CMSC 621: ADVANCED OPERATING SYSTEMS

PROJECT 2 REPORT

Introduction

This document describes the details of the project 2. There are 3 parts in the project Clocks, Multicast, and Commit (bonus assignment). All three functionalities are implemented. The document also describes more on the technical side of the project. The logs of all the test run can be found in the **“logs”** folder.

Prerequisites

All three functionalities are implemented using the CPP multicast UDP socket and all are distributed and run as a separate process. To enable multicast support in the operating system you need to enable broadcast mode on your network interface. Also, you will have to add an IP route in the operating system such that all the packets on the given IP address should be redirected to the interface. The following are the steps to enable the multicast mode.

1. ifconfig lo multicast
2. route add -net 224.0.0.0 netmask 240.0.0.0 dev lo

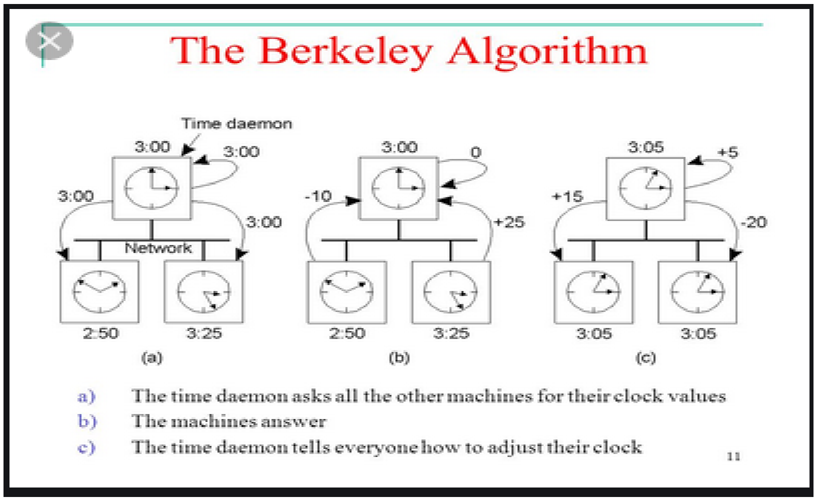
How to build a project

There is a **MakeFIle** along with the source code. You can build individual binaries also. It also supports clean command. Doing "make clean" will remove all binaries and generated object files.

“Make” command will generate 4 executables

- Berkeley
- Causal
- NonCausal
- DistributedME

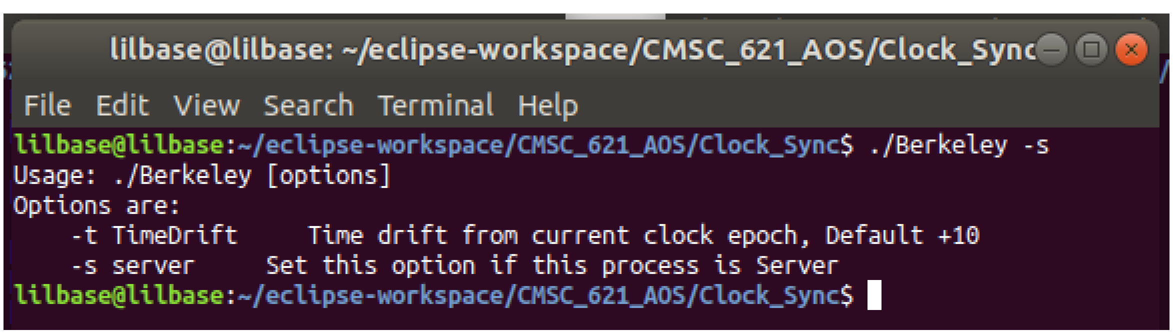
Assignment1: Berkeley



(Image is taken from https://slideplayer.com/slide/5237963/)

