**REPORT**

**Message Format:**

{

"data": "", #A random data is generated at the time of testing by tester.py

"conn\_type": "tom",#(tom/p2p)

"type": "app",#tells if message is coming from application layer

"timestamp": 0,#initially it is 0

"ack\_flag": 0,

"msg\_id": 0,

"ports": [4212, 4213],#known ports to each process

"p2p\_destination": 4212 #port for p2p to send message

}

**PseudoCode**:

1. All messages are timestamped with logical time.
2. On receiving any message from application, message will be multicast to all the processes including itself with its local timestamp sent.
3. On receiving message from a process as type “data”, message is put in a queue, queue is sorted. If a message is at head of the queue and has not sent acknowledgement, then acknowledgement is sent to all the processes.
4. On receiving acknowledgement, if the message at head has received acknowledgement from all the processes then it is processed and is delivered to application layer.
5. Once the head of queue message is processed, it is removed from queue and queue is sorted and if the process has not sent acknowledgement for this message, we multicast acknowledgement. And this continues until the queue is empty.

**Assumptions:** There are no failures of any processes. If there are any failures of threads or processes, code will not work as it should.

**Design Details:**

I have an application layer and a middleware layer.

Init\_Cluster():

When I run init\_cluster(). It spawns n number of processes passed as an argument.

Application\_Layer():

Once each process is spawned, it starts two threads, middleware on one thread and on other thread, it keeps listening for an output from the middleware.

Middleware –

Server() class:

It has a constructor which initializes a socket connection.

Anytime if I send a message from my tester function, it receives a message, creates a new thread to handle it and then looks for the type of message it is.

Peer-to-Peer (“p2p”):

If it is a peer to peer message, then a destination port is found from the message, and a socket connection is used to send the message.

After the message is received on the destination server, it displays the message by which process (port) it was received and sent.

Total Order Multicast (“tom”):

Once a message is received, there can be three types – “app” – message from application layer, “data” – multicasted message and “ack” – acknowledgement message.

Process\_app ():

If it is app, we first multicast this message to all processes including self, increment the timestamp and send message with this timestamp.

Process\_received\_data():

If it is multicasted message, we put this message in our queue and then sort this queue,

For the head of queue, multicast acknowledgement and mark that acknowledgement for the head of queue is sent in sent\_ack dictionary.

Process\_ack()

If it an acknowledgement, we check if it is in our ack dictionary which keeps the count of received acks. If it is in dictionary, we simply increment the count or else we add this new key in the dictionary.

A while loop keeps track if the head of queue has received all the acks and then pops it from queue and sorts again by timestamp, if not, multicasts ack for head to all the processes until the queue is empty.

**Notes:**

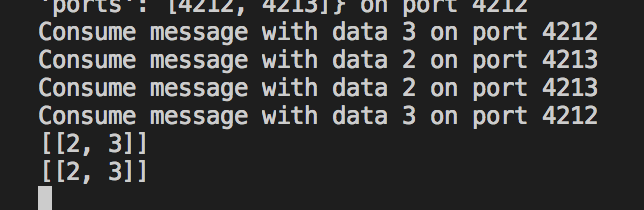
1. When sending any message, process will attach its own timestamp to it with the process id at decimal (As per total order multicasting).
2. While receiving any message, processes will take max of its own and incoming message timestamp and increment the timestamp
3. We maintain a dictionary to keep track of received acknowledgements for a message from application layer- ack={}
4. We maintain a dictionary to keep track of sent acknowledgement, used to check if the message at head has sent ack or not.
5. A queue is maintained which is sorted on any change, to handle the total order multicast for processing the messages in total order.
6. In every dictionary, we use msg\_id as the key for unique message tracking.

