**Homework 5**

**Causally Ordered Multicast Communication**

For this homework you will modify the Chat application to enforce causally ordered multicast communication. This protocol is to be used only for multicast messages, so it is appropriate for only some of the communication in the Chat application. Use it only in the case of the chat server sending a message to all clients.

Each process (whether client or server) maintains its own clock. The clock

* knows the id of the process it belongs to, and has a reference to the process (a local object reference)
* has a time vector
* has a message queue

The message queue holds any messages that haven’t been delivered (because they arrived out of order, namely too early). Each message has an associated time vector.

The time vector is simply an array of integers representing the “time” for each process in the system. So the current time of process i is stored in the ith element of the time vector. Since each process has a clock and the clock holds a time vector, each process is keeping track of the “time” not only for itself, but for every other process in the system.

When a process sends a message, it increments its own clock. When a process receives a message, it updates the clock values of the other processes. (Remember that the process that sent the message attached its own time vector to the message.) But message receipt must obey causal ordering, so the process holds back any messages that arrive too early. More specifically, the following should happen when messages are sent/received.

Before sending a multicast message, a process should which

* Invoke clock.updateTime\_Send(), which increments the clock (one clock tick).
* Attach the clock’s time vector to the message.

Upon receiving a multicast message, a process should:

* Check whether the message is deliverable by calling clock.deliverable().
* If the message is deliverable then:
  + deliver the message:
    - Invoke clock.updateTime\_Receive() to update the clock’s time vector using the time vector of the arriving message.
    - Process the message.
    - Try to deliver any queued messages that may now be eligible for delivery. Do this using:

while (clock.tryToDeliverOldMessages());

* If the message is not deliverable, then
  + Add the message to the clock’s message queue by calling clock.enqueueMessage().

The process must implement interface ClockApp, so it must implement the following method:

public void processMessage (Message m);

The clock calls this method when it finds deliverable messages on the queue. Thus a process that only sends multicast messages can make this method empty. A process that receives multicast messages must process the message by checking its message type, then applying the appropriate processing for that type of message. Whether a message is queued or is deliverable when it arrives (so it never goes onto the queue), it should be processed in the same way.

To create the clock, a process invokes:

clock = new CausallyOrderingClock(this, myId);

Details of the clock:

* A clock is created by a process i, and it keeps an object reference for this process and remembers its process id.
* A clock holds a timeVector, which contains one entry for each process in the system. The current time of process i is located at timeVector[i].
* A clock hold a queue of messages that arrived at process i but are not yet deliverable to process i.
* For a message from process i to be deliverable to process j, its time vector must meet these two conditions:
  + msg.timeVector[i] == clock.timeVector[i] + 1
  + for all k != i, msg.timeVector[k] <= clock.timeVector[k]
* Updating the clock’s time vector using the time vector of the arriving message:
  + for each k, clock.timeVector[k] = max(clock.timeVector[k], msg.timeVector[k])
* Each time the clock ticks,
  + check to see whether any queued messages are deliverable to process i; if so, deliver them.

See pp. 250-251 for the algorithm details. I have given you the code for the clock class (CausallyOrderingClock.java).

Note that if you temporarily disconnect a client (and reconnect it), it will miss any messages that were sent while it was disconnected. Then after it is reconnected, the next message it receives will look out of order, since the clock time of the sender has skipped ahead. Thus the client will simply queue all messages from that point on, since the messages that were sent while it was disconnected will never arrive. Similarly, if you connect a client after some messages have been sent, the new client will queue all messages (waiting for the messages it missed).