Distributed Systems Assignment 4

Mohammad Haghir Ebrahimabadi

March 2019

1 Messaging layer

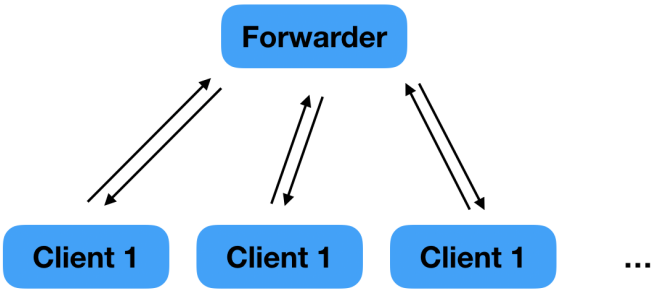
For implementing messaging layer zeromq package in python (pyzmq) has been used. Since both totally ordered and unordered messaging is desired, underlying infrastructure has been considered for both of them. In what follows , I first explain the structure of messaging layer for multicasting and the format of sending and receiving messages. Then I will describe the same topics for unordered messaging. In the next part I will explain how to have Totally ordered messaging. Then I will provide a test case and the results for my test case. There is also a section at the end that provides sufficient information for running the code.

1.1 Mulicast messaging

In this section first I explain the structure of my multicast messaging layer, then the sending and receiving functions will be described.

1.1.1 Messaging layer structure

Zeromq has different types of sockets like PUB, SUB, ROUTER, DEALER, REQ, REP, etc. which communication between not any pair of them, but some pairs of them is valid. Here for providing a message passing layer for totally ordered multicast I use the PUB/SUB pattern. PUB which is a publisher and binds to an address and SUB which is a subscriber and can connect to PUB's address. Since we have multiple clients which want to communicate with others, I used a zmq forwarder device or a proxy to take a message from a client or subscriber and multicast it to all of the clients. Thus, forwarder needs to have a SUB socket for receiving messages from a client, also needs to have a PUB socket to multicast it to clients (in totally ordered multicast when a client multicast, send the message to others, the client itself must receive the message). In below figure you can see how clients can communicate through the forwarder.



Each forwarder device can have a frontend and a backend address which here I bind the frontend to a SUB socket and backend to a PUB socket. Clients also have a PUB and a SUB socket. I connect the PUB socket of a client to SUB socket of the forwarder and SUB socket of the client to PUB socket of the forwarder.

One more socket has been considered in forwarder which has been used for synchronization. Suppose we have four banks that want to perform multicast messaging. For synchronizing them, one way is to make sure all of them are subscribed and then start messaging. You can see the forwarder code in below. In forwarder code, after clients subscription, there is a forever loop in zmq.proxy function which if everything works well prevents the code from going further (i.e. to except or finally part).

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

The client side code for this part, multicasting and synchronization has been shown in figures below respectively.

```
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")
```

```
# 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
```

1.1.2 Sending function

For sending a message zmq.send_multipart() function has been used. For having totally ordered multicasting, I used a special process as sequencer or leader which needs to send a global sequence number. Thus the sending function which its code can be seen in the figure below, has a seq_num input. A message can be sent from a client or sequencer to all. If a seq_num input is not specified, the sender is a client and an empty string will be sent instead of a sequence number. All of the inputs to send functions must be in binary format (or a byte-like object). I will explain the sequence number in the section 2, Totally ordered multicasting.

```
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])
    else:
        self.client_sender.send_multipart([self.identity.encode(), b'', msg])
```

1.1.3 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. In below you can see the code.

```
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
```

1.2 Unordered messaging

In this part same as multicast messaging, first the structure of messaging layer is described, then sending and receiving functions has been explained.

