Col724 ACN Assignment 2: Totally Ordered Multicast

1. Summary

In some scenarios an alternative to distributed ledger technology (blockchain) is if we can apply a totally ordered multicast algorithm. This assignment asks you to implement it.

This is a fairly open-ended assignment, but you must carefully document all your assumptions.

2. Details

You need to implement the totally ordered multicast "on top" of a lower-layer messaging interface which implements a standard multicast algorithm, You can assume that your layer intercepts all incoming and outgoing network traffic, but must be able to deal with both ordered-multicasts and regular unordered point-to-point messages.

3. Hints and Design Decisions

- Think of this as a protocol on top of the transport layer
- What is the format/layout of the messages?
- You can assume all network sends are either `send_unordered(msg, destination)` or `send_tom(msg, recipients)`

How should you handle receiving both multicast and point-to-point messages

4. Multicast & Ordering (brief Background)

Also see https://www.cs.princeton.edu/courses/archive/fall22/cos418/docs/L6-vc.pdf

A *Multicast* message is a one-to-many message. It is much like a broadcast, but it directed to a much smaller collection of hosts. These hosts may be on the same network, or they may be on different networks.

Here we are going to discuss implementing multicast as an application-level protocol above a reliable unicast. Specifically, we are going to ponder the order in which the messages arrive, and how we can take control and ensure that the application receives messages in the right order -- for various definitions of "the right order."

Traditional IP layer multicast (PIM) considers an efficient network-layer implementation using router trees to avoid duplication messages whereas we are

discussing an application-layer implementation and are more concerned with ordering guarantees than the duplication of messages.

Ordering Guarantees

When we are multicasting from one host to many hosts, the message may arrive at each of the hosts at a different time. As a result, if a single host dispatches several multicast messages, they may get "crossed in the mail". The situation can become further tangled if several hosts are multicasting.

Depending on the nature of the interaction of the hosts of the distributed system, we may or may not be concerned with the ordering of the messages. For example, if we know that every message will be completely independent of every other message, a simple *reliable multicast* will do -- we don't need to do anything special to ensure that the messages arrive in any particular order.

But what if our system isn't quite so relaxed. It may be the case that each host expects its own messages to be received in the order in which they were sent, but that it doesn't matter how they are interleaved with messages from other hosts. This is known as FIFO ordering.

A stricter ordering requirement is to ensure that all causally related messages, independent of the host, are received in the order in which they were sent. Here we are discussing the prevention of causality violations by enqueing messages and delivering them to the application in the proper order.

The strictest ordering requirement is *total ordering*. Total ordering requires that the messages be delivered in the same order as if they would be if the communication was instantaneous. In other words, the messages should be received in the same order they would be if messages were received at exactly the same time that they were sent. A *reliable*, *total ordering* multicast is known as an *atomic multicast*. By assuming that the unicast is reliable, we will be constructing an atomic multicast.

FIFO Multicast Protocol

We can ensure FIFO ordering in our multicast protocol by using a per source sequence number. Each host maintains a counter and of messages sent and sends this count, a sequence number, with each multicast message.

