SE424: Distributed Systems	**Recitation 11**
Semester 1 5785	**19 January 2025**
Lecturer: Michael J. May	**Kinneret College**

Totally Ordered Multicast Tool

In class we saw one use of the Lamport logical clock - a totally ordered multicast system that lets processes to communicate asynchronously but still agree upon a single, global ordering of all events. In recitation today we will develop a tool that implements a protocol that supports the totally ordered multicast system described in class. We'll call the tool *TOM*.

1 System Description

The totally ordered multicast system described in class is based upon the following rules:

1. Each process in the system maintains a single logical clock 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 and process identifier to the message and forwards it to all of the other processes in the system. The message is then put in a "pending" queue until all of the other processes in the system have acknowledged receiving the message. The queue is sorted by logical clock time. In case of a tie, the message with the smaller process identifier goes first.
3. When a process receives a message from another process, it puts the new message in its queue (ordered by logical clock and process identifier) and sends an acknowledgement to all of the other processes in the system.
4. When a process receives an acknowledgement from another process, it puts the acknowledgement in the queue as well.
5. When a message has been acknowledged by all other processes in the system and is at the head of the queue, it can be removed from the queue and output. In our tool, we will output the messages that have been completely acknowledged to an output text file.

1.1 A note on ACKs

Processes don't have to keep track of who has sent an acknowledgement for a given message; a counter for each message suffices. Once the expected number of acknowledgements has been received and the message is at the head of the queue, it can be executed.

To make the tool simpler, we will ensure that all nodes receive n acknowledgements for each message received. We'll do that by having each sender record an ACK to itself when it sends a message. That ensures that all nodes will receive n acknowledgements.

If we didn't do that, we'd need to enforce a difference between the sender of a message and all other nodes. Consider a system with n processes P. Let m_1 be a message sent by process p_i.

1. p_i will receive $n-1$ acknowledgements - all processes except for itself.
2. All other processes ($p_j \in P$ such that $i \neq j$) will receive $n-2$ acknowledgements. The original sender p_i will not send an acknowledgement and p_j won't send an acknowledgement to itself.

As noted above, we will **not** be making this distinction in our tool - we'll ensure that all nodes get n ACKs.

1.1.1 Living without ACKs

As noted in class, it's possible to build the TOM tool without any ACKs at all. If all nodes submit messages regularly and we can enforce FIFO and reliable messaging (using ACKs or TCP), we don't need to use the ACK mechanism at all. It's sufficient to see a message from another node to know that it received all previous ones already.

If we implemented the system without ACKs, we would need to then compare who we have received messages from so far, not just counting messages as we did with ACKs.

1.2 Pending Messages and Output

When a message arrives, it is first put in the *pending queue* until it accrues enough ACKs to be moved to the output file. The tool therefore needs to check each incoming message in the following manner:

- If the message is a MESSAGE (see format below), enter it into the pending message queue immediately. Write down its important fields. Send an ACK for it to all other nodes (including yourself, whether by sending a message or just by adding 1 to the ACK counter).
- If the message is an ACK (see format below), find the corresponding message in the pending queue and increment the ACK counter for it.

When an ACK arrives, it's possible that one or more messages can be removed from the pending messages queue and output to the output file (one of the parameters above). Therefore, after the arrival of an ACK, you need to go over the pending queue and see which (if any) messages can be deleted from the queue and output to the output file. Keep in mind that only the head of the pending queue can be removed if it's ready. Any message behind the head must wait, no matter how many ACKs it has. If the head is stuck for some reason (ACKs are late) and a few other older messages are fully acknowledged, once the head is (finally) released, all of the ready messages behind it will be released as well.

2 Tool Interface

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

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

outputFilePath The path to write out the final processed 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.

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

2.1 Neighbors

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
```

```
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: /10.9.11.87
: 1
Started to listen on ServerSocket[addr=localhost/127.0.0.1,localport=5003]
Sending messages from: localhost/127.0.0.1:5003
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: MESSAGE-1-localhost/127.0.0.1:5003-Hello world!
Choose what to do:
1. Send a new message:
2. Print status
3. Quit
```

Figure 1: Tool Startup

```
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.

3 Protocol Details

The tools communicate among a group of statically defined neighbors. There are two messages sent between the neighbors:

MESSAGE-1234-10.0.0.2:5000-Text here The message uses - as a field delimiter. It has the following fields:

1. The logical clock of the sender at the time the message was sent
2. The IP address and port of the sender
3. The text of the message sent.

ACK-1234-10.0.0.2:5000 The message uses - as a field delimiter. It has the following fields:

1. The logical clock of the message that is being acknowledged.
2. The IP address and port of the sender of the message being acknowledged.