1.2.1 Messaging layer structure

For unordered messaging zmq ROUTER/DEALER pattern has been employed. The broker uses a ROUTER socket and creates a poll that can listen to multiple clients and watch for new events or arriving messages and forward them to specified destination which was included in the arriving message. In below you can see the ROUTER code.

```
def router_uno(self):
    self.context = zmq.Context()
    self.router = self.context.socket(zmq.ROUTER)
    self.router.bind("tcp://*:5562")

    # Initialize poll set
    self.poller = zmq.Poller()
    self.poller.register(self.router, zmq.POLLIN)

    # Switch messages between sockets
    while True:
        self.socks = dict(self.poller.poll())

        if self.socks.get(self.router) == zmq.POLLIN:
            sender_id, empty, receiver_id, message = self.router.recv_multipart()
            print("Received ", message.decode("ascii"), " from ", sender_id.decode("ascii"), "!")
            print( "Forwarding to ", receiver_id.decode("ascii"))
            self.router.send_multipart([receiver_id, b'',sender_id, message])
```

The client side for unordered messaging is a DEALER socket. In Below you can see the code for client.

```
# 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")
```

When you set identity for a DEALER socket and then send a message using zmq.send_multipart(), the first part of the message will be the socket identity.

For running the code it is required to specify an identity for each client which will be described more in next sections.

1.2.2 Sending function

AS I mentioned above, the ROUTER needs the receiver address to forward the message. Thus I use the zmq.send_multipart() function while the first field is automatically the sender identity as I mentioned above, I send an empty string as the second field, and the third and fourth fields will be receiver identity and message respectively.

```
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])
```

1.2.3 Receiving function

Receiving function receives the unordered message forwarded by the ROUTER.

```
def recv_uno(self):
    empty, sender_id, msg = self.client_deal.recv_multipart()
    print ("Received ", msg.decode("ascii"), " from ", sender_id.decode("ascii") , "!")
    return msg
```

2 Totally ordered messaging

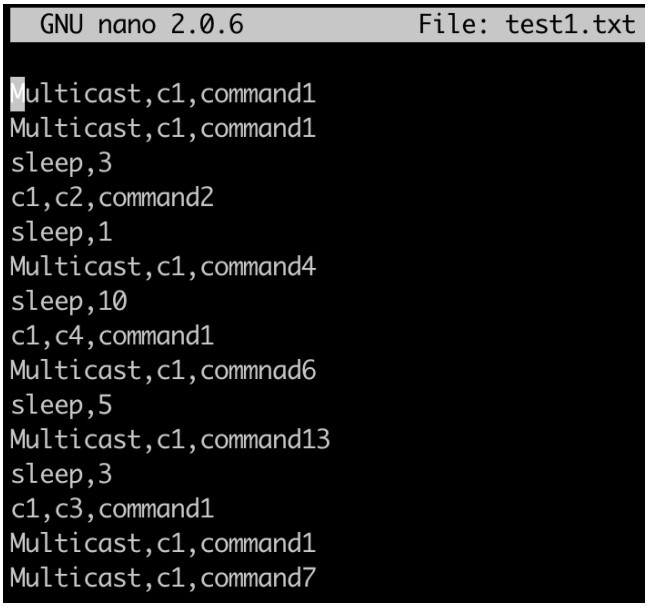
Once we have the messageing layer, we need an algorithm to have totally ordered messaging. I implemented totally ordered multicasting using a sequencer or leader. The sequencer keeps a global sequencer number, let call it G, which initially must be set to 0. When a client multicast a message to other clients and the sequencer as well, the clients receive the message and buffer it in their queue. The sequencer receives the message M and sets $G = G+1$ and multicast both G and message to the clients. Clients keep a local sequence number, let call it L. A client delivers the buffered message M to application once receives the message $< G, M >$ from sequencer and $G = L+1$. Once the message is delivered to the application, it sets $L = L + 1$. In below you can see sequencer and the code that handles both delivering totally ordered and unordered messages to application respectively.

```
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())
```