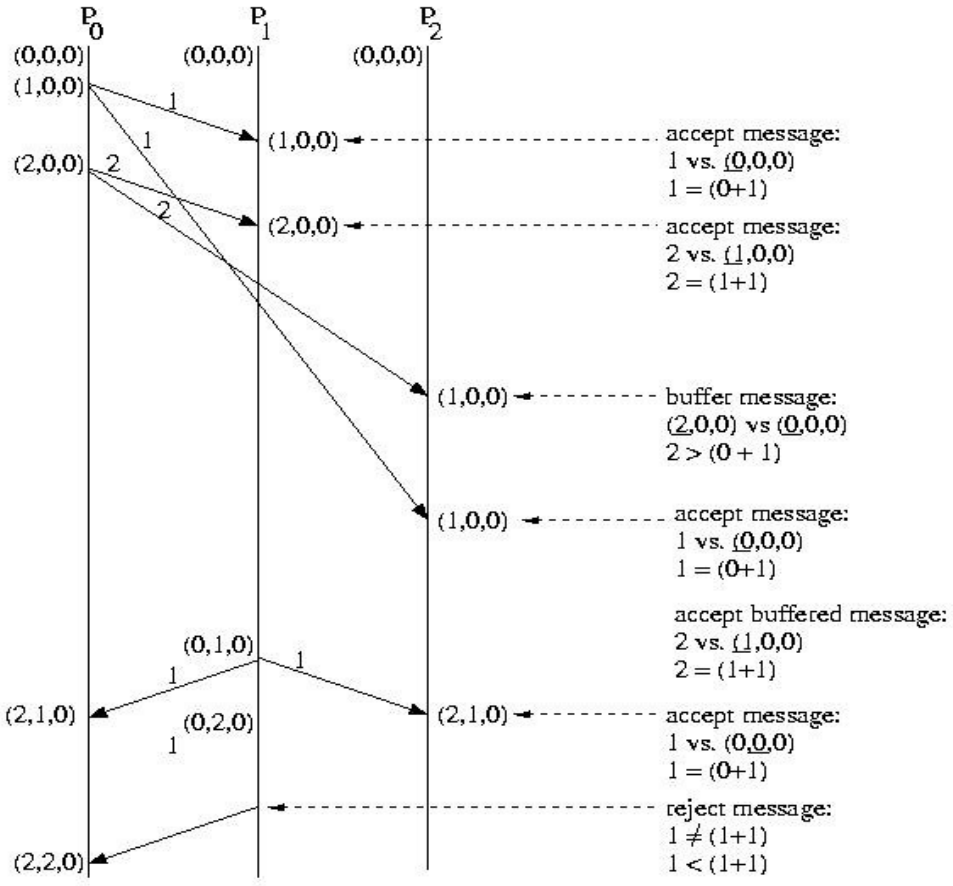
Each potential receiver maintains a queue for each potential sender (or at least the ability to create such a queue). Each potential receiver also maintains the "expected sequence number" associated with each possible sender. Since the host should receive all multicasts, this number should be incremented by exactly one with each multicast message from a particular host.

When a multicast message is received, the sequence number is compared to the expected sequence number. If the sequence number is as expected, the message is passed up to the application and the "expected sequence number" associated with the sender on the receiver is incremented.

If the sequence number of the message is lower than the expected sequence number, the message is thrown away -- it is a duplicate of a message that has already been received.

If the sequence number of the message is higher than the expected sequence number, the message is queued -- it is not yet passed up to the application. The reason is that one or more earlier messages from the same sender have yet to arrive.

Once the expected message has arrived, the queue is check. This queue is probably maintained as a priority queue sorted by sequence number. Messages are dequeued and passed up to the application until the queue is empty, or the next message in the queue is not the "expected message". Below is an example of this protocol at work:



Casual Ordering Multicast Protocol

We ensure that messages are delivered without causality violations -- by buffering messages that arrive too early. We determine if a message has arrived too early using a vector timestamp e.g.: lamport vector clock (https://lamport.azurewebsites.net/pubs/time-clocks.pdf).

