COL 724 Assignment 2

Nikhil Unavekar
2020CS10363
Oct 13 2023

1. Messaging Layer-

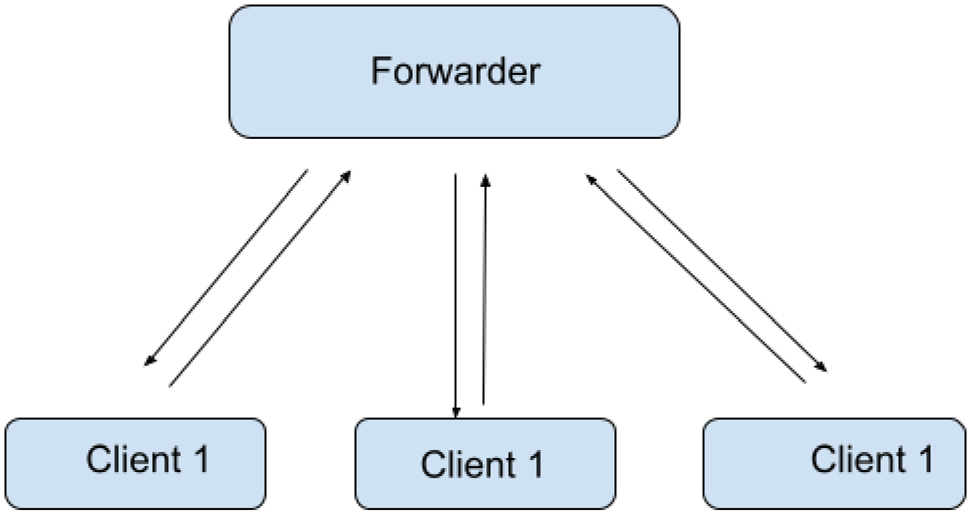
To implement the messaging layer for this project, we have utilized the ZeroMQ (pyzmq) package in Python. This is essential because our system supports both totally ordered and unordered messaging. In this section, we will describe the structure of the messaging layer for multicasting and explain how messages are sent and received. Then, we will delve into unordered messaging. Subsequently, we will explain how to achieve totally ordered messaging. Lastly, we will present a test case and the results of that test case. Towards the end of this section, we will provide comprehensive information for running the code.

2. Multicast Messaging-

In this subsection, we will first outline the structure of our multicast messaging layer, followed by an explanation of the sending and receiving functions.

3. Messaging Layer Structure-

ZeroMQ offers various types of sockets such as PUB, SUB, ROUTER, DEALER, REQ, REP, and more. While communication is not possible between any pair of these sockets, certain pairs are valid. To create a message-passing layer for totally ordered multicast, we have chosen the PUB/SUB pattern. PUB represents the publisher, which binds to an address, and SUB stands for the subscriber, which can connect to the address of PUB. Given that we have multiple clients wishing to communicate with each other, we employ a ZeroMQ forwarder device or proxy. This device is responsible for receiving messages from a client or subscriber via a SUB socket and multicasting them to all clients. In the case of totally ordered multicast, when a client multicasts a message to others, it is crucial for the client itself to receive the message.



Each forwarder device is equipped with a frontend and a backend address. In this implementation, we bind the frontend to a SUB socket and the backend to a PUB socket. Clients also have both a PUB and a SUB socket. We establish a connection by linking the PUB socket of a client to the SUB socket of the forwarder, and vice versa, connecting the SUB socket of the client to the PUB socket of the forwarder.

An additional socket for synchronization is employed within the forwarder. Imagine a scenario where multiple entities, such as four banks, wish to engage in multicast messaging. To synchronize them effectively, one approach is to ensure that all parties are successfully subscribed before commencing messaging. Below, you will find the forwarder code for reference.

