14
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 16
Updated Counter Value: 17
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 19
Updated Counter Value: 20
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
```

```
Initial Counter Value: 12
Updated Counter Value: 13
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 15
Updated Counter Value: 16
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 18
Updated Counter Value: 19
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 21
Updated Counter Value: 22
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
```

```
Initial Counter Value: 11
Updated Counter Value: 12
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 14
Updated Counter Value: 15
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 17
Updated Counter Value: 18
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
Initial Counter Value: 20
Updated Counter Value: 21
Released File Access!

Trying to Access Shared File!
Access Granted by for File!
```

3) Learnings:

By implementing this algorithm, I learned how the Centralized algorithm for distributed locking mechanism works. I also learned how to use conditional variables along with mutex locks for waiting and signaling to avoid deadlock situations.

4) Issues:

While implementing this algorithm, I was unsure of how to grant access to shared resources and apply locking mechanism in distributed systems. For solving this problem, I initially used locking with various flags. Then I realized this could be better implemented using conditional variables and applied the same.

-x-