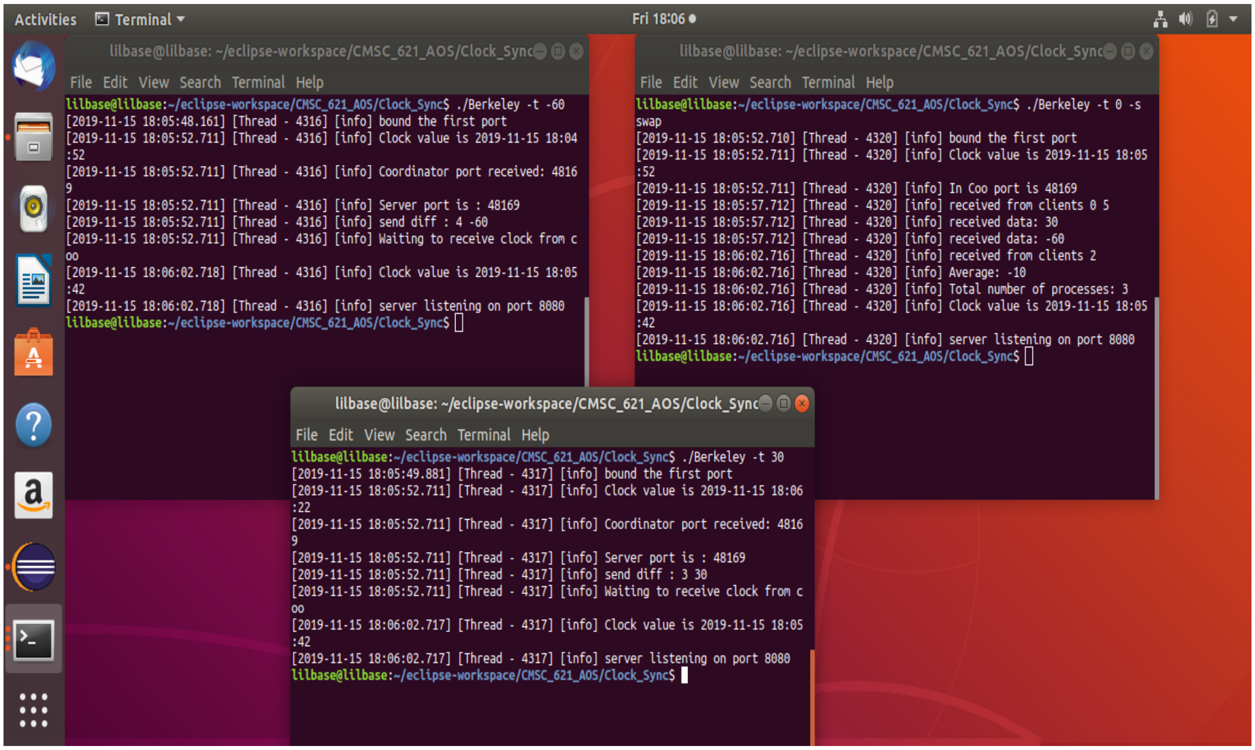
Berkeley time synchronization is implemented using UDP multicast sockets. One of the processes acts as a coordinator and proactively starts the algorithm by sending its own time. All the Clients receive the time sent by the Server, calculates their time drift and respond to the server on the point to point connection. Then the server calculated the difference of the average and send back the avg to all clients and adjusts their own clock. Receiving clients uses this average to synchronize their clock as well. The logs of the test run on Berkeley is in the logs folder

Running Berkeley executables



-t : This option sets the time drift from current local time. Also accepts negative values

Berkeley test run screenshot

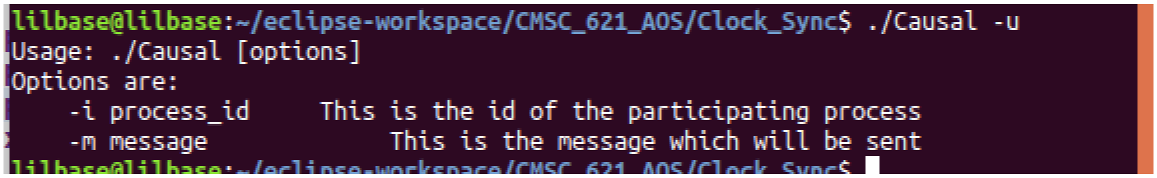


In the test run, 2 processes were acting as Clients and one Server. The clients wait until the Server is up. In the screenshot we can see at the end of the protocol all three processes sets it's time to **" 2019-11-15 18:05:42"**

