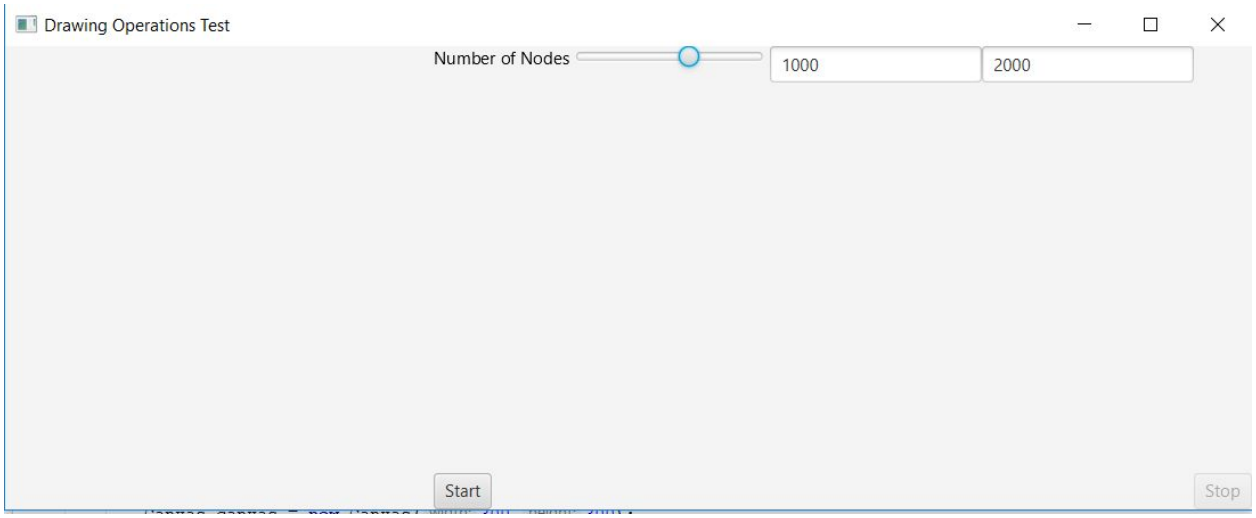
Miles McDowall and Daniel Michelin
3/16/18
COMP480
Dr. Howser

Purpose: This provides a graphic simulation of distributed nodes coordinating asynchronously to enter a critical section.

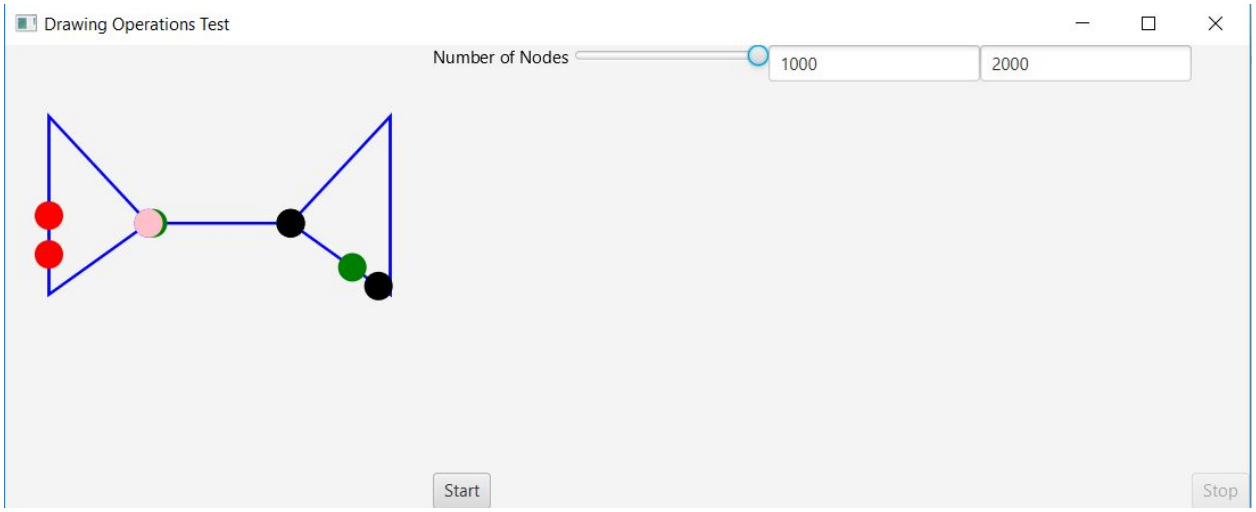
Usage: Compile and run the project or use the included JAR file. Use the slider to determine how many nodes you want to have in your simulation, then press start. The command line will show how the system communicates through each node one another, and the screen will show graphically what is happening.
Tracked statements:
Requesting Critical Region
Entering Critical Region
Sending acknowledgements
Unordered list of priority queue (priority queues are stored as a heap and so can't be iterated through in order without dequeuing each item in succession)