In the forwarder code, following client subscriptions, a perpetual loop is executed within the zmq.proxy function. This loop, if everything proceeds as expected, ensures that the code does not progress beyond this point, thereby preventing it from reaching the except or finally sections.

```python
def forwarder_tom(self):
    try:
        self.context = zmq.Context() # Socket facing clients
        self.frontend = self.context.socket(zmq.SUB)
        self.frontend.bind("tcp://*:5559")
        self.frontend.setsockopt(zmq.SUBSCRIBE, b"")
        # Socket facing servics
        self.backend = self.context.socket(zmq.PUB)
        self.backend.bind("tcp://*:5560")


        # Socket to receive signals
        self.syncservice = self.context.socket(zmq.REP)
        self.syncservice.bind('tcp://*:5561')

        # Get synchronization from subscribers
        subscribers = 0
        while subscribers < self.SUBSCRIBERS_EXPECTED:
            # wait for synchronization request
            msg = self.syncservice.recv()
            # send synchronization reply
            self.syncservice.send(b'')
            subscribers += 1
            print("+1 subscriber (%i/%i)" % (subscribers, self.SUBSCRIBERS_EXPECTED))
        self.backend.send("start".encode())

        zmq.proxy(self.frontend, self.backend)

    except Exception as e:
        print(e)
        print("bringing down zmq device")
    finally:
        pass
        self.frontend.close()
        self.backend.close()
        self.context.term()
```

The client-side code for multicasting and synchronization is illustrated in the following figures.

```python
class client_(object):
    def __init__(self, config):
        self.identity = config.id
        self.c_socket()

    def c_socket(self):
        self.context = zmq.Context()

        # Multicasting
        self.client_receiver = self.context.socket(zmq.SUB)
        self.client_receiver.connect("tcp://localhost:5560")
        self.client_receiver.setsockopt(zmq.SUBSCRIBE, b'')

        self.client_sender = self.context.socket(zmq.PUB)
        self.client_sender.connect("tcp://localhost:5559")

        # unordered messaging
        self.client_deal = self.context.socket(zmq.DEALER)
        self.client_deal.setsockopt(zmq.IDENTITY, (self.identity).encode())
        self.client_deal.connect("tcp://localhost:5562")

        # synchronization with publisher
        self.syncclient = self.context.socket(zmq.REQ)
        self.syncclient.connect('tcp://localhost:5561')

        # sending a synchronization request
        self.syncclient.send(b'')

        # waiting for synchronization reply
        self.syncclient.recv()
        while True:
            self.msg = self.client_receiver.recv()
            if (self.msg).decode("ascii") == "start":  # All clients are connected, Start!
                break

    def send_tom(self, msg, seq_num = None):
        print("sending ", msg.decode("ascii"), " from ", self.identity,  " to all!")
        if seq_num != None:
            self.client_sender.send_multipart([self.identity.encode(), seq_num ,msg])  # to be able to send sequence number
        else:                                                                          # using sequencer
            self.client_sender.send_multipart([self.identity.encode(), b'', msg])

    def recv_tom(self):
        sender_id, seq_num ,msg = self.client_receiver.recv_multipart()
        print ("Received ", msg.decode("ascii"), " from ", sender_id.decode("ascii"), "!")
        if seq_num.decode("ascii") == "":
            return [msg]
        else:
            return [msg, seq_num]   # to be able to receive sequence number sent by sequencer

    def send_uno(self, receiver_id, msg):
        print("Sending ", msg.decode("ascii"), " to ", receiver_id.decode("ascii"), "!" )
        self.client_deal.send_multipart([b'', receiver_id, msg])
```

EMS OUTPUT DEBUG CONSOLE TERMINAL

4. Sending Function -

The transmission of messages is accomplished using the zmq.send_multipart() function. In the context of totally ordered multicasting, a unique process, acting as a sequencer or leader, is responsible for dispatching global sequence numbers. Therefore, the sending function, as depicted in the figure below, accepts an additional input: seq_num. A message can be sent from either a client or the sequencer to all participants. When a seq_num input is not specified, it signifies that the sender is a ‘client, and an empty string is sent in lieu of a sequence number. It's important to note that all inputs to the send functions must be in binary format, or more precisely, they should be byte-like objects. Further clarification on sequence numbers is provided in Section 2, which covers Totally Ordered Multicasting.

