Code:

https://github.com/PerdoGarcia/cs262/tree/main/ps3

Engineering Notes:

3/1/2025

- Tentative plan for the machines
    - 6 threads in the same program, two for each "machine"
    - One thread is listening and updating a shared array (with a lock) with new events
    - The other thread is the machine
        - Has a while loop, while it is going on it waits until the next (clock) second, does some number of operations, and then waits
        - Logical clock is just some int
        - Learned about how to kill threads nicely: https://stackoverflow.com/questions/11436502/closing-all-threads-with-a-keyboard-interrupt
    - Use python random library for generation of random numbers https://docs.python.org/3/library/random.html

3/2/2025

- Started coding up 1 machine
- Each machine will start by running 1 thread, which will then create 2 smaller threads
- Passed in parameters for each machine: clock speed (picked by main thread, not the machine's thread), port number (that will be opened for listening)
- Decided instead of having 1 thread create 2 threads, I can just have the originally spawned thread be the listener for each machine, and just make a subthread for executing the instructions
- Decided on using sockets + json because gRPC requires a lot of setup (proto) and I am not super familiar with Python classes and I already have my own threading going on and don't want to deal with threading to use gRPC
- Decided to use the multiprocessing Queue implementation from this library (https://docs.python.org/2/library/multiprocessing.html#multiprocessing.Queue) to avoid race conditions

- Running into an error where the sub-threads are not being joined when the main thread tries to exit with a keyboard interrupt
    - Fixed the bug - turns out the selectors.DefaultSelector.select() function blocks if you set timeout = None, so the whole thread was blocked up and hence unjoinable. Hence, I set the timeout to -1 in order to poll so that the thread polls instead so that when I hit ctrl + C (maybe later change to like a 1 min timeout for the experiments) the thread joins + main program exits correctly
- Realized shouldn't use threads, should use process, changed as needed

3/3/2025

- Plan for the machines
    - On run_event, all machines run together, connecting to each other, then delivering messages on hard coded ports. Using the thread safe queue the messages are then received with sockets + json.
    - Using a lot of sleeps just so that things work for trials, will change later
- Socket won't connect sometimes but sometimes it will, perhaps wait for all connections to start before running
    - Run_messages - waits for all connections to connect using connect_to_other_machines
- Having issues where machines won't connect to each other correctly
    - Introducing a mapping connections that is stored
    - Kept getting `BlockingIOError` when waiting for messages during connections
- Confused on what is actually printed in the logger