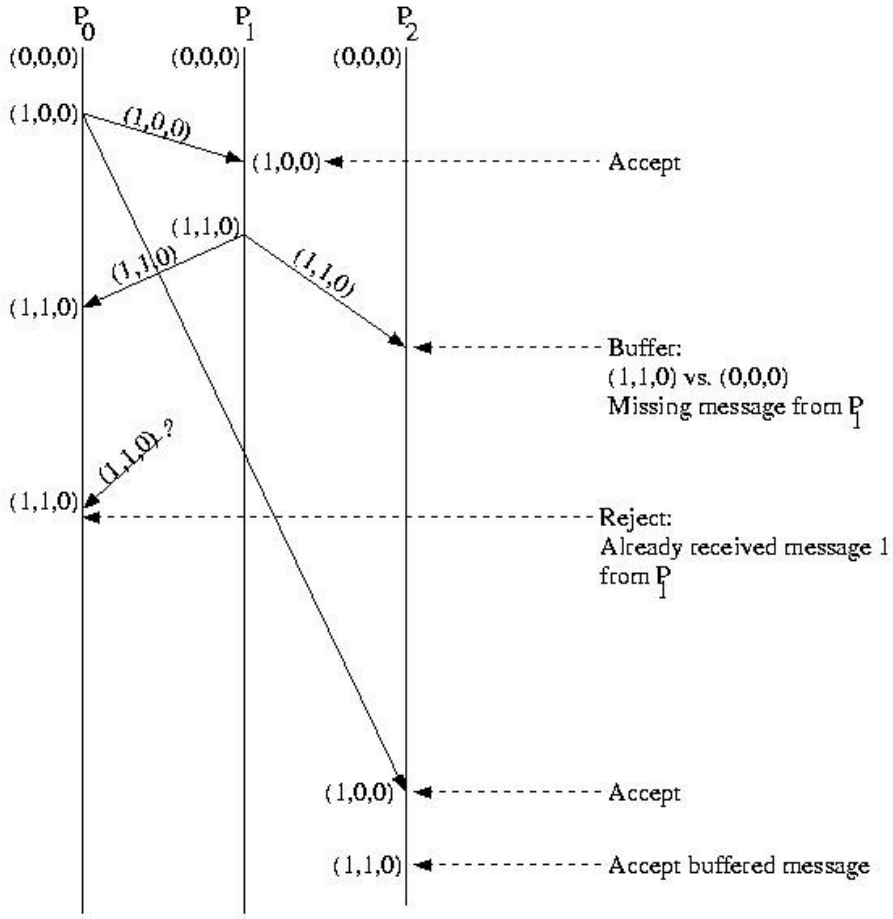
The key observation is that with a multicast protocol, all hosts within the group should (eventually) see the same messages. As a consequence, each host should see the same number of messages from each other host.

So, our vector contains one entry for each host. This entry counts the total number of messages *received* from the corresponding host. The entry for a host that corresponds to itself is used to count the messages it has sent.

Each host sends a copy of its vector with each message and compare the sender's vector with its own on receive:

- If any entry in the sender's vector, that was sent as a "timestamp" with the multicast message is greater than the corresponding entry in the receiver's local copy for the vector, the receiver buffers the message. This is because the sender has received a message, which is potentially causally related to the message it subsequently sent, that the receiver has not yet received. If the incoming message were passed up to the application, a causality violation might result.
- If the sender's entry in the message's timestamp is more than one greater than the sender's entry in the local time vector, the message is also buffered. This ensures that the protocol ensures FIFO ordering.
- If the sender's entry in the messages timestamp is less than the sender's entry in the local timestamp, the message is rejected -- it is a duplicate.
- If none of the above are true, the message is accepted. Accepting a message offers the opportunity to dequeue previously enqueued messages, if they can now be accepted.

Below is an example of the causal ordered multicast protocol:



Total Order Multicast

Total ordering requires that all messages are seen by all hosts in the same order. This could be easily achieved if we had a global clock or counter that could place serial numbers on messages. Then multicasts would just be accepted in order of serial number, and buffering could be used to handle missing messages. Some systems emulate this approach using a *central sequence number server*.

For now, we'll consider a distributed approach that can function in light of differing local times (serial numbers), called the *two-phase multicast*. In this approach, local times are used. The local time is incremented any time an operation is performed. Any time a system discovers that another system has a greater time, it resets its own time to the greater time.

Here's how it works:

- The local time is incremented. The message is sent containing the local time.

- The receiver buffers the message. It then sets its local time to the time of the sender, if the sender's time is higher. Increments its local time, and sends an ACK that contains the local time.
- The receiver waits until it has received all of the replies. It then determines the highest local time among the ACKS, and resets its clock, if necessary. It increments its clock and sends a message to all of the original recipients containing the "commit time." This is the time at which the recipient considers the message to have been received.
- Applications on the recipient host can see the message only after all messages received between the "acknowledgements" and "commitment" of the message have been committed. This ensures that they won't receive earlier commitment times.

4 What to Submit

You must submit your code, examples, test-cases, and a report.

Think carefully of how you can test the code for correctness. These are a major evaluation component. The test cases should ideally consist of multiple processes with random delays in sending/responding to messages, so that the correctness of the implementation is tested. At least one testcase should show that the total order of messages is different from real-time order.

The report should have describe the pseudo-code of the algorithm implementation, design details, and some experimental evaluation.

Component	Weight
Working code	50%
Test-cases and examples	30%
Report	20%