5. Receiving function-

The receiving function uses zmq.recv multipart() function, and return as output the message or both message and sequence number if it is not an empty string.

Totally Ordered Messaging: After establishing the messaging layer, the next step is to implement a mechanism for achieving totally ordered messaging. I have realized totally ordered multicasting through the utilization of a sequencer or leader. The sequencer maintains a global sequencing number, referred to as 'G,' initially set to 0. When a client multicasts a message to other clients and the sequencer, the clients receive the message and store it in their message queue. The sequencer, upon receiving a message 'M,' increments 'G' by 1 and then multicasts both 'G' and the message 'M' to all clients. The clients, on the other hand, maintain a local sequence number, denoted as 'L.' A client is responsible for delivering the buffered message 'M' to the application only when it receives a message in the format '<G, M>' from the sequencer, and when 'G' equals 'L+1'. Upon successful delivery of the message to the application, the local sequence number 'L' is incremented by 1. In the following code snippets, you can observe the implementation of the sequencer and the code responsible for delivering both totally ordered and unordered messages to the application. If you have any specific parts you'd like to paraphrase further or have additional queries, please feel free to ask.

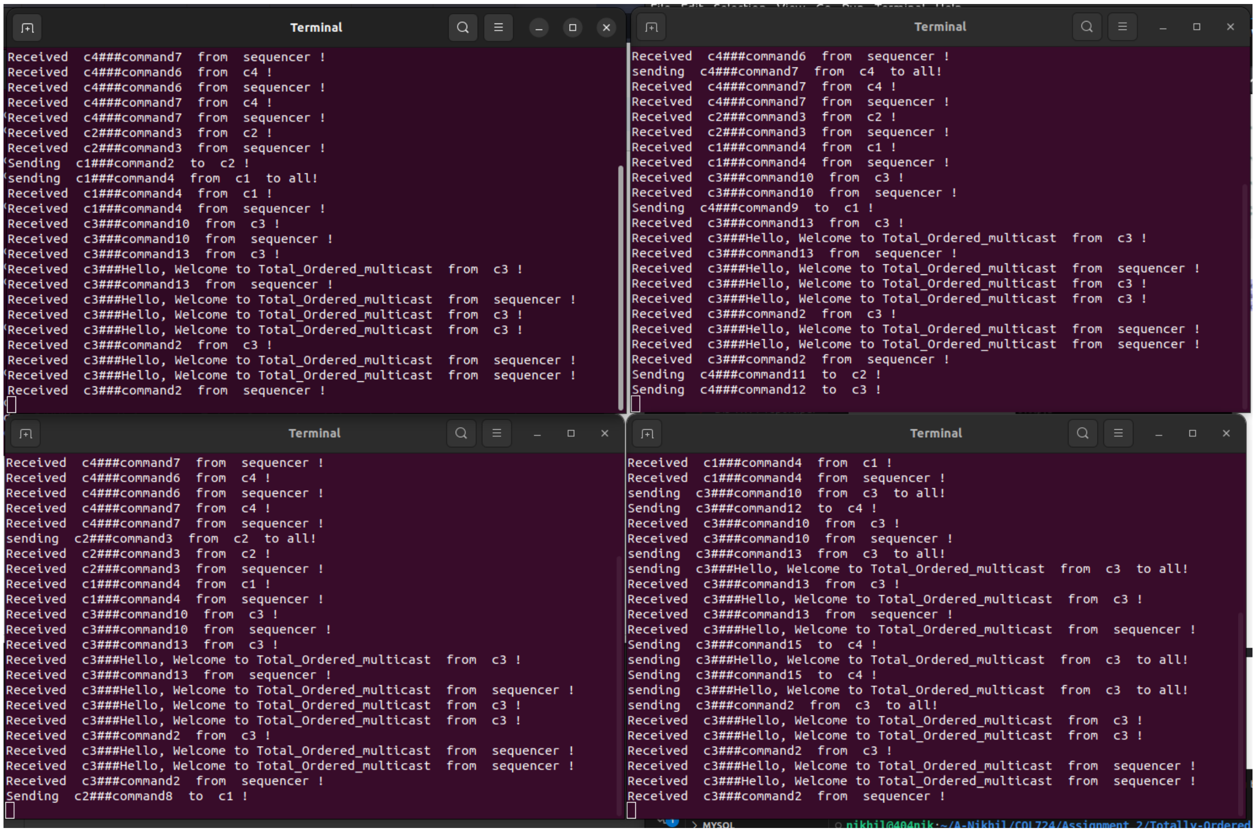
```python


if __name__ == "__main__":

    config, _ = get_config()
    c = client_(config)

    if config.id == "sequencer":
        g_seq_num = 0  # global sequence number (initial value set to 0)
        while True:
            msg = c.recv_tom()
            if len(msg) == 1:
                g_seq_num += 1
                c.send_tom(msg[0], str(g_seq_num).encode())


    else:
        q1 = Queue(100)  # queue for storing unorderd messages
        q2 = Queue(100)  # quese for storing totally ordered messages

        def rcv1(q):
            while True:
                msg = c.recv_uno()
                q.put(msg)

        def rcv2(q):
            while True:
                msg = c.recv_tom()
                q.put(msg)

        def send_commands():
            with open(config.com, 'r') as f:   # opening command files
                reader = csv.reader(f)
                cmd = list(reader)
                for item in cmd:
