Project -2

Akanksha Bhosale
CZ44818

Introduction:

The design document describes program design of a n-node distributed application that consists of following implementations of Berkeley Synchronization,Non-total ordered multicast and totally ordered multicast.

Design Approach & Learnings:

Assignment 1 :

- In Berkeley algorithm implementation, the client-server architecture is used with TCP transport layer protocol for communication.
- The time daemon first asks all the nodes involved in a communication for their current logical clock and sends along its own time which is a random no. generated by the time server.
- All other processes initially having their logical clock zero generates their logical clock as a random number and distributed application nodes send the difference between their logical clock and time daemon's received clock to the Time daemon or Server.
- The time daemon then after receiving the expected logical clocks' differences from all of the nodes initiates the procedure of synchronization.
- It takes all the differences and computes the average of it. Then sends back the adjustment time to every node and also synchronizes its own logical clock by adding this average to its own logical clock.
- Thus this is a basic single server and multiple clients implementation performing logical clock synchronization.

Issues and Learnings:

1. The time daemon starts calculating the average or procedure of synchronization as soon as any of the node's logical clock is received while it should wait for all the involved node's logical clocks and then should start the computation.

   Resolution:

   Synchronization is started by the time daemon only when connected node's count is equal to the no. of expected nodes. For this one parameter indicating the no. of processes involved in a communication is passed through command line while running the time daemon or Server code. Thus learnt the concept of conditional waits during inter process communication.

Steps for execution:

Makefile consists of following steps for executing an application :

- In order to clean all the compiled .out files "**make clean**" cleans out files present in current directory by executing the command rm *.out
- To compile the server and client "**make compile**" compiles as:
  gcc s5.cpp -o s5.out -lpthread
  gcc c6.cpp -o c6.out

- To run a server executable "**make runserver**" runs
  ./s5.out 9090 3
- Similarly, in order to run a client "**make runclient**" runs the client
  ./c6.out 127.0.0.1 9090

Here 9090 is the port no. where the time daemon is running and all other distributed nodes in a communication connects to it. Also 3 is the no. of processes involved in a communication.

Assignment 2 :

Part 1 – Non-total order Multicast

It is implemented with peer to peer architecture communicating through sockets.
In non-total multicast approach, every process can serve as a server and a client where the communication happens through the ports specified while running the process or that particular node.
In order to implement this there are separate threads for performing the tasks of the server and client where the server thread reads the data sent by the client thread.
Now in inter process communication where multiple processes are involved in a communication any process can send a message to any other process in a group.
In the implementation the no. of processes is being kept fixed as three. So every process binds itself to the port provided as a first argument and connects to the other two ports along with its own port
Thus if execuatble.out p1,p2,p3 is given then the process will bind or acts as a server for its own local port i.e. p1 and connects to the other two ports p2 and p3 along with itself.
As a result the client process thread can send the data to itself and to other clients in the group.
For second process the order will be ./executable p2,p1,p3 as p2 is its local host port to which it binds and sends the data to its own port and other mentioned two ports.
Here any process or node can send a message i.e. a random number to all the processes on mentioned ports and other processes delivers them to the application layer in any order it is received by each of the processes i.e. FIFO
Thus it does not follow the total ordering mechanism.

Issues and Learnings:

During implementation communication was not happening between the processes while on a single port each of the client and server threads were working perfectly.

Resolution:
All the node's ports are saved in an portarray and then every thread while sending or writing the data iterates over all of the ports as a result of which data is communicated across the nodes and multicasting worked between the processes.

Steps for execution:
Makefile consists of following steps for executing an application :

```
compile:
gcc seq4.cpp -o seq4.out -lpthread
run1:
./seq4.out 9090 9091 9092
run2:
./seq4.out 9091 9090 9092
run3:
./seq4.out 9092 9091 9090
```

clean:
rm *.out

Here if we analyze the output of the execution (attached in the zip file) we observe that even if one of the processes sends the data to other nodes in a group in between if any other node sends the data, it is received and delivered to the application processes or nodes
Every received message contains the port of the sender that sent the message.

Part 2 – Total order Multicast

As per the implementation of sequencer algorithm, every node sends its logical clock to the sequencer and sequencer multicasts the message to all other nodes in the group along with the incremented global sequence number.
Every node maintains its own local message buffer where messages are buffered and are delivered to the application only when it satisfies following conditions:

- Si + 1 = GS from sequencer
- It has received the message from the sequncer

Steps for Execution:

- Makefile consists of following steps for executing an application :
- In order to clean all the compiled .out files "**make clean**" cleans out files present in current directory by executing the command rm *.out
- To compile the server and client "**make compile**" compiles as:
- g++ node.cpp -o node.out -lpthread
- g++ sequencer.cpp -o sequencer.out -lpthread

- ./sequencer.out 9092 9090 9091 9093
- ./node.out 9090 9091 9093 9092
- ./node.out 9091 9090 9093 9092
- ./node.out 9093 9090 9091 9092

Issues and Learning:

Faced issues in implementing multicast between sequencer and nodes with TCP . In order to resolve it followed algorithm steps and referred online resources.

Design Tradeoff :

- Processing overhead is one of the primary concerns in socket communication. To lower this overhead, for every active incoming requests,separate function handler processrequests handles it.
- The handler abstraction is providing a secure layer for interprocess communication.

Possible improvements :

- To reduce the communication overhead and improve the performance time we can have our own fast socket design implementation.
- Efficient buffer management can improve interoperability.
- Efficient logging can help tracking any unexpected events.