Assignment2.1: Causal Ordering

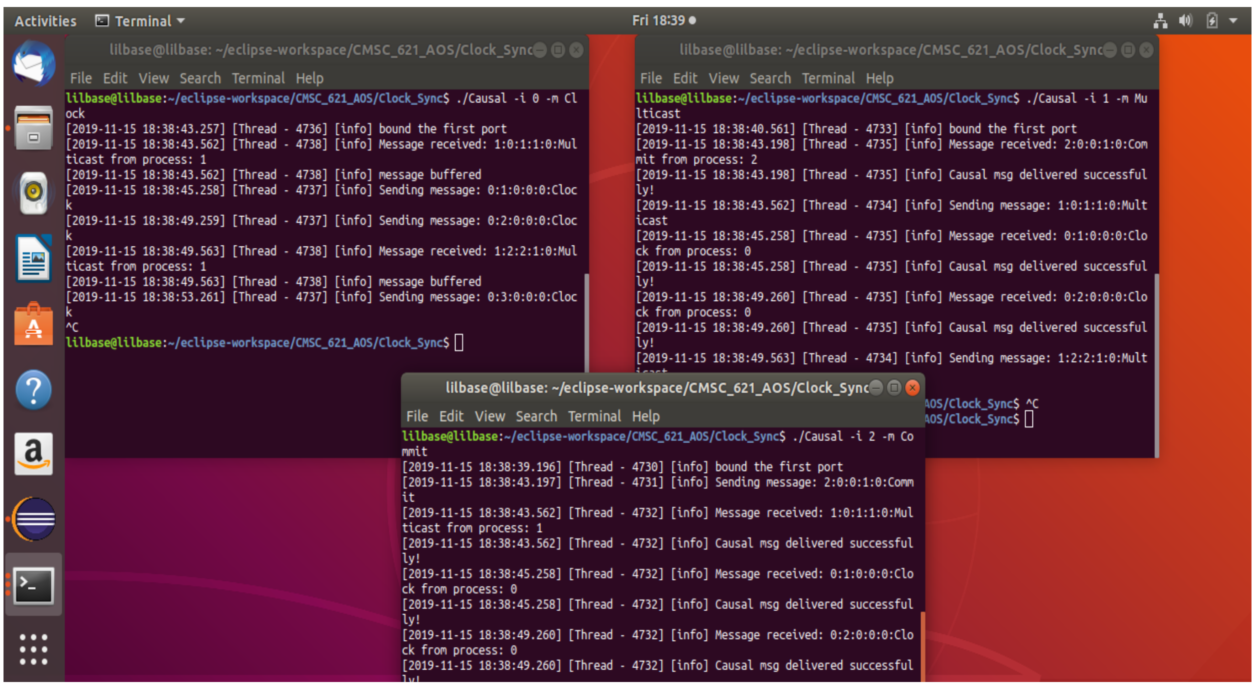
In causal ordering each process multicasts messages to all other processes. But the processes should receive the multicast messages in the same order they were send satisfying the causality. Causality is implemented by buffering the message if received out of the order. If all the rules of Causality are satisfied then the message is immediately delivered to the application else the message is buffered.

Running Causal executables



- process_id: This is the process id of the participating process. Accepts integers
- Message: This is the string message which needs to be sent.