                    if item[0] == "Multicast":  # checking to see if it is a multicast message
                        if item[1] == config.id:
                            c.send_tom((config.id + "###" +item[2]).encode())
                    elif item[0] == "sleep":
                        time.sleep(int(item[1]))
                    else:
                        if item[0] == config.id:
                            c.send_uno(item[1].encode(), (config.id + "###" + item[2]).encode())


        def tom_uno():
            cmd_list = []
            seq_list = []
            l_seq_num = 0    # local sequence number (initial value set to 0)

            with open(config.o + "/test_result_" + config.id +"_tom.txt", 'wb', 0) as g, \
                        open(config.o + "/test_result_" + config.id +"_uno.txt", 'wb', 0) as h:
                while True:

                    if not q1.empty():     # checking for unorderd messages
                        #print(item1)
                        item1 = q1.get()
                        h.write((item1.decode("ascii") + "\n").encode()) # write received uorderd messages in the file

                    if not q2.empty(): # checking for totally orderd messages
                        #print(item2)
                        item2 = q2.get()
                        if len(item2) == 1:         # checking whether it is from a client or the sequencer
                            cmd_list.append(item2)
                        else:
                            seq_list.append(item2)
                    if len(seq_list) > 0:         # checking to see if there is any uprocessed message from sequencer
                        if int(seq_list[0][1].decode("ascii")) == (l_seq_num + 1):
                            if [seq_list[0][0]] in cmd_list:
                                idx = cmd_list.index([seq_list[0][0]])
                                g.write((cmd_list[idx][0].decode("ascii") + "\n").encode()) # delivering the received
                                del cmd_list[idx]                                             # message to the application
                                del seq_list[0]
                                l_seq_num += 1
                    time.sleep(0.01)           # For preventing from high cpu usage



        # threads for receiving messages and sending commands
        threading.Thread(target = rcv1  , args = (q1,)).start()
        threading.Thread(target = rcv2 , args = (q2,)).start()
```