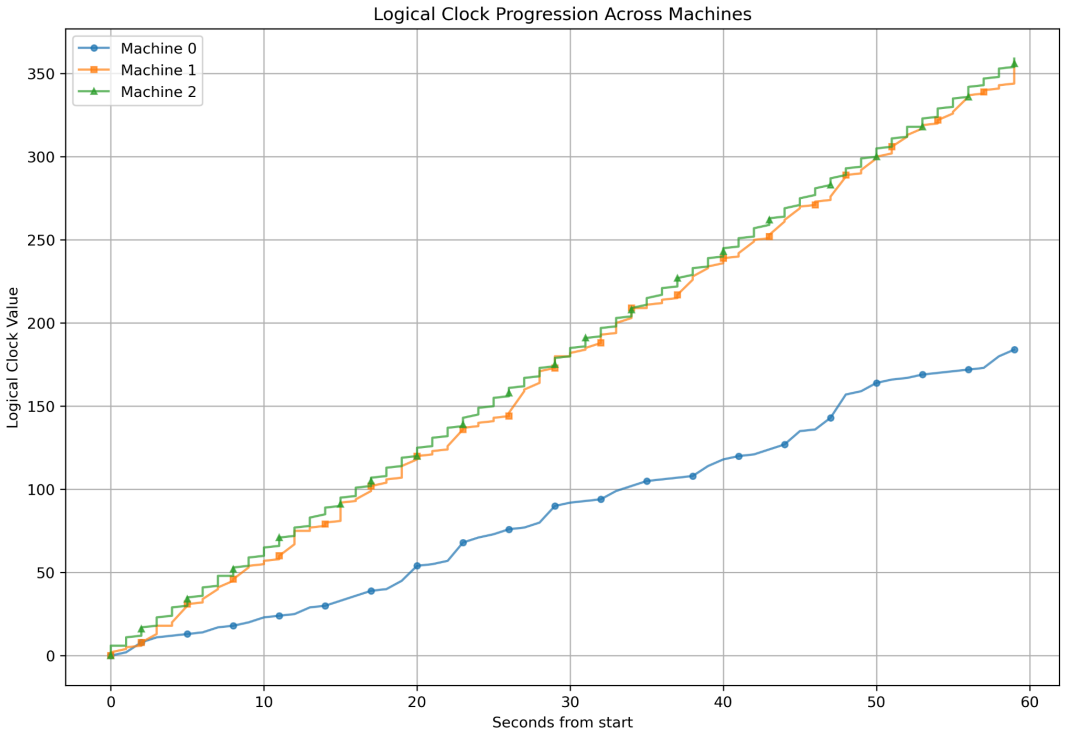
3/4/2025

- Fixed timing issues with both logical clock and how I was processing time
- Basically the jumps felt so inconsistent and the time would actually go down
- After discussing with a friend I changed the clock to let the logical clock increment
    - ```
      clock = max(clock, message["time"]) + 1
      ```
    - Where message["time"] is the time of the message sent, and clock is the logical clock time of the recipient system
    -
- Implemented the tests and experiments after I fixed the logging
      ```

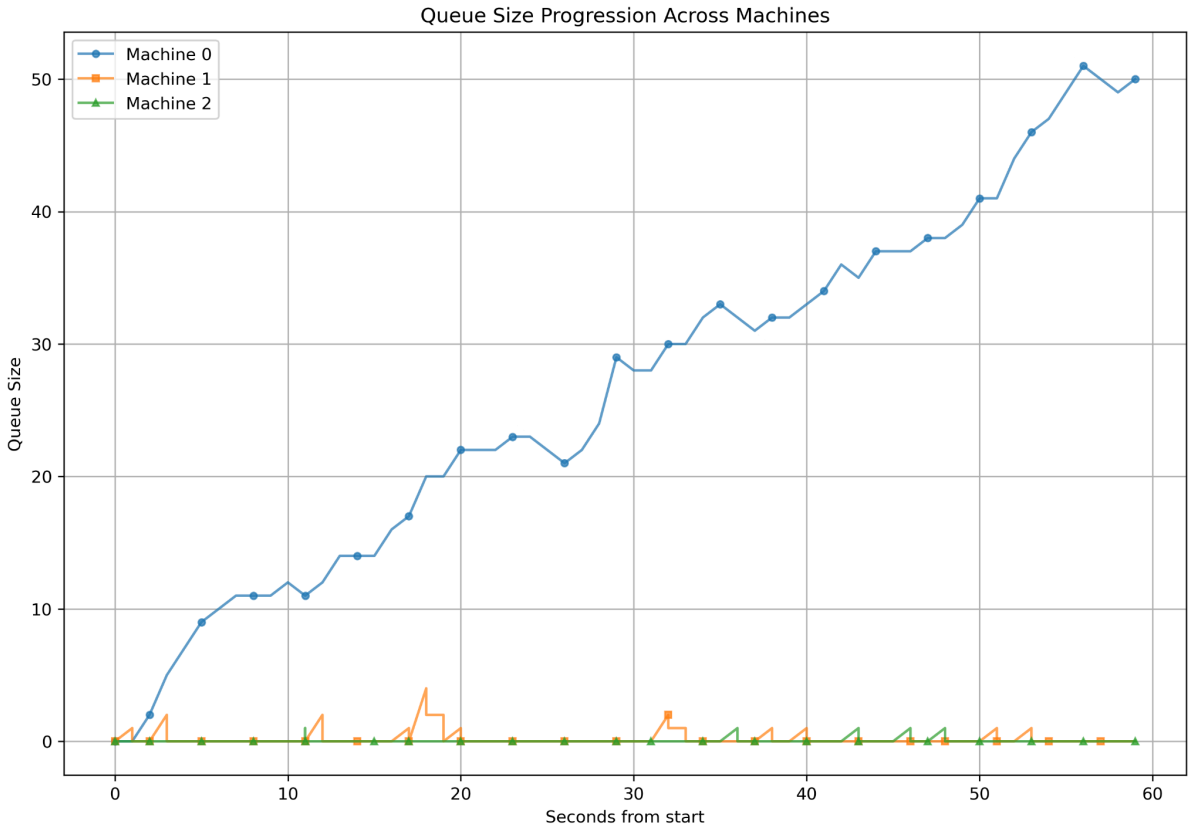
- Analyzed 5 trials as well as the additional experiments
  - Check the folder for all the results, but generalized results are down below
- Did use AI for the production of the graphs and unit testing

Observations/Conclusions:

We will first be looking at the trial with graphs of a clock speeds 1-3-6 (where machine 0 has a speed of 1, machine 1 has a speed of 3, and machine 2 is the fastest.