Causal test run screenshots:



In the screenshot, we can see that the "process 0" multicast message received from "process 1" is buffered because of the violation of the causality property. Where are process 1 and process 2 were able to deliver the message to application successfully? The logs of the above run can be found in the "logs" folder with names "CausalOrdering_4730.txt, CausalOrdering_4733.txt, and CausalOrdering_4736.txt".

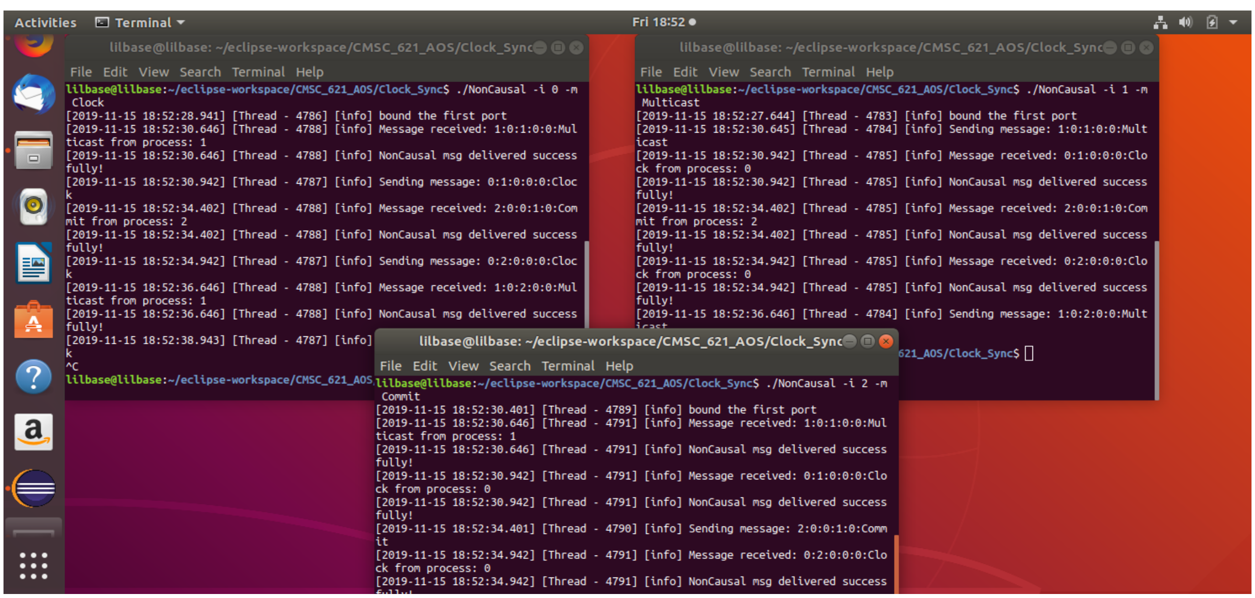
1. Each process creates and binds to the multicast socket.

2. Each process run with 2 threads one acts as a multicast message sender and one acts as a multicast message receiver
3. On the sender thread side, the process increments it's own vector clock by 1 and sends the message in this "processid: <comma separated vector clock>: message" format.
4. On the receiver side, the received message is split using ":" and check for the sender process id. Using this data Causality rules are verified and the decision is taken if the message can be delivered to the application or it should be buffered

Assignment2.2: Non-Causal Ordering

In non-causal ordering each process multicasts messages to all other processes. On the receiver, side message can be received in any order. No message is checked for the Causality rule. All the messages received in whatever order are delivered to the application. Executing the Non-Causal ordering binary is similar to the Causal Ordering binary i.e command line arguments are the same for both.