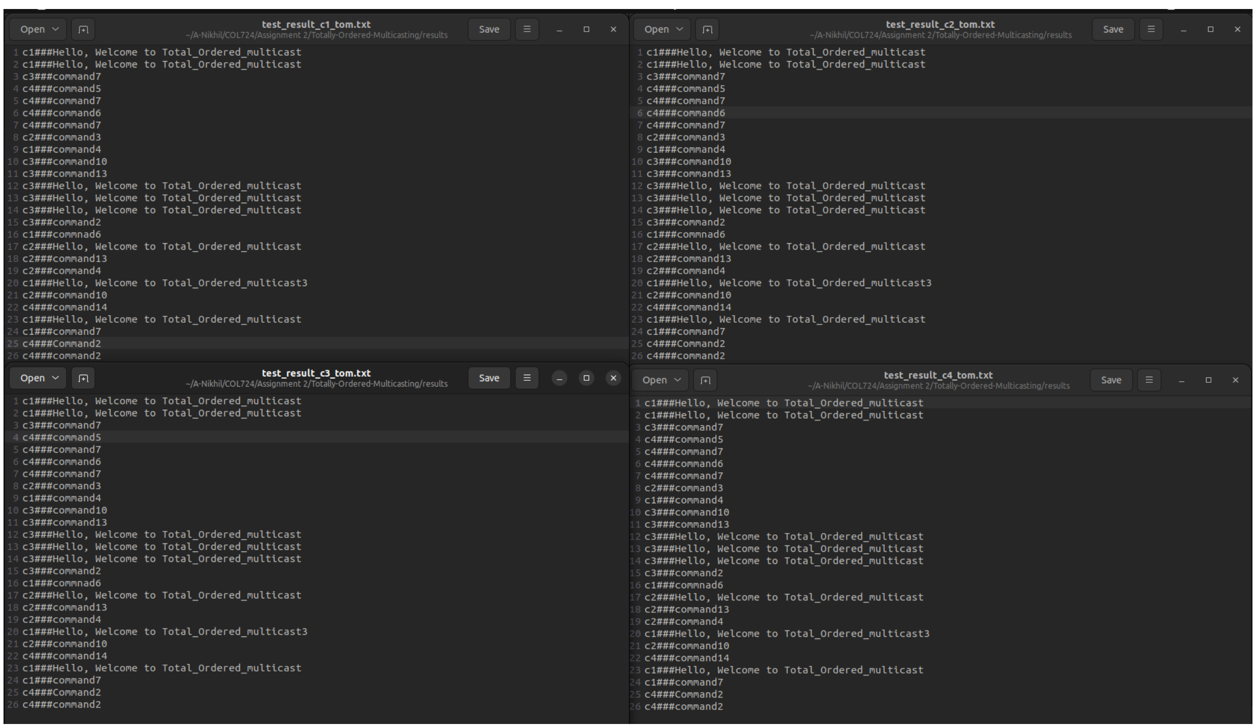
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
nikhil@404nik:~/A-Nikhil/COL724/Assignment 2/Totally-Ordered-Multicasting$
```

6. Testing and Results -

In the testing phase, we assume there are multiple clients that need to send messages in either a totally ordered or unordered manner. For this purpose, we use text files for each client. These text files contain three different types of operations. Below is an example of the content in one of these client files: The first type of operation starts with "Multicast," indicating that the client (e.g., "c1") should multicast "command1". The second type of operation starts with "sleep," specifying that client "c1" should not send any messages for 3 seconds. The third type of operation begins with the client's identity, signifying that "c1" should send "command2" to "c2". Each client has its own file with a similar format of operations. Clients read their set of operations from their files and execute them. The content of the test files for each of the four clients is provided in a specific folder. Each client records the messages it receives from others, whether they are ordered or unordered, in two separate files. The input test file, output directory for saving results, client identity, and the number of expected clients for the forwarder device must be specified. Below is an example of the order of received messages for each client. "Client identity + " is added to each command to indicate which client sent the message (though this is not required for the code to function correctly and can be removed within the code). Furthermore, each client maintains a file containing the messages it received in an unordered manner. The received unordered messages for each client are displayed, along with an indication of which client sent each message at the beginning. I have considered 4 clients + 1 sequencer.


Terminal
Received c4###command7 from sequencer !
Received c4###command6 from c4 !
Received c4###command6 from sequencer !
Received c4###command7 from c4 !
Received c4###command7 from sequencer !
Received c2###command3 from c2 !
Received c2###command3 from sequencer !
Sending c1###command2 to c2 !
sending c1###command4 from c1 to all!
Received c1###command4 from c1 !
Received c1###command4 from sequencer !
Received c3###command10 from c3 !
Received c3###command10 from sequencer !
Received c3###command13 from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###command13 from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###command2 from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###command2 from sequencer !
Terminal
Received c4###command6 from sequencer !
sending c4###command7 from c4 to all!
Received c4###command7 from c4 !
Received c4###command7 from sequencer !
Received c2###command3 from c2 !
Received c2###command3 from sequencer !
Received c1###command4 from c1 !
Received c1###command4 from sequencer !
Received c3###command10 from c3 !
Received c3###command10 from sequencer !
Sending c4###command9 to c1 !
Received c3###command13 from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###command13 from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###command2 from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###command2 from sequencer !
Sending c4###command11 to c2 !
Sending c4###command12 to c3 !
Terminal
Received c4###command7 from sequencer !
Received c4###command6 from c4 !
Received c4###command6 from sequencer !
Received c4###command7 from c4 !