**Test Cases:**

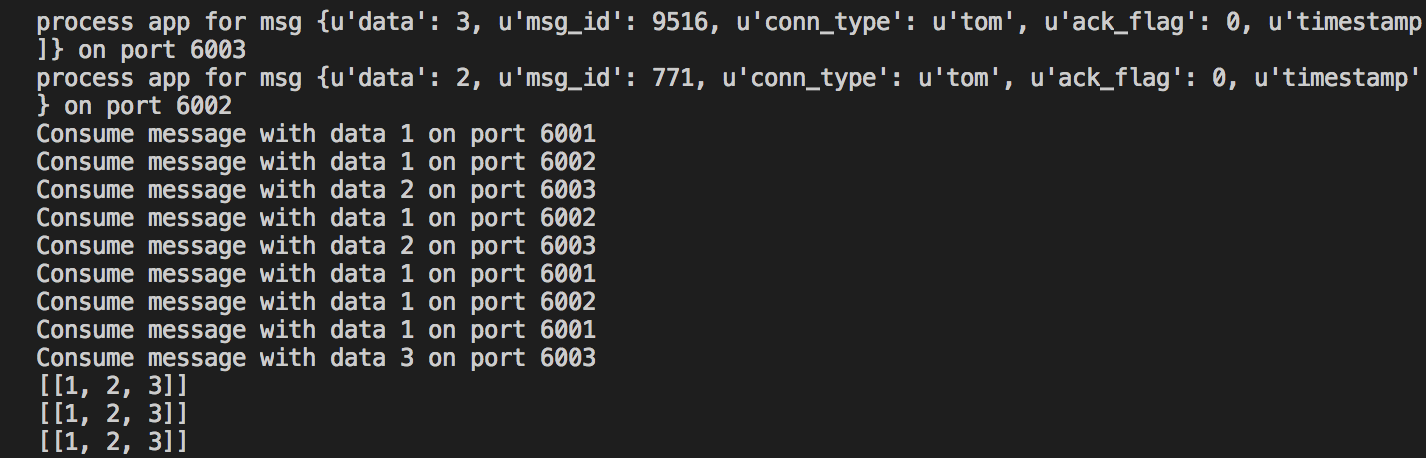
1. The test cases involved, sending just one message in total order multicast and peer to peer.
2. Sending two messages simultaneously to 2 processes, which maintain the order of the message in total order multicast and in peer to peer displays the message.
3. Sending 3 messages simultaneously to 3 processes, which maintain the order of the message in total order multicast and in peer to peer displays the message.
4. The final output is printed on the screen as list elements.

TOM testcase:

Below example shows the output for 2 process and 2 simultaneous messages on each process, output-

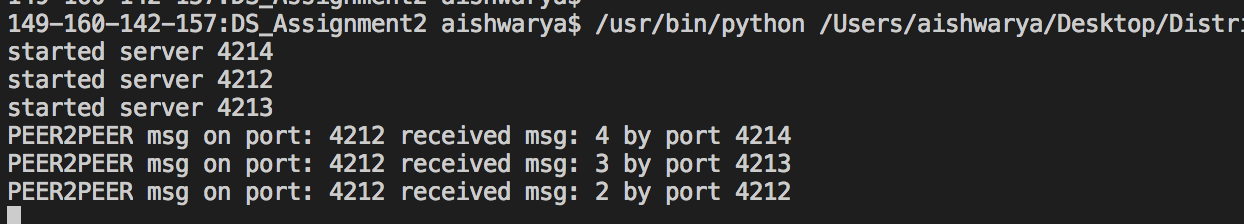


Below example shows the output for 3 process and 3 simultaneous messages on each process, output-

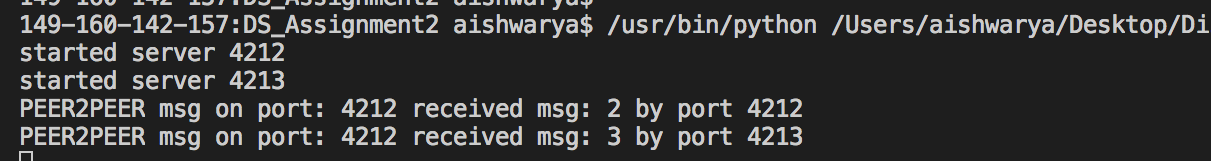


Peer to Peer Testcase:

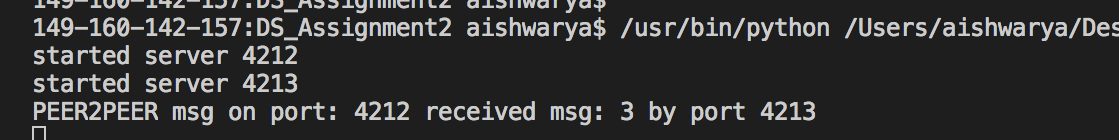
Below example shows 3 processes, trying to send message to a port at the same time-



Below example shows 2 processes, trying to send message to a port at the same time-



Below example shows 1 process, trying to send message to a port –



**Challenges Faced:**

1. Initially the sockets were not synchronized and since it has a blocking call, I could not send more messages to same process. To handle this issue, I have used threads.
2. Since, we are using threads to process the received request on each thread, concurrent access of the same class objects causes, overwriting of the messages. To handle this issue, I have used locks on threads, so that only one process will access the class objects at any given time (critical section).
3. For more than 2 processes, the program faces issues with threads such as broken pipe sometimes. Almost all the time, if the pipe is not broken, we get same sequence of messages in all the three processes.
4. Sometimes the sequence for 3 processes can change since threads are simultaneously trying to access the process.