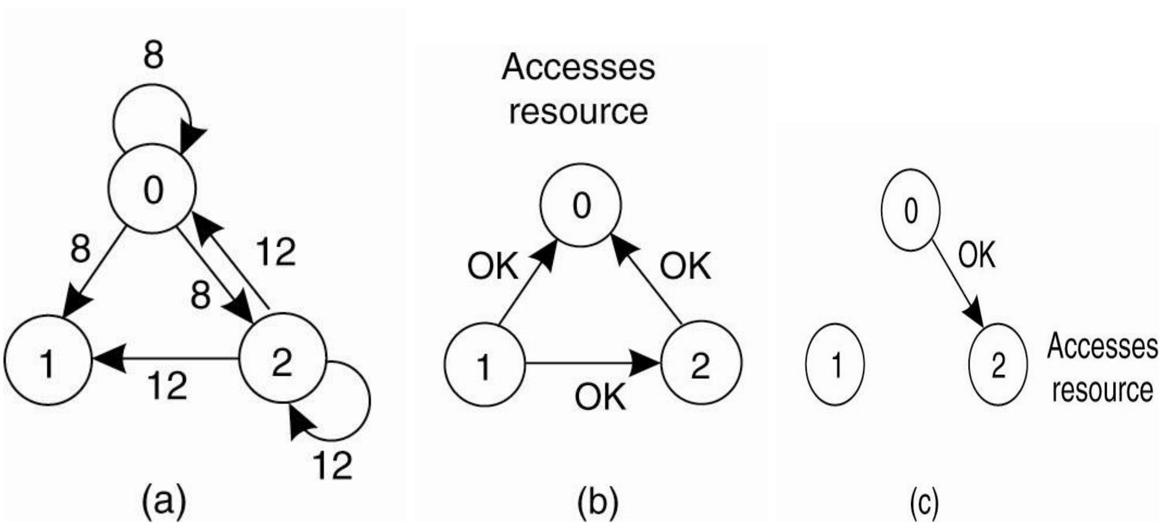
Non-Causal test run screenshots:



We can see from the screenshot that, no message is checked for the Causality rules and delivered to the application in the received order. Logs of this run are attached in the logs folder.

Bonus Assignment: Distributed Mutual Exclusion

I have implemented the "Distributed Mutual Exclusion Algorithm". Every process in the multicast group is involved in this algorithm by sending the multicast request to all other processes requesting to use the shared resource. In our case, the shared resource is a text file that holds a counter value and updated by each process. Another process read this updated counter value before incrementing it.



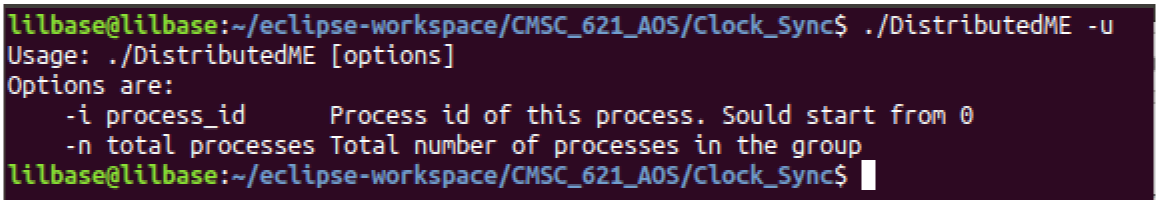
(Above diagrams are taken from the ppt of the Prof. Nilanjan Banarjee)

Distributed Mutual Exclusion algorithm has three cases:

1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender
2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request
3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins

The program makes use of the UDP multicast socket to send a request to all other processes in the group. On each receiver side above three cases are checked. If the process received the reply from all other processes then the request is granted for that service, subsequently, it sends the reply back to all other processes who sent a request message. Eventually, all the processes will get a chance to update to counter in the file.