Received c4###command7 from sequencer !
sending c2###command3 from c2 to all!
Received c2###command3 from c2 !
Received c2###command3 from sequencer !
Received c1###command4 from c1 !
Received c1###command4 from sequencer !
Received c3###command10 from c3 !
Received c3###command10 from sequencer !
Received c3###command13 from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###command13 from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###command2 from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###command2 from sequencer !
Sending c2###command8 to c1 !
Terminal
Received c1###command4 from c1 !
Received c1###command4 from sequencer !
sending c3###command10 from c3 to all!
Sending c3###command12 to c4 !
Received c3###command10 from c3 !
Received c3###command10 from sequencer !
sending c3###command13 from c3 to all!
sending c3###Hello, Welcome to Total_Ordered_multicast from c3 to all!
Received c3###command13 from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###command13 from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Sending c3###command15 to c4 !
sending c3###Hello, Welcome to Total_Ordered_multicast from c3 to all!
Sending c3###command15 to c4 !
sending c3###Hello, Welcome to Total_Ordered_multicast from c3 to all!
sending c3###command2 from c3 to all!
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from c3 !
Received c3###command2 from c3 !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###Hello, Welcome to Total_Ordered_multicast from sequencer !
Received c3###command2 from sequencer !


Open
test_result_c1_tom.txt
~/A-Nikhil/COL724/Assignment 2/Totally-Ordered-Multicasting/results
Save
c1###Hello, Welcome to Total_Ordered_multicast
c1###Hello, Welcome to Total_Ordered_multicast
c3###command7
c4###command5
c4###command7
c4###command6
c4###command7
c2###command3
c1###command4
c3###command10
c3###command13
c3###Hello, Welcome to Total_Ordered_multicast
c3###Hello, Welcome to Total_Ordered_multicast
c3###Hello, Welcome to Total_Ordered_multicast
c3###command2
c1###commnad6
c2###Hello, Welcome to Total_Ordered_multicast
c2###command13
c2###command4
c1###Hello, Welcome to Total_Ordered_multicast3
c2###command10
c4###command14
c1###Hello, Welcome to Total_Ordered_multicast
c1###command7
c4###Command2
c4###command2
Open
test_result_c2_tom.txt
~/A-Nikhil/COL724/Assignment 2/Totally-Ordered-Multicasting/results
Save
c1###Hello, Welcome to Total_Ordered_multicast
c1###Hello, Welcome to Total_Ordered_multicast
c3###command7
c4###command5
c4###command7
c4###command6
c4###command7
c2###command3
c1###command4
c3###command10
c3###command13
c3###Hello, Welcome to Total_Ordered_multicast
c3###Hello, Welcome to Total_Ordered_multicast
c3###Hello, Welcome to Total_Ordered_multicast
c3###command2
c1###commnad6
c2###Hello, Welcome to Total_Ordered_multicast
c2###command13
c2###command4
c1###Hello, Welcome to Total_Ordered_multicast3
c2###command10
c4###command14
c1###Hello, Welcome to Total_Ordered_multicast
c1###command7
c4###Command2
c4###command2
Open
test_result_c3_tom.txt
~/A-Nikhil/COL724/Assignment 2/Totally-Ordered-Multicasting/results
Save
c1###Hello, Welcome to Total_Ordered_multicast
c1###Hello, Welcome to Total_Ordered_multicast
c3###command7
c4###command5
c4###command7
c4###command6
c4###command7
c2###command3
c1###command4
c3###command10
c3###command13
c3###Hello, Welcome to Total_Ordered_multicast
c3###Hello, Welcome to Total_Ordered_multicast
c3###Hello, Welcome to Total_Ordered_multicast
c3###command2
c1###commnad6
c2###Hello, Welcome to Total_Ordered_multicast
