

Results and Findings from Logical Clock Tests

We created 3 virtual machines communicating with one another. We ran each virtual machine 5 times, 1 minute each. The value for the clocks ticks was generated randomly at each run. For the following results, the clock ticks were generated from the range 1-6.

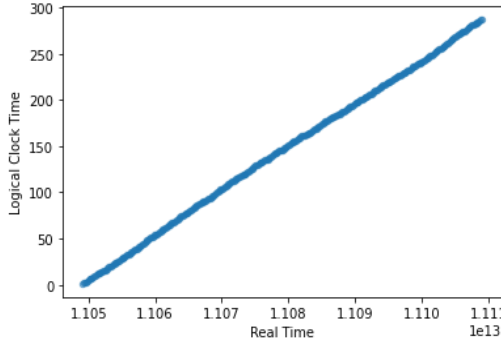
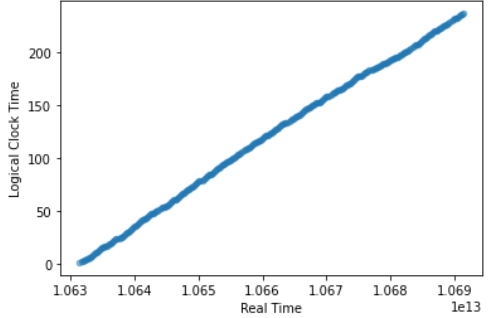
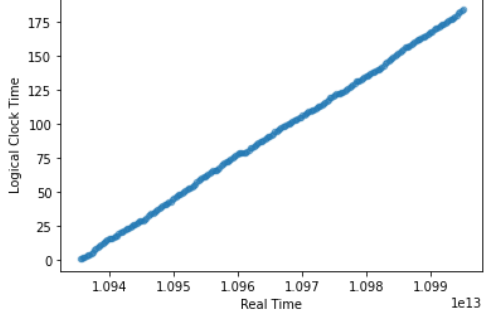
Number of Ticks	Run	Maximum Jump in Logical Clock Value	Maximum Length of Message Queue
6	VM1 RUN 1	1	0
6	VM3 RUN 3	1	1
6	VM3 RUN 5	1	0
5	VM3 RUN 1	3	1
5	VM3 RUN 2	1	0
5	VM1 RUN 2	1	0
4	VM1 RUN 3	8	1
4	VM2 RUN 3	13	1
4	VM2 RUN 4	1	1
4	VM2 RUN 5	28	0
3	VM1 RUN 5	13	2
3	VM3 RUN 4	6	1
3	VM1 RUN 4	8	1
1	VM2 RUN 1	13	9
1	VM2 RUN 2	10	9

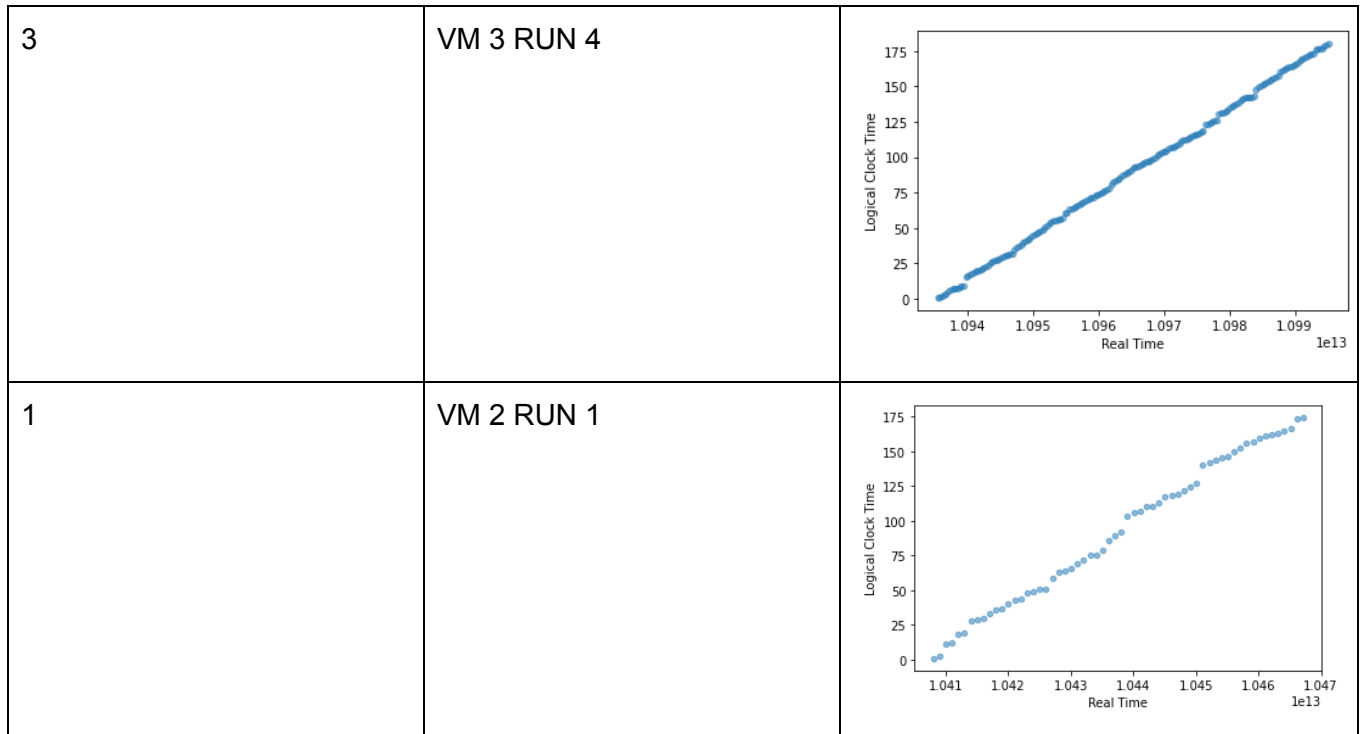
It can be noted that higher ticks per second are associated with smaller queues and smaller jumps in the logical clocks. This makes sense since the higher ticks means program makes more operations per second and so the message queue is likely to be shorter at any given point than for machines with slower logical clocks.

