SE424: Distributed Systems	**Recitation 12**
Semester 1 5785	**26 Jan 2025**
Lecturer: Michael J. May	**Kinneret College**

Causally Ordered Multicast P2P Tool

1 Review of Vector Clocks

In class we discussed the concept of a vector clock - a logical time mechanism which expands upon the Lamport logical clock. Vector clocks (invented by Mattern, so they are sometimes called Mattern vector clocks) include a logical time stamp for each process in the group. Messages are then sent out along with a vector of logical time stamps. For example, in a communication group with four nodes A, B, C, D, we might see the following message and time stamp sent from node A:

$$m_1 = \text{“Hello”}, m_1.ts = [A = 10, B = 11, C = 12, D = 16]$$

The implication is that the message m_1 was sent from A as its 10th event. The message was sent after A had seen messages influenced by 11 events at B, 12 events at C, and 16 events at D. We can use the vector clocks to determine causality between messages and events. For instance, if we see another message m_2 sent from C with the following content and time stamp:

$$m_2 = \text{“Bye!”}, m_2.ts = [A = 11, B = 12, C = 14, D = 16]$$

We may be certain that the message m_1 causally preceded message m_2 since the time stamps of m_1 are strictly less than or equal to the values in the time stamp of m_2:

	m_1		m_2
A	10	$\leq$	11
B	11	$\leq$	12
C	12	$\leq$	14
D	16	$\leq$	16

Based on the analysis of the message time stamps we can conclude that $m_1 < m_2$.

2 Review of Causally Ordered Multicast

Causally Ordered Multicast (COM) is an application which uses vector clocks to achieve a goal: ensure that nodes in a group receive group communication messages in causal order.

Achieving that goal requires us to adapt the vector clock sending and receiving algorithm a bit. The algorithm for COM is as follows. For the following steps, let N be the set of communicating processes, let the local vector clock for node p be denoted c, and let the entry for p in c be $c[p]$:

Sending When sending a message, the sending node increments $c[p]$ in the vector clock. The message is then sent with the current time stamp of c attached to it.

Receiving When a node receives a message, it performs the following steps. Let the message be called m and then time stamp attached to the message be $m.c$:

(1) The recipient checks if $\forall q \in N - i, m.c[q] \leq c[q]$ for some i and $m.c[i] == c[i] + 1$.

(2) If the check in step (1) returns true, m is accepted and c is updated such that $c[i] = m.c[i]$. After that, check the pending message queue to see if any other message are acceptable based on the new value for c.

(3) If the check in step (1) returns false, put m in the pending message queue.

3 COM Tool

The application you will develop is similar to the tool we developed for totally ordered multicast (TOM) last time. The *Causally Ordered Multicast (COM)* system described in class is based upon the following rules:

1. Each process in the system maintains a single logical clock (vector) which represents its logical time.
2. When a process wants to send a message to the other processes in the system, it attaches its current logical time stamp and process identifier to the message and forwards it to all of the other processes in the system.
3. When a process receives a message from another process, it checks if the message is showable immediately (as in the Receiving rule above). If yes, it is shown immediately and the local logical clock is updated accordingly. Otherwise, it is put in the pending queue until the message is showable.

3.1 Pending Messages and Output

When a message arrives and it's not immediately showable, it is put in the *pending queue* until any missing messages which causally preceded it have arrived. The tool therefore needs to check each incoming message in the following manner:

- If the message is showable immediately, process it as above. Then, check if any messages in the pending queue can be deleted from the queue and output to the output file due to the updated clock. Releasing one message from the queue might lead to multiple ones being freed, so we need to iterate over the pending queue until no more messages can be removed.

3.2 Sending Messages and Delay

To make the tool more interesting, we will have the sending function of the tool wait between 0 and 10 seconds between each successful send. This will give us enough time to send messages in a non-FIFO manner and thereby exercise the pending queue.