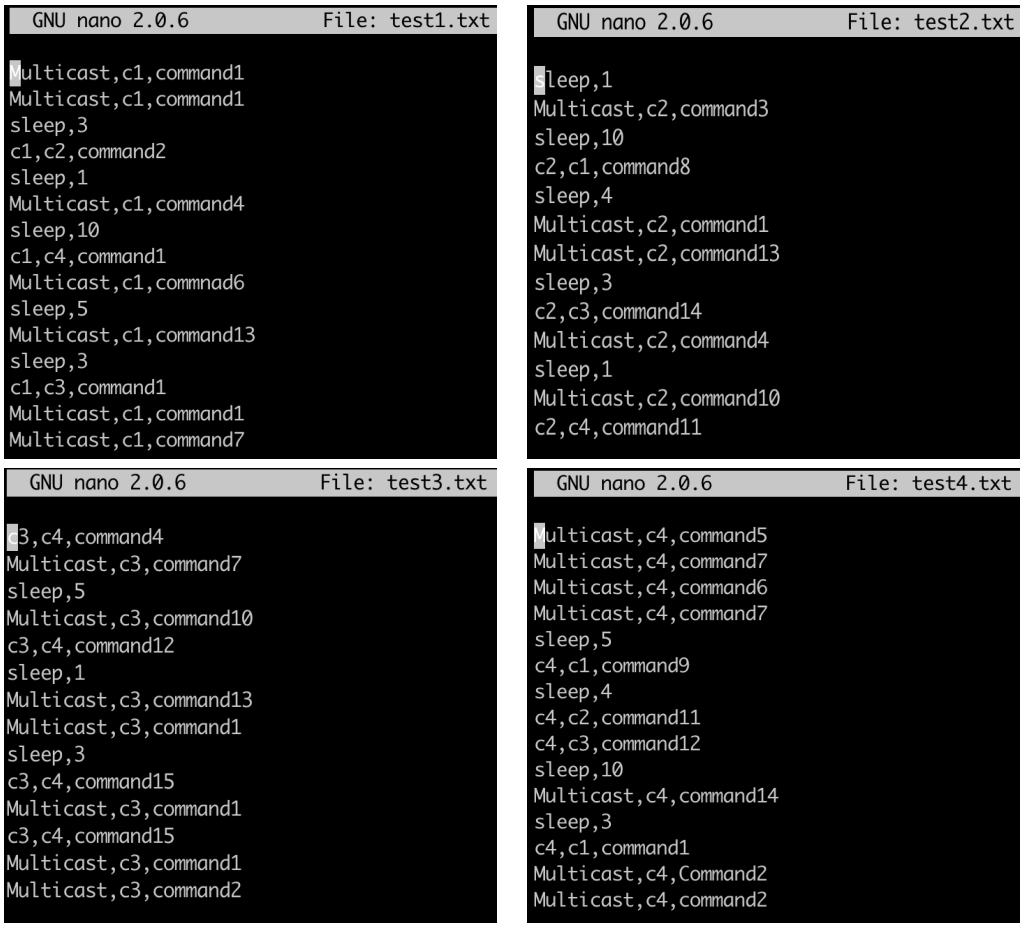
```
def tom_uno():
    cmd_list = []
    seq_list = []
    l_seq_num = 0    # local sequence number (initial value set to 0)

    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

            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 s
                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
                        del cmd_list[idx]                                           # message to the
                        del seq_list[0]
                        l_seq_num += 1
            time.sleep(0.01)           # For preventing from high cpu usage
```

3 Test and results

For testing, assume we have a number of clients that they want to send messages in totally ordered or unordered manner. I consider a .txt file for each client which inside it I have three different types of operations. In below you can see an example of contents of the file for one of my clients.

