B534 : Total Order Multicast

Siddharth Pathak

sidpath@iu.edu

March 20, 2019

Design:

The node is divided into two parts: user layer and network layer.When the node is started it spawns a new thread for the network layer. The main user process then waits for incoming messages from the network layer.

Network Layer (Middleware):

The Network Layer is a TCP server which is listening for new incoming messages. It exposes functions to the user layer to send peer to peer messages to the other node, and to send multicast messages to every other nodes in the group.
It also keeps a queue to keep track of the received messages, and to send a message to the user layer. It sends message to the user application via inbox. While the user layer is waiting on a signal from the network layer, if the network layer finds that there is a message ready to be processed by the user layer, it adds the message to the user layer's inbox, and then sets the signal. After setting the signal, user layer reads the message and processes it.
All the communication over the network is sent by the user layer to the network layer which then handles all the sending, clock incrementing, etc. The user layer doesn't care about the inside details. Network layer is an abstraction.

The network layer defines the following format of the message:

$$(MessageType, Clock, Payload)$$

Where Message Type can be `P2P`, `MC`, and `ACK` for peer to peer, multicast and acknowledgment respectively.

If the message is of type `ACK`, then the message payload is the message ID.

Depending on the message type, the network layer processes the message.

Processing of messages is discussed in the Algorithm section.

User Layer/Process:
The user layer is always waiting for incoming messages from the network layer. It waits using a Event variable. It calls wait on the event variable of the network layer. And since wait is a blocking

call, it waits indefinitely. Once network layer decides that there is a new message for the user layer, it signals the event and adds message to the inbox. As soon as the event is signaled, the user process reads the inbox, and processes the message from it. Since, the network layer is a different thread spawned by the user process, the data is shared.
For testing purposes, the tester function calls the test method of the user process which sends multicast and peer to peer messages.

Algorithm Implementation:

Receiving Messages:
Once the message is received by the middleware/network layer is processes the messages depending on its type.
If the message is of type `P2P` i.e. peer to peer then the network layer immediately attaches the message to the user inbox queue.
If the message is of type `MC` i.e. multicast message, then it does the following steps:

- Take the maximum of the incoming clock and current clock and increment by 1.
- It adds the message to the network layer queue. The queue is a priority queue which is always sorted with the message having the minimum lamport clock value at the top.
- Once the queue is sorted again after adding the message, the network layer checks if it has sent an acknowledgment for the message which is at the top. If it has not then it sends an `ACK` type of message.

If the message is of type `ACK` i.e acknowledgment message, then it increments the increment count for that message. It then checks if the message at the top of the priority queue has received all the acknowledgments. If it has, then it removes the message from the queue and sends it to the user process.
If the acknowledgment count is not equal to the number of nodes in the group, then its sends acknowledgment for the message to all the other nodes in the group including itself.
Middleware only sends acknowledgment if the message it at the top of its queue, and it has not sent acknowledgment for it before. And it only processes message if the message is at the top of queue, and it has received all the acknowledgments for that message. Middleware increments the clock while receiving and sending the acknowledgments.

Sending Messages:
For sending peer to peer message, the network layer increments the clock once before sending the message to the destination node.
For sending multicast message, the network layer increments the clock only once while issuing the message. Then sends message with that clock to all the nodes in the group.
For sending acknowledgment message, the network layer increments the clock once before sending the message to every node in the group.

Priority Queue:

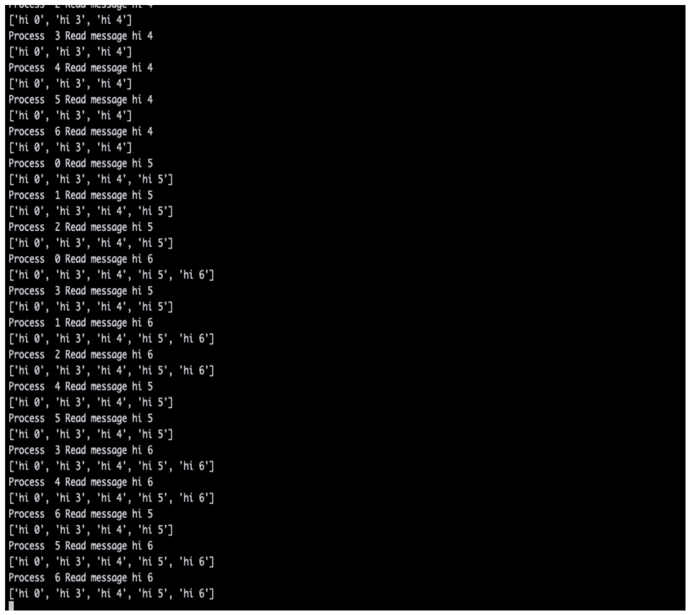
Priority queue is the crux of the algorithm. It stores the message with least lamport clock at the head. If there is a tie, it checks for the PID of the process which is sent along with the message. The queue stores the message in the following format:

$$(Clock, PID, MessageID, Payload)$$

The message ID is essentially the combination of clock and the PID which is unique for every message.