4 Tool Interface

The tool will have a command line interface in Java. The tool will accept three parameters:

neighborsFilePath The path to the neighbors file (parsing code provided)

outputFilePath The path to write out the final procesed messages to

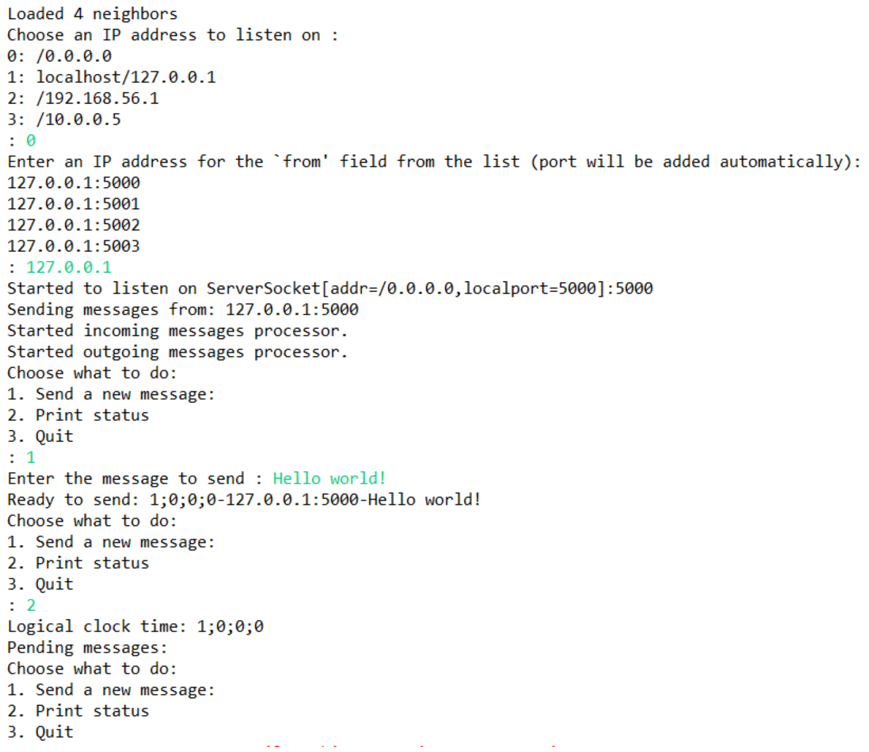
port The port to listen on

Once started up, the tool will ask for two additional values:

IP Address The IP address to listen on

From name What name to write in the "from" field in the messages (see below). If the user chose an IP address which is not 0.0.0.0, it will just use the IP address in text format. Otherwise, it will ask for an IP to write. The IP address must be in the neighbors list; otherwise the tool will not work.

A screen shot of the opening of the app is shown in Figure 1:



```
Loaded 4 neighbors
Choose an IP address to listen on :
0: /0.0.0.0
1: localhost/127.0.0.1
2: /192.168.56.1
3: /10.0.0.5
: 0
Enter an IP address for the `from' field from the list (port will be added automatically):
127.0.0.1:5000
127.0.0.1:5001
127.0.0.1:5002
127.0.0.1:5003
: 127.0.0.1
Started to listen on ServerSocket[addr=/0.0.0.0,localport=5000]:5000
Sending messages from: 127.0.0.1:5000
Started incoming messages processor.
Started outgoing messages processor.
Choose what to do:
1. Send a new message:
2. Print status
3. Quit
: 1
Enter the message to send : Hello world!
Ready to send: 1;0;0;0-127.0.0.1:5000-Hello world!
Choose what to do:
1. Send a new message:
2. Print status
3. Quit
: 2
Logical clock time: 1;0;0;0
Pending messages:
Choose what to do:
1. Send a new message:
2. Print status
3. Quit
```

Figure 1: Tool Startup

4.1 Neighbors and Vector Clock Format

On startup, the tool automatically parses the neighbors file provided and loads the neighbors into a `Vector`. The neighbors file is assumed to have the following format:

```
127.0.0.1:5000
127.0.0.1:5001
127.0.0.1:5002
127.0.0.1:5003
```

Each line contains an IP address and port divided by a colon (:). There shouldn't be any duplicates in the file, but the tool can ignore any such errors.