4 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

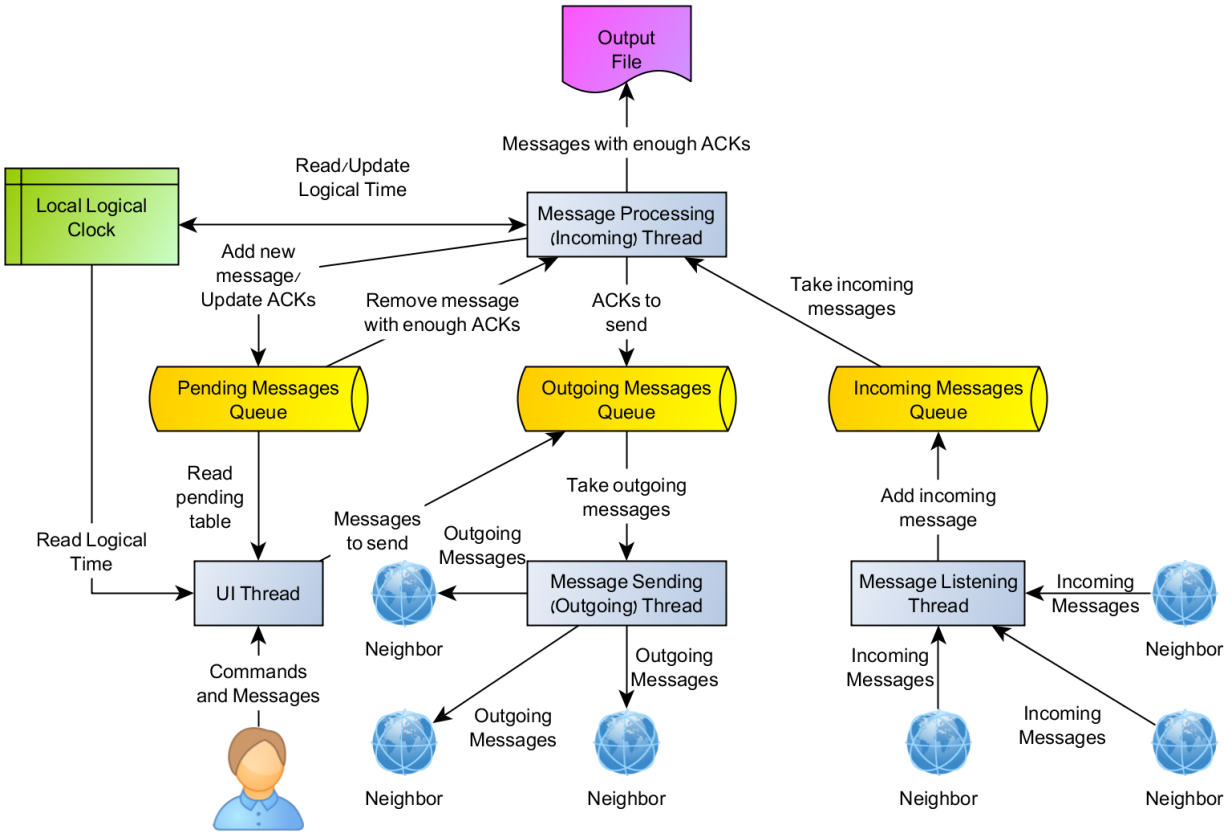

Figure 2: Queues schematic

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.

4.1 UI Thread

The UI thread offers a menu based interface that enables:

- Send a new message
- Print out the status of the pending queue and the current logical clock (see Figure 3). 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

4.2 Message Listening Thread

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

```
Logical clock time: 4
Pending messages:
1 ACKs on MESSAGE-1-localhost/127.0.0.1:5003-Hello world!
3 ACKs on MESSAGE-3-localhost/127.0.0.1:5003-Goodbye!
Choose what to do:
1. Send a new message:
2. Print status
3. Quit
: 2
Logical clock time: 4
Pending messages:
1 ACKs on MESSAGE-1-localhost/127.0.0.1:5003-Hello world!
3 ACKs on MESSAGE-3-localhost/127.0.0.1:5003-Goodbye!
Choose what to do:
1. Send a new message:
2. Print status
3. Quit
:
```

Figure 3: Printing the status

4.3 Message Sending (Outgoing) Thread

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

4.4 Message Processing (Incoming) Thread

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

- If the message is an ACK, it updates the pending message queue as mentioned above in Section 1.2.
- If the message is a MESSAGE, it processes it as mentioned in Section 1.2 and prepare an ACK to send. It then puts the ACK in the outgoing message queue to send to all neighbors (including the current node itself).

  As shown above, the ACK message doesn't contain the body of the message or the IP address of the acknowledging node, just the logical time stamp, IP address, and port of the original sender of the message.

5 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 statically, so each node must be a neighbor of every other node (a clique). If a neighbor is listed in the file and is not active, no participant in the group will be able to output any messages. It is not possible to add or remove nodes from the group during the course of a run.

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 can add an artificial "delay" to the tool. When greater than 0, the delay would cause the sending thread to delay before sending any message or acknowledgement.

Premature Acknowledgements If we introduce delay manually in the system, it is possible that a node will receive the acknowledgement of a message before the message itself (breaking FIFO). In such a situation, the recipient of the early acknowledgment must reserve an entry in the pending queue, register the acknowledgment, and fill in the message body when the actual MESSAGE arrives.

ACK Counters This tool doesn't keep track of which nodes have acknowledged which messages. It just counts the number of ACK messages received for each message.

6 What to do

I have given you a partial Intellij project for TOM, with most of the structure in place and labeled TODO tasks. You will fill in the following functionality:

1. Sending of messages to all neighbors when the send message command is given.
2. Receiving messages from neighbors and processing them accordingly.
3. Managing the pending table to ensure that messages which can be removed from the front of the queue are in the correct manner.

6.1 Extra Features

If you complete the tool as described in the previous section, add the following features to the tool:

Dynamic Groups Add JOIN and LEAVE messages to the protocol to let neighbors declare their entry and exit from the group. The use of the JOIN and LEAVE messages should be in addition to the neighbors file. The result is that the total number of ACKs needed depends on the number of neighbors who have sent JOIN messages before the message was sent. That way you are not waiting for ACKs from neighbors who are not online.

Failure Detection Add a HEARTBEAT message to the protocol which can be used to detect whether a neighbor is still around. Send out the HEARTBEAT every 1 second. If a HEARTBEAT isn't heard for 5 seconds, you can treat it as a LEAVE from the neighbor.

Long Lived Connections Adapt the tool to use long lived connections between neighbors. That means keeping multiple connections open - one for each neighbor. The use of long lived connections makes receiving and sending a bit easier to understand, but it makes the management of the connections and their status a bit harder. In particular, you'll need to open a bunch of Threads - one to listen for each active neighbor.