c2###command13
c2###command4
c1###Hello, Welcome to Total_Ordered_multicast3
c2###command10
c4###command14
c1###Hello, Welcome to Total_Ordered_multicast
c1###command7
c4###Command2
c4###command2
Open
test_result_c4_tom.txt
~/A-Nikhil/COL724/Assignment 2/Totally-Ordered-Multicasting/results
Save
c1###Hello, Welcome to Total_Ordered_multicast
c1###Hello, Welcome to Total_Ordered_multicast
c3###command7
c4###command5
c4###command7
c4###command6
c4###command7
c2###command3
c1###command4
c3###command10
c3###command13
c3###Hello, Welcome to Total_Ordered_multicast
c3###Hello, Welcome to Total_Ordered_multicast
c3###Hello, Welcome to Total_Ordered_multicast
c3###command2
c1###commnad6
c2###Hello, Welcome to Total_Ordered_multicast
c2###command13
c2###command4
c1###Hello, Welcome to Total_Ordered_multicast3
c2###command10
c4###command14
c1###Hello, Welcome to Total_Ordered_multicast
c1###command7
c4###Command2
c4###command2

7. How to run the Code -

To run the code, you can use the provided Makefile with the "run_all" target. This will set up and run the forwarder, sequencer, and multiple clients. Here are the instructions for running the code using the provided script: Open a terminal window. Navigate to the directory where your project files are located. Ensure you have make and gnome-terminal installed on your system. Run the following command to execute the "run_all" target in the Makefile:

“make run_all”

This command will start the forwarder, sequencer, and multiple clients in separate terminal windows.

You should see terminal windows opening for the forwarder, sequencer, and each client. The forwarder and sequencer will start first, followed by the clients. Make sure you wait a few seconds after starting the forwarder and sequencer before the clients are launched.

The code will start running, and you will see the messages and results in the terminal windows.

Please note that you may need to adjust the terminal application (gnome-terminal) to your specific terminal emulator if you are not using GNOME Terminal.

Additionally, make sure you have the necessary dependencies and libraries installed to run the code, as specified in your project's documentation.

If you encounter any issues or need further assistance, please let me know.
Results will be generated in result folder. For both ordered and unordered multicast.

8. Code Implements Two-Phase Multicast:

Code implements two-phase multicast approach as follows:

Local Sequence Numbers (L): Each client in your system maintains its local sequence number (L). This number represents the order in which a client has processed messages.

Global Sequence Number (G): The sequencer, identified by the ID "sequencer" in your code, maintains the global sequence number (G). This number is initialized to 0 and is incremented each time the sequencer receives a message.

Message Sending: When a client wants to multicast a message, it includes its local sequence number (L) in the message. The client sends the message to the sequencer and other clients.

Sequencer's Role: The sequencer assigns a global sequence number (G) to each message it receives. It then multicasts the message, along with the global sequence number, to all clients.

Determining Order: Clients use the global sequence number (G) received from the sequencer and their local sequence number (L) to determine when to deliver messages to their application. A message is delivered to the application if $G = L + 1$, ensuring that messages are processed in the correct order.