It is interesting to note that clock tick of 4 sees the biggest jump in the logical clock value; we would expect the clock 1 to have the biggest jump since there is a higher probability of clock 1

having to update its clock to a much higher value based on a faster clock. However, this behaviour is certainly possible; in longer runs of the program however, we may see the highest jumps corresponding to lower clock tick values.

Drift Plots

Number of Ticks	Run	Drift Plots(Real Time Plotted Against Logical Clock Time)
6	VM 3 RUN 5	
5	VM 3 RUN 2	
4	VM 2 RUN 4	



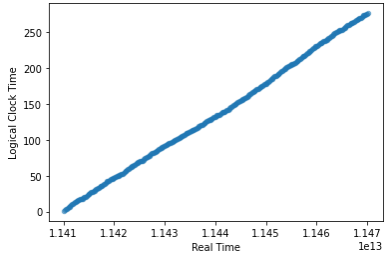
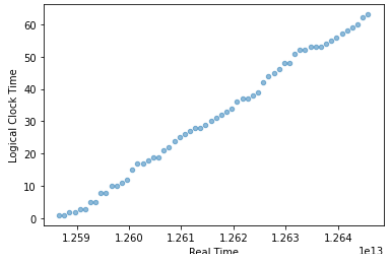
It is important to note that we see significant drift happening as the number of ticks decreases.

What happens when there is a Lower Probability of Internal Events Happening?

We notice that the length of the message queue is significantly longer than in the normal scenario where the maximum message queue length achieved was 9. The average jump when the clock tick was 1 per second was around 12 in the normal scenario: in this case however the average jump size has decreased to a 2.

Probability of Interval Event(as compared to the original Probability)	Number of Ticks	Maximum Jump	Maximum Length of Message Queue
50%	6	1	0
25%	1	3	95

Probability of Internal Event	NumTicks	Drift Plots
-------------------------------	----------	-------------

Happening		
50%	6	
25%	1	

What happens when we vary the range of clock ticks?

- I. Clock ticks range is between 3 and 5 instead of 1 to 6.

Number of Ticks	Maximum Jump in Logical Clock Value	Maximum Length of Message Queue
3	9	1
4	4	1

- II. Clock ticks range is between 1-3.

Number of Ticks	Maximum Jump in Logical Clock Value	Maximum Length of Message Queue
3	2	1

Design Notes and Decisions

Overview of Classes

The first thing we focused on in this project is the high level design of the classes we wanted to create and how they would interact with each other. We started with the outline of a VirtualMachine class that would include the functionality for the logical clock. We decided to start with that class because it had all new functionality versus our last design project, and the other class designs would depend on how we approached the VM class.