Running Distributed Mutual Exclusion executables



- process_id: This is the process id of the process
- total_processes: The number specifying the total number of processes in the multicast group

Distributed Mutual Exclusion - test run screenshots:

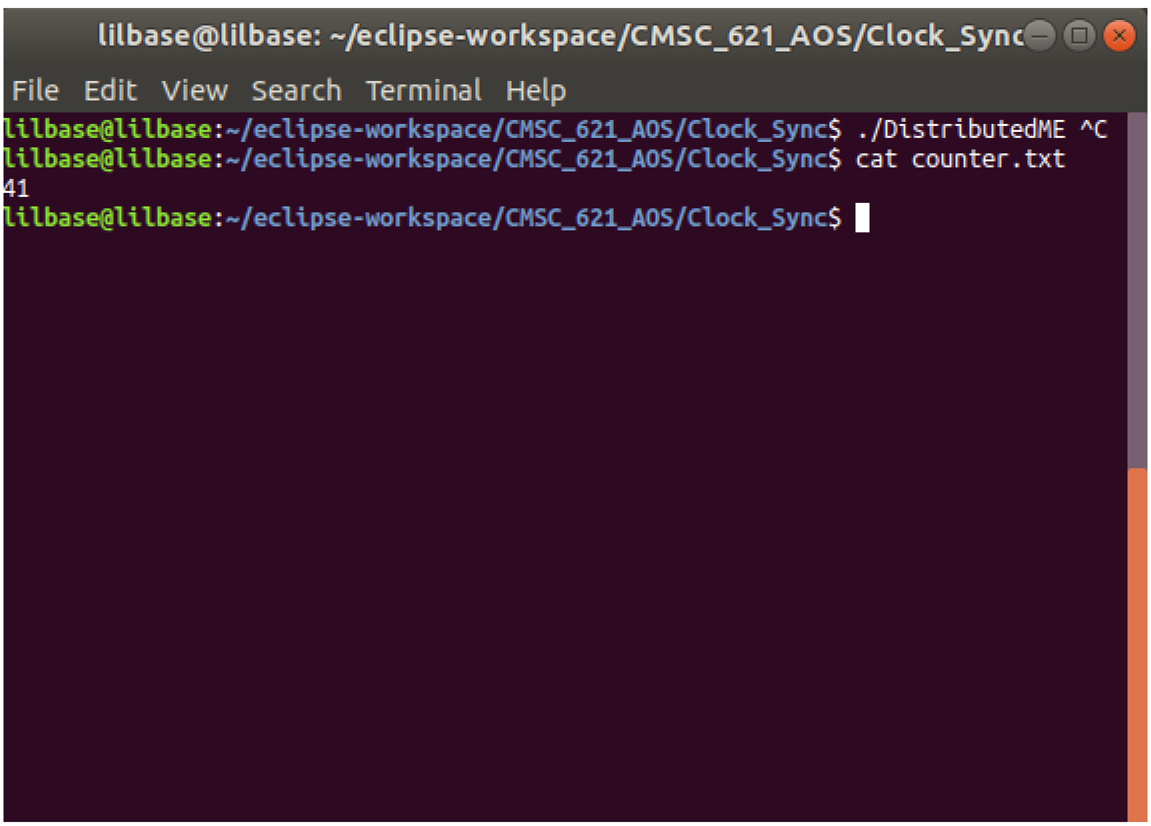


Figure 3.1: Current counter value in the file.