The vector clock must be the same size as the neighbors vector and be ordered in the same order. For example, in the above neighbors list, we might have a vector of the form:

$$[1, 4, 6, 3]$$

That vector implies that the message was sent after:

- 1 message from 127.0.0.1:5000
- 4 messages from 127.0.0.1:5001
- 6 messages from 127.0.0.1:5002
- 3 messages from 127.0.0.1:5003

For this scheme to work, all nodes must have the same ordering in the neighbors file or perform some consistent sorting on the neighbors names.

5 Protocol Details

Like the TOM tool shown before, the tool communicates among a group of statically defined neighbors. There is one kind of message sent between the nodes:

40;10;30-127.0.0.1:5000-Text Here! The message uses - as a field delimiter. It has the following fields:

1. The logical vector clock of the sender at the time the message was sent delimited by semi-colons (;). The order of the vector clock must be identical to the order in the neighbors list (either as listed or after some consistent sorting).
2. The IP address and port of the sender
3. The text of the message sent.

Strictly speaking, the IP address and port of the sender can be skipped and the system will still work. Removing the field will just make it harder to identify who the sender is if that's important.

6 Threading and Processing

Since we are not writing a GUI, we will need to separate out various pieces of logic into different threads and use a producer/consumer model. We can identify four threads which need to run in parallel: (a) The UI thread, (b) the message listening thread, (c) the message sending (outgoing) thread, and (d) the message processing (incoming) thread.

All threads communicate using a `BlockingQueue` interface. I used `LinkedBlockingQueue` as the concrete class.

The relationships between the threads is shown in Figure 2.

6.1 UI Thread

The UI thread needs to offer a menu based interface to the user to allow him to:

- Send a new message
- Print out the status of the pending queue and the current logical clock. To print out the pending queue, the UI thread also needs read access to the pending queue.
- Quit - close all of the threads and the listener

6.2 Message Listening Thread

The message listening thread needs to listen on the `ServerSocket` and get incoming messages. Once an incoming message arrives, it adds it to the *incoming messages queue* and goes back to listen.

6.3 Message Sending (Outgoing) Thread

When the tool needs to send a message, it is first added to the *outgoing messages queue.* The outgoing thread consumes from the outgoing message queue and then sends the message to all neighbors in the neighbors list.

As noted above, to make the pending queue be useful, we'll add a `Random` class here to wait between 0 and 10 seconds between each successful send (use `Thread.sleep`).