After deciding on the functionality of the VirtualMachine class, we then designed the other classes based on that design and our learnings from the chat project. Our original notes for the different classes and basic relation diagrams are included below:

VirtualMachine Class

- constructor
 - Takes in (ticksPerSecond) and sets it
 - Set log file
 - Create an ArrayList or something for message queue
 - Initialize logical clock // integer or array of integers
 - VirtualMachineThread
- sendMessage // either one method or two separate ones
 - Takes in machine/machines to send message to
 - One machine
 - Both machines
- readMessageFromQueue
- internalWhatever // not sure if this is something we have to implement
- generateRandomNumber // generate a random number to determine action
 - If 1, sendMessage to x
 - If 2, sendMessage to y
 - If 3, send to machines x and y
 - Else, internal event
- checkForMessageInQueue // every clock tick first check for a message in queue
 - If yes, readMessage
 - If no, generateRandomNumber
- updateLogicalClock // update based on either time from message or the current logical clock time
 - If you readMessage from the queue // not sure this is really where logic goes

- If the time in the message is < the current logical clock, keep current clock value // or maybe increase by 1, but I think it's supposed to stay the same because the event is essentially useless at this point
- Else you update the logical clock to be that time you read + 1 // I think it's +1 to show that it was causal from the time the message was sent
- logEvent
 - Takes in event
 - Write event, systemTime, logicalClock
- addToMessageQueue(Message m)
 - Adds m to the field messageQueue

Main class

- Use random number generator to determine the ticks per second for each VM
- Create all virtual machines (VirtualMachine vm1 = new VirtualMachine(4);)

Server class

- Creates a ServerSocket at a specified port for clients to connect to
- Each time a new client connects, creates a Socket and Thread for that client
- Give each client a unique id to keep track of the different virtual machines for message sending

Message class // To pass messages between virtual machines

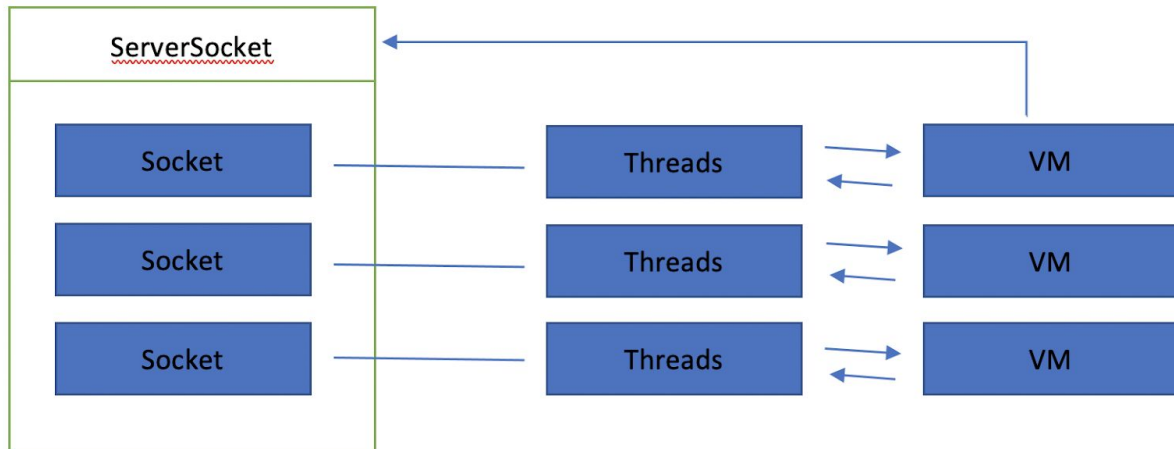
- Sender and recipient as virtual machines
- Logical clock (which is the message)

VMThread Class // the virtual machines need to be threaded so they can take in messages from the server as they are running