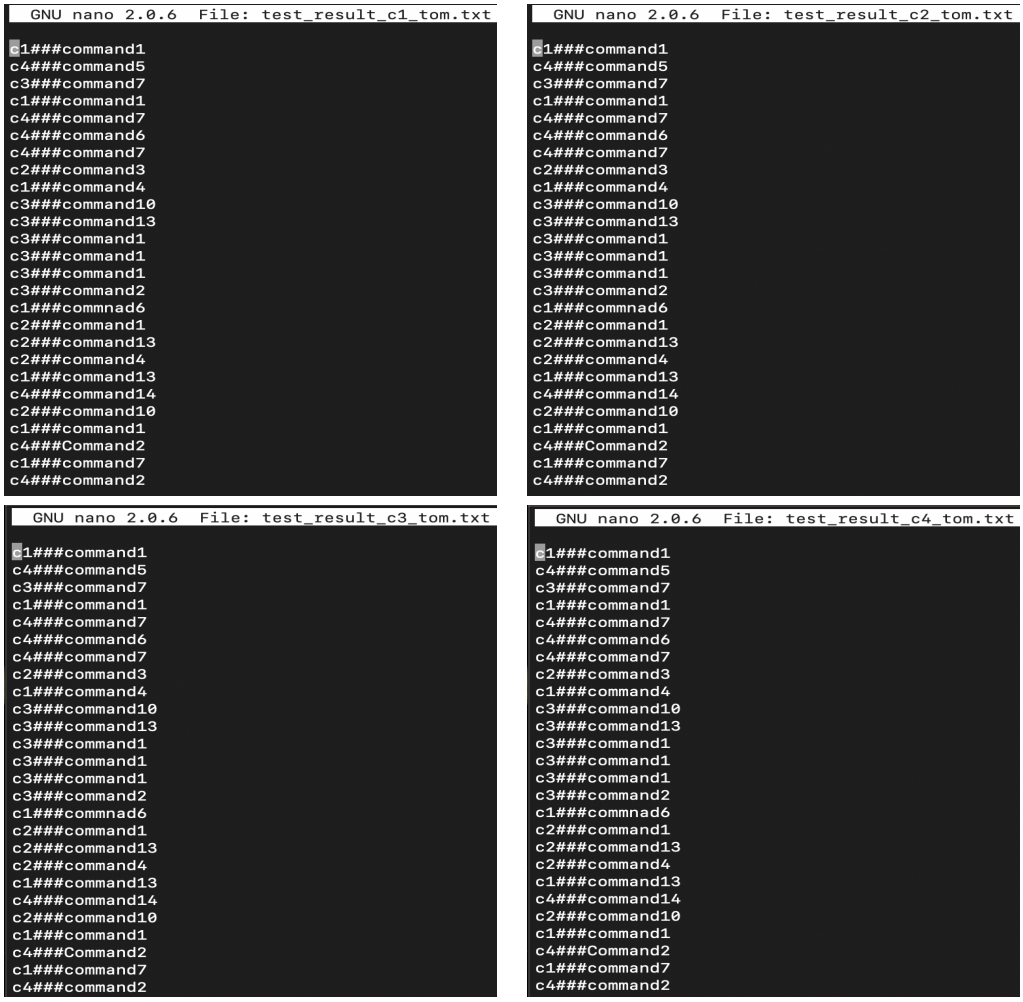


You can see three different types of lines. For example first line starts with "Multicast" which means the client (here "c1") must multicast "command1". For example third line starts with "sleep" which means client "c1" should not send anything for 3 seconds. As an example of the third type of operation, the fourth line starts with client identity which means "c1" must send "command2" to "c2". Each client has a file with the same format of lines. Clients must read their own set of operations from their file and perform them. Below you can see the contents of test file for each of my 4 clients. You can find these files inside test folder that I provided with my codes.