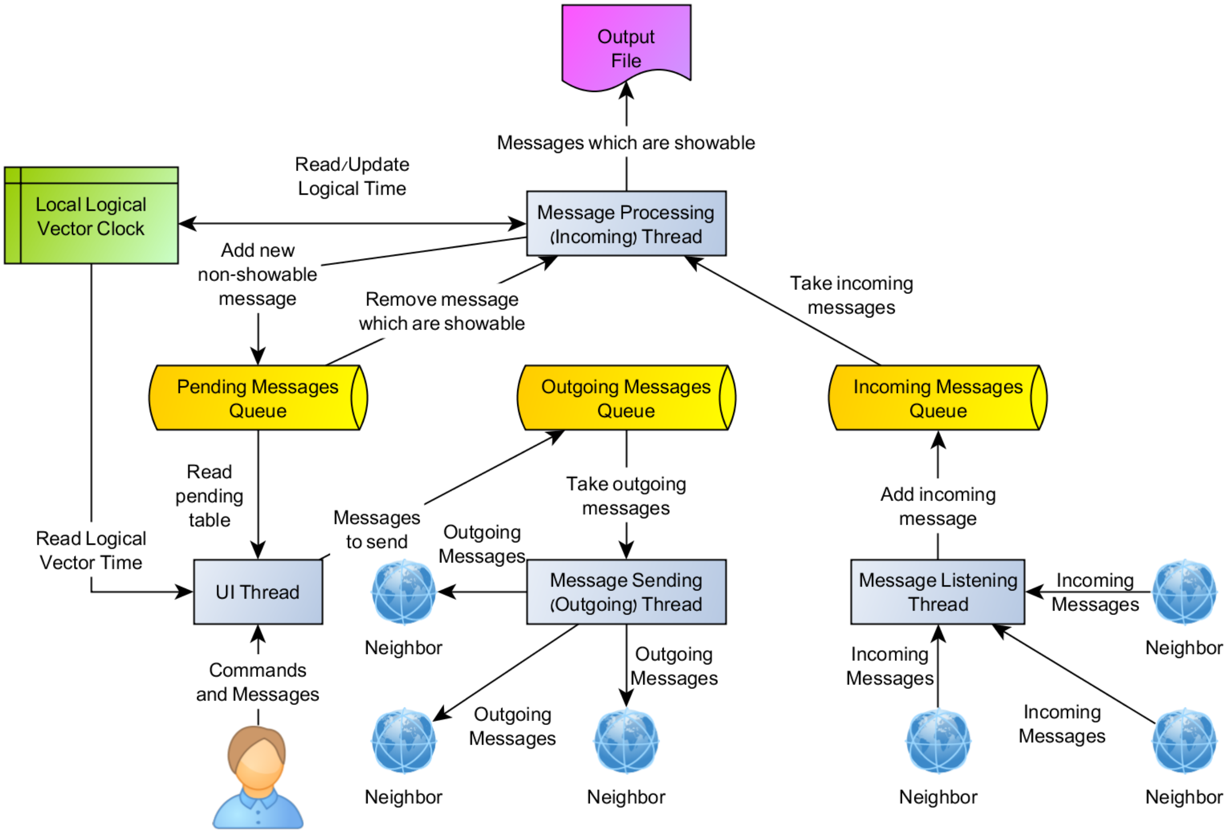


Figure 2: Queues schematic

6.4 Message Processing (Incoming) Thread

When a message arrives, the incoming messages thread takes it off of the incoming messages queue and processes it.

- If it's not in a format that we recognize or the logical vector time stamp on it has the wrong length, we just ignore it.
- If it's formatted OK and showable, we immediately output it to the output file.
- If it's formatted OK and not yet showable, we put it in the pending queue as above in Section 3.1.

7 Design Considerations

There were a couple of design considerations that went into the tool as written. There are alternate solutions to the problems given, but this presents a fuller picture of the tool as written:

Static Groups The groups are defined ahead of time, so each node must be a neighbor of every other node. That means that the graph must be a clique and each node must be aware of all other nodes. If a neighbor is listed in the file and is not active, things will still work, but we'll waste space in the vector clock time stamp.

Asynchronous Listening and Sending To make the tool work better, separate listening and sending threads are used. They communicate using a producer/consumer model.

Delay Since our experiments will be performed in a small local area network, message delivery is almost immediate. That makes it difficult to see the use of the "Pending Messages" queue. To make it clearer, we add an artificial "delay" to the tool of up to 10 seconds.

8 What to do

Your task today is to complete the Causally Ordered Multicast tool. I have given you much of the running code to get you started. You must add the following functionality:

1. Message sending logic - `ComMain.doSendMessage()`
2. Print status logic - `ComMain.doPrintStatus()`
3. Incoming message processing - `IncomingProcessor.run()`
4. Deciding if a message is showable now (if its elements are all less than or equal to the elements in the local logical clock except for one which is greater by 1) - `SharedState.messageIsNext(Message msg)`
5. Listening for incoming messages (very similar to the logic for TOM) - `Listening.run()`
6. Outgoing message processing (very similar to the logic for TOM) - `OutgoingProcessor.run()`

I recommend starting from the "empty" COM code so you don't start from scratch.