Testing:
I have provided a tester code. It starts each node and its middleware according to the config.json file. It starts each node as a new Python process using the multiprocessing library. Tester waits for few seconds to make sure each node and its middleware has started.

Test case 0:
For testing the total order multicast there should be some kind of parallelism for generation of messages. The user process makes each node wait before sending a message indefinitely. Thus, there are nodes which are ready to be send messages. Once all the nodes reach that state, the tester process sets the variable True which resumes all the thread which were waiting. Since any thread can resume first, there is some randomness and parallelism in sending of the messages. Each process prints out the messages it has received, from which we can confirm the order. Since there is lot of I/O involved, sometimes the threading breaks resulting in one of the nodes not sending the message. I have attached screenshots for successful runs:



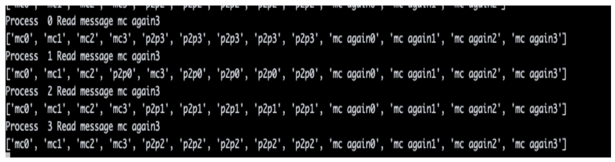
```
Process  2 Read message hi 4
['hi 0', 'hi 3', 'hi 4']
Process  3 Read message hi 4
['hi 0', 'hi 3', 'hi 4']
Process  4 Read message hi 4
['hi 0', 'hi 3', 'hi 4']
Process  5 Read message hi 4
['hi 0', 'hi 3', 'hi 4']
Process  6 Read message hi 4
['hi 0', 'hi 3', 'hi 4']
Process  0 Read message hi 5
['hi 0', 'hi 3', 'hi 4', 'hi 5']
Process  1 Read message hi 5
['hi 0', 'hi 3', 'hi 4', 'hi 5']
Process  2 Read message hi 5
['hi 0', 'hi 3', 'hi 4', 'hi 5']
Process  0 Read message hi 6
['hi 0', 'hi 3', 'hi 4', 'hi 5', 'hi 6']
Process  3 Read message hi 5
['hi 0', 'hi 3', 'hi 4', 'hi 5']
Process  1 Read message hi 6
['hi 0', 'hi 3', 'hi 4', 'hi 5', 'hi 6']
Process  2 Read message hi 6
['hi 0', 'hi 3', 'hi 4', 'hi 5', 'hi 6']
Process  4 Read message hi 5
['hi 0', 'hi 3', 'hi 4', 'hi 5']
Process  5 Read message hi 5
['hi 0', 'hi 3', 'hi 4', 'hi 5']
Process  3 Read message hi 6
['hi 0', 'hi 3', 'hi 4', 'hi 5', 'hi 6']
Process  4 Read message hi 6
['hi 0', 'hi 3', 'hi 4', 'hi 5', 'hi 6']
Process  6 Read message hi 5
['hi 0', 'hi 3', 'hi 4', 'hi 5']
Process  5 Read message hi 6
['hi 0', 'hi 3', 'hi 4', 'hi 5', 'hi 6']
Process  6 Read message hi 6
['hi 0', 'hi 3', 'hi 4', 'hi 5', 'hi 6']
```

Sample test case output

The array is the inbox of the process, which represents the order of the incoming messages.

Test case 1:
Send multicast message from each node, then send few p2p messages in a ring fashion and then send

multicast messages again. In this test case, most of the times the messages arrive in an ordered manner in which they were sent, since sending is done sequentially and there is no network delay.



```
Process  0 Read message mc again3
['mc0', 'mc1', 'mc2', 'mc3', 'p2p3', 'p2p3', 'p2p3', 'p2p3', 'p2p3', 'mc again0', 'mc again1', 'mc again2', 'mc again3']
Process  1 Read message mc again3
['mc0', 'mc1', 'mc2', 'p2p0', 'mc3', 'p2p0', 'p2p0', 'p2p0', 'p2p0', 'mc again0', 'mc again1', 'mc again2', 'mc again3']
Process  2 Read message mc again3
['mc0', 'mc1', 'mc2', 'mc3', 'p2p1', 'p2p1', 'p2p1', 'p2p1', 'p2p1', 'mc again0', 'mc again1', 'mc again2', 'mc again3']
Process  3 Read message mc again3
['mc0', 'mc1', 'mc2', 'mc3', 'p2p2', 'p2p2', 'p2p2', 'p2p2', 'p2p2', 'mc again0', 'mc again1', 'mc again2', 'mc again3']
```

Sample test case output

The array is the inbox of the process, which represents the order of the incoming messages.

Test case 2:

Send few p2p messages in a ring fashion and then send multicast messages twice from each. In this test case, most of the times the messages arrive in an ordered manner in which they were sent, since sending is done sequentially and there is no network delay.

Config File:

Config file is a JSON file, which contains the IP and port for each node in the group.