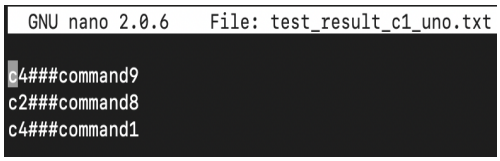


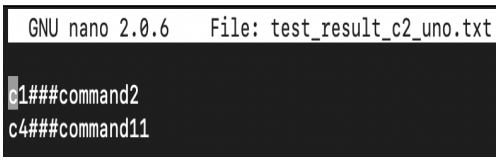
Each client writes the set of messages it receives from others, ordered or unordered, to two separate files. The input test file and output directory for saving results must be specified as will be explained in next section. Also, an identity for each client, and The number of expected clients for the forwarder device must be set which will be described in next section. Below you can see the order of received messages for each client. I added "Client identity + " to

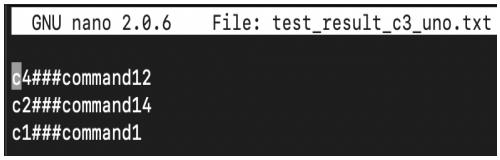
each command to see which client multicasted the message (it is not needed for code to work correct and can be removed inside code).

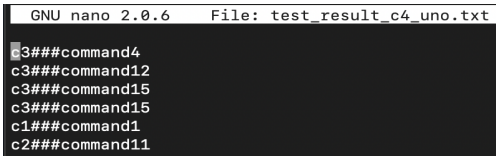


As mentioned above each client also have a file contains the messages it received unordered. Below you can see the received unordered messages for each client. You can see which client sent the message by looking at the beginning of the message.









You can find the files contains above information inside test folder that I provided with my codes. Below you can see part of "c4" client log.

```
sending  c4###command5  from  c4  to all!
sending  c4###command7  from  c4  to all!
sending  c4###command6  from  c4  to all!
sending  c4###command7  from  c4  to all!
Received  c4###command5  from  c4 !
Received  c4###command7  from  c4 !
Received  c4###command6  from  c4 !
Received  c4###command7  from  c4 !
Received  c3###command7  from  c3 !
Received  c3###command4  from  c3 !
Received  c4###command5  from  sequencer !
Received  c1###command1  from  c1 !
Received  c1###command1  from  c1 !
Received  c4###command7  from  sequencer !
Received  c4###command6  from  sequencer !
Received  c4###command7  from  sequencer !
Received  c3###command7  from  sequencer !
Received  c1###command1  from  sequencer !
Received  c1###command1  from  sequencer !
Received  c2###command3  from  c2 !
Received  c2###command3  from  sequencer !
Received  c1###command4  from  c1 !
Received  c1###command4  from  sequencer !
Sending  c4###command9  to  c1 !
Received  c3###command10  from  c3 !
Received  c3###command12  from  c3 !
Received  c3###command10  from  sequencer !
Received  c3###command13  from  c3 !
Received  c3###command1  from  c3 !
Received  c3###command13  from  sequencer !
Received  c3###command1  from  sequencer !
Sending  c4###command11  to  c2 !
Sending  c4###command12  to  c3 !
Received  c3###command1  from  c3 !
Received  c3###command1  from  c3 !
Received  c3###command15  from  c3 !
```

Below you can see router log.

4 How to run the code

The code was written in python 3.7.1 using pyzmq 18.0.0 package on macOS Mojave 10.14.3. Four files are required to run the code which are config.py, F_R.py, client.py, and main.py. The F_R.py which is the code for forwarder and router must be run first with the following format:

python F_R.py –nc <number of expected clients + 1>
where –nc is the number of expected clients plus the sequencer which in my test case is $4 + 1 = 5$.

For running sequencer the following line must be run:
python main.py –id sequencer

For running each of clients the following line must be run:
python main.py –id <client identity> –com /path/to/test/file -o /path/for/saving/results

The .sh executable file is included which uses the following format to run for example the F_R.py file:
osascript -e "tell application \"Terminal\" to do script \"python /Users/mohamadhaghir/Dis_HW4/F_R.py

–nc 5 \""

For running the .sh file in ubuntu "gnome-terminal -x" can be used.

5 Acknowledgement

zormq guide book at `http://zguide.zeromq.org/page:all`, also this video at
`https://www.coursera.org/lecture/cloud-computing/2-2-implementing-multicast-ordering-1-XmDt`
hepled me to implement this code.