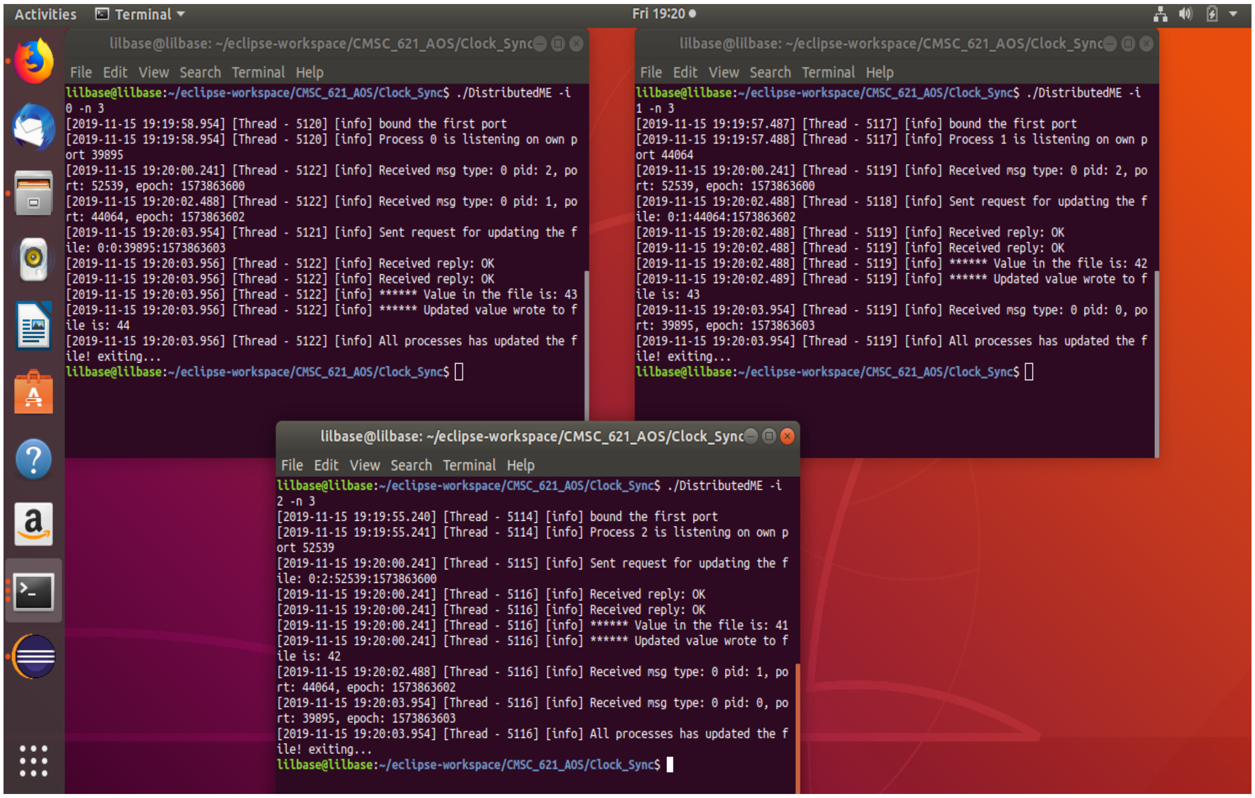


Figure 3.2: Each process updated the counter file one by one

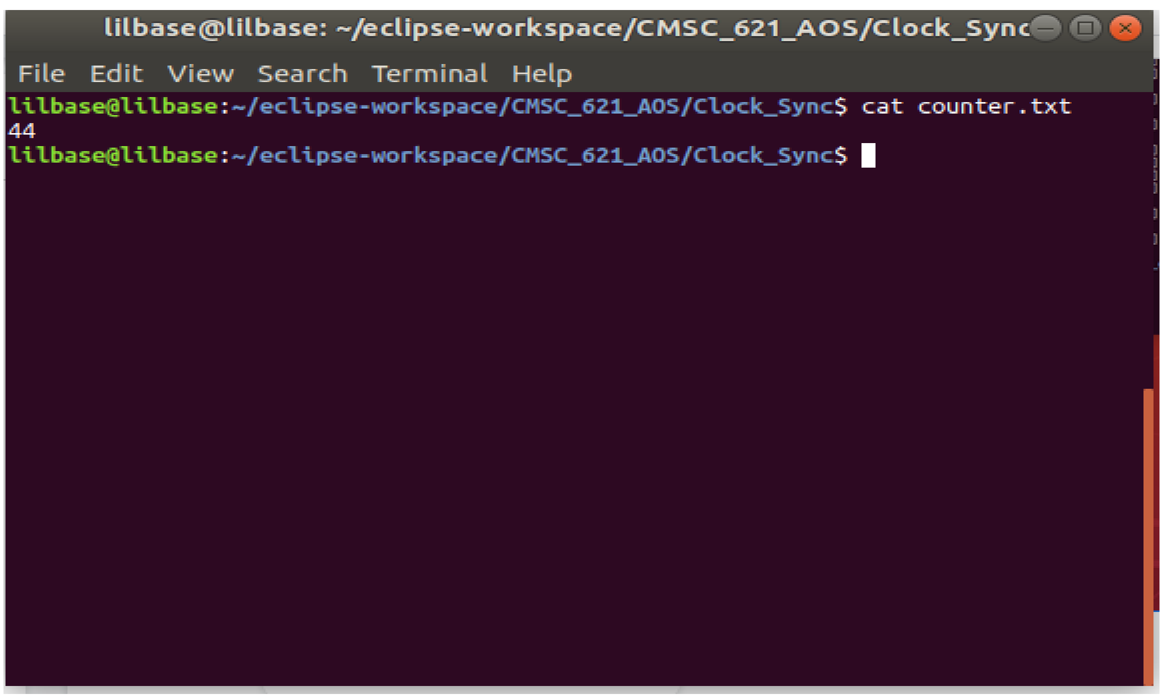


Figure 3.3: Updated counter value after the test run

How to automate/run above programs

I have provided a PYTHON script which will help you in executing the multiple instances of the above given programs without any manual interventions. Please use below commands to execute the binaries of the 3 project parts

- Berkely: "./helper.py berkeley"
- Causal: "./helper.py causal"
- NonCausal: "./helper.py noncausal"
- Bonus: "./helper.py bonus"

Above commands will execute the given binaries in the different shell.

What I have learned from this Project

- The major learning from this project was writing a multicast programs using C++. Also learned to write a distributed program
- Learned how Berkeley clock algorithm works behind the seen and implemented same using actual system clock by providing command line option to control drift seconds
- Learned how Distributed Mutual Exclusion algorithm works and implemented same as a part of Bonus Assignment. All the participating processes takes part in coming to consensus that which process will get a chance first to use/update/modify the shared resource in the distributed system
- Also learned the advantages of Causal Ordering system over Non-Causal system and how it helps in maintaining the correct system state by buffering the out of order messages
- How to execute binaries from the python programs in their own shell

Challenges/Issues faced in the implementation of this Project

- First challenge was to understand how to write a socket program which works in multicast environment. Online resources helped me in understanding how multicast socket works and implemented in C++