Before Initializing:



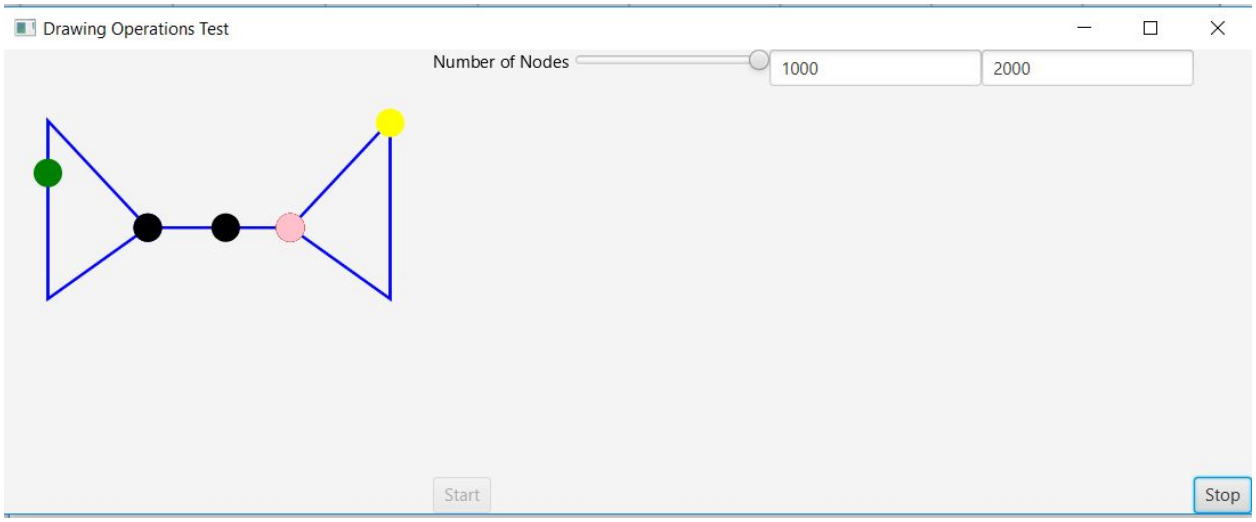
A user can select the minimum (left) and maximum (right) speeds as well as the number of nodes (from 1 to 10).
The start button will initialize the simulation.

When Stopped:



Here we can see the nodes moving about. Green is just entering the critical section. As the “Start” button is available, it is clear that we have stopped the simulation. Note that at this point the number of nodes & speeds cannot be changed.

When Running:



Here we can see a black node in the critical region. As “Stop” is available, it is evident that the simulation is running.

Algorithm Explanation:

The algorithm we use is Lamport's mutual exclusion algorithm. This algorithm essentially depends on each node asking every other one for permission to enter. If that node has the highest timestamp, then every other node sends an ack to that one. Otherwise send an ack to the highest node

Initial Process

1. Push its request in its own queue (ordered by Lamport time stamps)
2. Send a request to enter critical section to every node.
3. Wait for acks from every other node
4. If its own request is at the head of its queue and all acks have been received, enter the critical section.
5. Upon exiting the critical section, remove its request from the queue and send a release message to every process.

Other processes

1. After receiving a request from any node, push the request in its own request queue (ordered by Lamport time stamps) and reply with a time stamp.
2. After receiving release message, remove the corresponding request from its own request queue.

Problems:

We chose to use the javafx threads as our threading solution because it provides tight and easy coupling between the UI and the data model. Some problems we had were making sure the UI can properly shut down without an exception being thrown. The solution was to have a hashmap of each animation and cease their callbacks upon pressing the stop button.

A second problem we encountered was an issue with the priorityqueue and how objects were being dequeued from it—first, the compareTo method was sorting elements incorrectly, meaning that objects were entering the Critical Region out of order. Second, objects weren't being polled at the right time—peeks were necessary to ensure queue integrity but we were polling in such a way that a race condition was created.

We learned a few interesting things. First among them that handling timestamps without objective clocks can be very difficult. Most of our logic problems sprung from this issue. Secondly was dealing with threading lifecycles in javafx. They're easy to work with and their lifecycle is simple, however there is a bit of work to be done to make sure that the process is killed/paused correctly. Thirdly, both of us learned a lot about javafx (and why it makes sense that most people just want to use Electron for GUI applications).

Recommendation:

Yes. It teaches people UI and asynchronous development, both crucial skills for beginning software developers. It was also a very insightful look at how distributed systems can go wrong, which is very helpful for real-world situations where a developer would otherwise need to solve these byzantine errors without any experience in doing so.