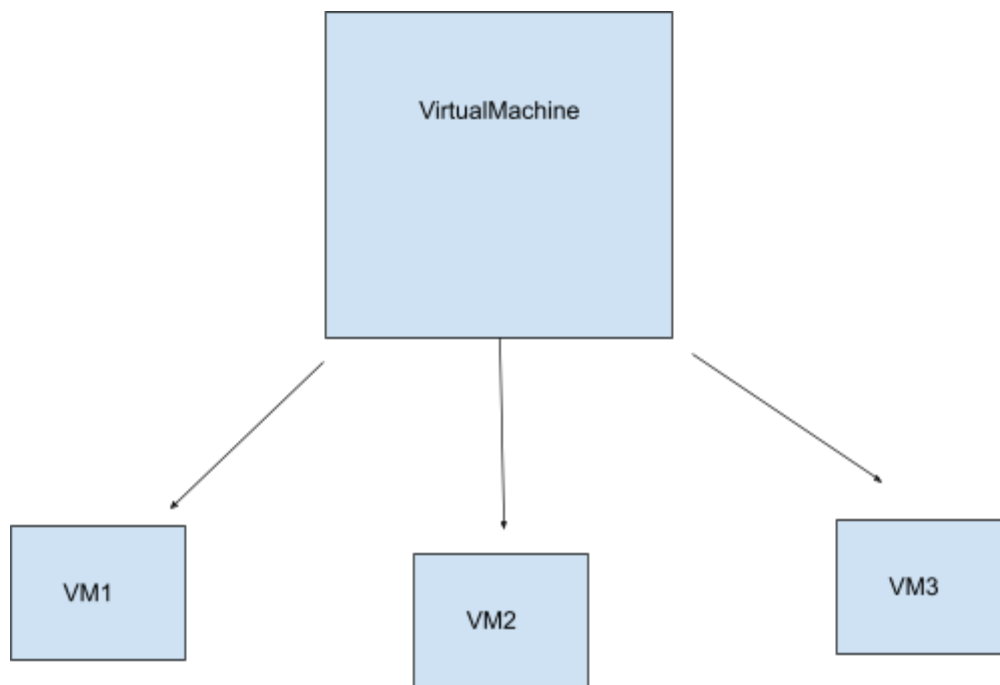
- addMessageToVMQueue // take messages from server and put them in queue for VM

Class Diagrams

Our thoughts of the relations of classes for connecting to the server were inspired by our chat program. We chose to have one server with one ServerSocket that would create a new Socket and "ServerThread", which handles receiving messages from the VM, for each VM when it connects. In addition, when each VM is created, it makes a separate thread, the "ListenerThread", which handles receiving messages from the server. By using all of these threads, we could have multiple VMs connect to the server at the same time and also have the VMs send and receive messages at the same time.



We also have a diagram highlighting the template nature of the `VirtualMachine` class. In actuality, we chose to put the main method in that class instead, so this image does not quite capture the model we have. However, the functionality of the `VirtualMachine` could easily be extracted into a separate class if desired.



Sending Messages

One of the biggest design decisions and struggles we faced was around sending messages between the server and virtual machines. Our initial plan was to be smart and use RMI, as Jim suggested. We built a simple server and client program to test out RMI and got the communication working, but ran into some issues when trying to do more than send simple messages.

We then decided to try something (presumably) simpler and use Java's object serialization, but ran into a number of issues with that. Most of our issues seemed to boil down to the way that object serialization (and perhaps RMI) assumes the code works. Essentially, object serialization using `ObjectInputStream` and `ObjectOutputStream` assumes that there is always an object to be serialized, or it just closes the connection. So in setting up threads that were waiting to receive objects, we kept getting issues with Sockets closing, exceptions throwing, etc.

We spent a lot of time debugging and finally decided to just abandon the object serialization this time and instead use `BufferedReaders` and `PrintWriters`, which worked instantly. So we got things working, which is of course a success, but we have some work to do to figure out how to pass objects in threaded environments.

Getting Virtual Machines to Run at the Same Time

This was another fun adventure and learning experience in Java. We tried a number of different ways to get the three virtual machines to begin at the same time (or as much at the same time as is possible).

Our first attempt was to use a sort of broadcast from the server to tell all of the machines to start running. We found ways that we could send broadcast messages with `Datagrams`, but then got bogged down by adding more connections, especially that would be used just to start the programs. We then decided to try just sending a specially formatted message from the server to each VM once three had connected, but found that there were sometimes still issues with that message being processed at the same time.

We then discovered the `Java Executor` class, which was created to run threads simultaneously, so changed the VM class to run as a thread and executed a pool of 3 threads to run at the same time. Well, it turns out that running at the same time does not actually mean running at the same time for executors. Essentially, the executor can run the threads at the same time, but it's not necessarily at the same rate or at the same start time. And we found that this changed with every run, likely because the tick values changed, so the executor focused on running the faster threads first.

Finally, we discovered that you could very easily run programs at the same time in bash scripts using a simple `&` symbol between each program call. So we ended up using that.

MessageQueue

For the message queue, we chose to use a LinkedList implementation. Even though it takes up more memory than some other data structures, it is one of the faster (if not the fastest) option for adding elements and taking elements from the front of the queue, which were the two main actions we needed.

System Time

A final design decision we made was around system time. There are a few ways to get the system time in Java and apparently a lot of debate around which method is the best. We chose to use `System.nanoTime()` because it is supposedly impervious to going backward if the user resets the computer clock. Although we did not need time to the nanosecond, we felt that building a practice of using better time methods is good to start now.