- The prerequisites for the multicast socket implementation was not documented on the various online resources. Little but of googeling helped me in configuring/satisfying the prerequisites by enabling multicast mode on the network interface and adding the IP route in the system for multicast IP
- In the bonus assignment keeping the process alive after the shared resource is used was challenging. Solved the issue by comparing the number of replies send so far by the process. If number of replies are equal to total number of processes in the group – 1. Then it means process now can gracefully shutdown to avoid deadlock.

Creativity added to this Project

- Made a used of multicast socket concept of C++ for all the three parts of the project
- Researched and used "**spdilog**" the multithreaded logging library. I am using a rotating file appender and console appender. Thus, this logger logs the debug messages to the output stream and files simultaneously
- Make use of standard C++ practices by splitting declaration and definitions to header(.h) and source files (.cpp) respectively

References:

- https://stackoverflow.com/questions/19555121/how-to-get-current-timestamp-in-milliseconds-since-1970-just-the-way-java-gets
- https://github.com/RashmiPrava21/OperatingSystems-Clocks-Multicast-and-Commit
- https://www.tldp.org/HOWTO/Multicast-HOWTO-6.html
- https://www.gnu.org/software/libc/manual/html_node/Socket_002dLevel-Options.html#Socket_002dLevel-Options

- https://www.thegeekdiary.com/how-to-configure-multicast-on-an-ip-address-interface/
- https://stackoverflow.com/questions/16007311/multicasting-on-the-loopback-device
- https://stackoverflow.com/questions/2364574/receiving-responses-from-n-number-of-clients-in-reply-to-a-broadcast-request-o
- https://www.geeksforgeeks.org/vector-erase-and-clear-in-cpp/