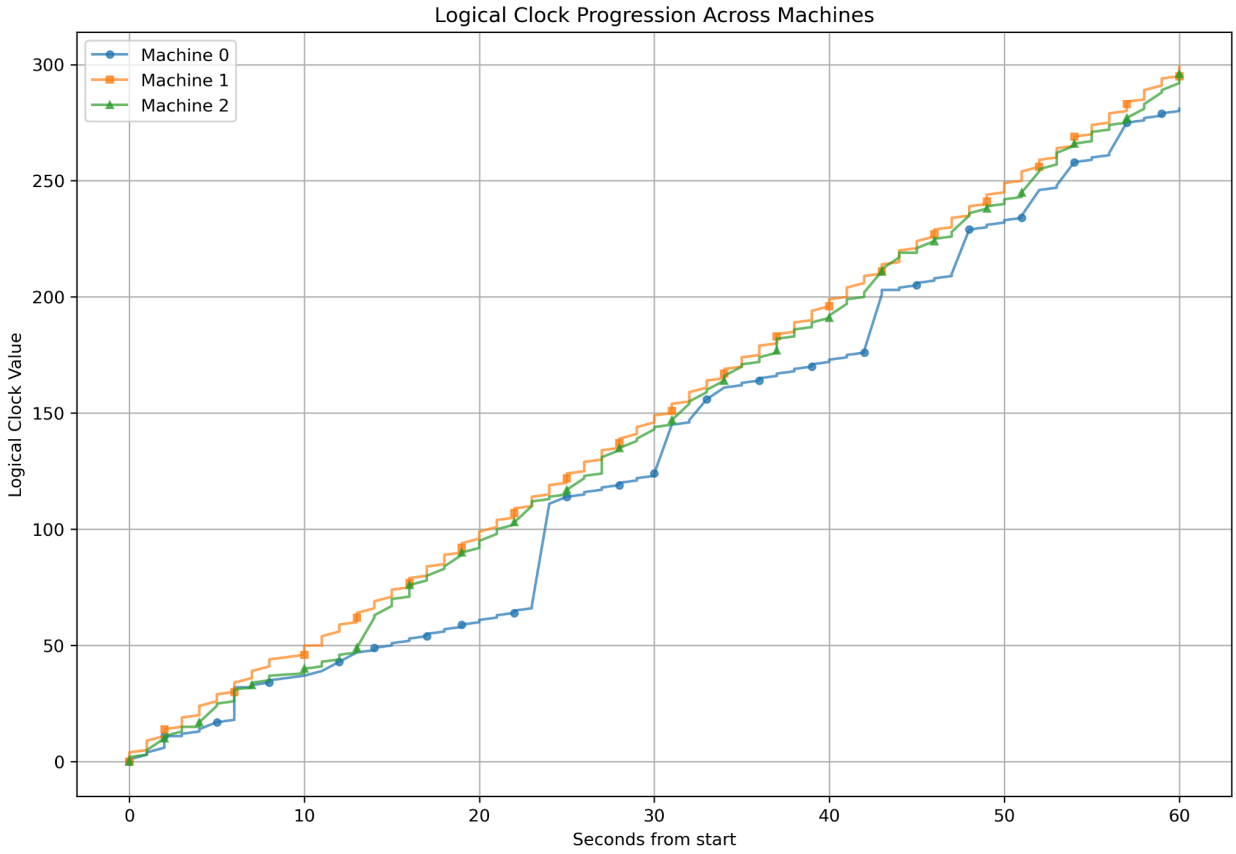


- This displays the logical drift between the clocks as they run their simulation. Across all trials with different combinations, it is clear that the fastest clock is the most up to date regardless of the other two. The higher the clock rate, the more up to date the logical clock value actually is. The converse also holds.
- It was interesting to see that if a big jump was made it was most likely from an internal event to a receive. So it was more consistent/reliable to get a clock from interacting from another machine rather than stick to the own machines' logs.

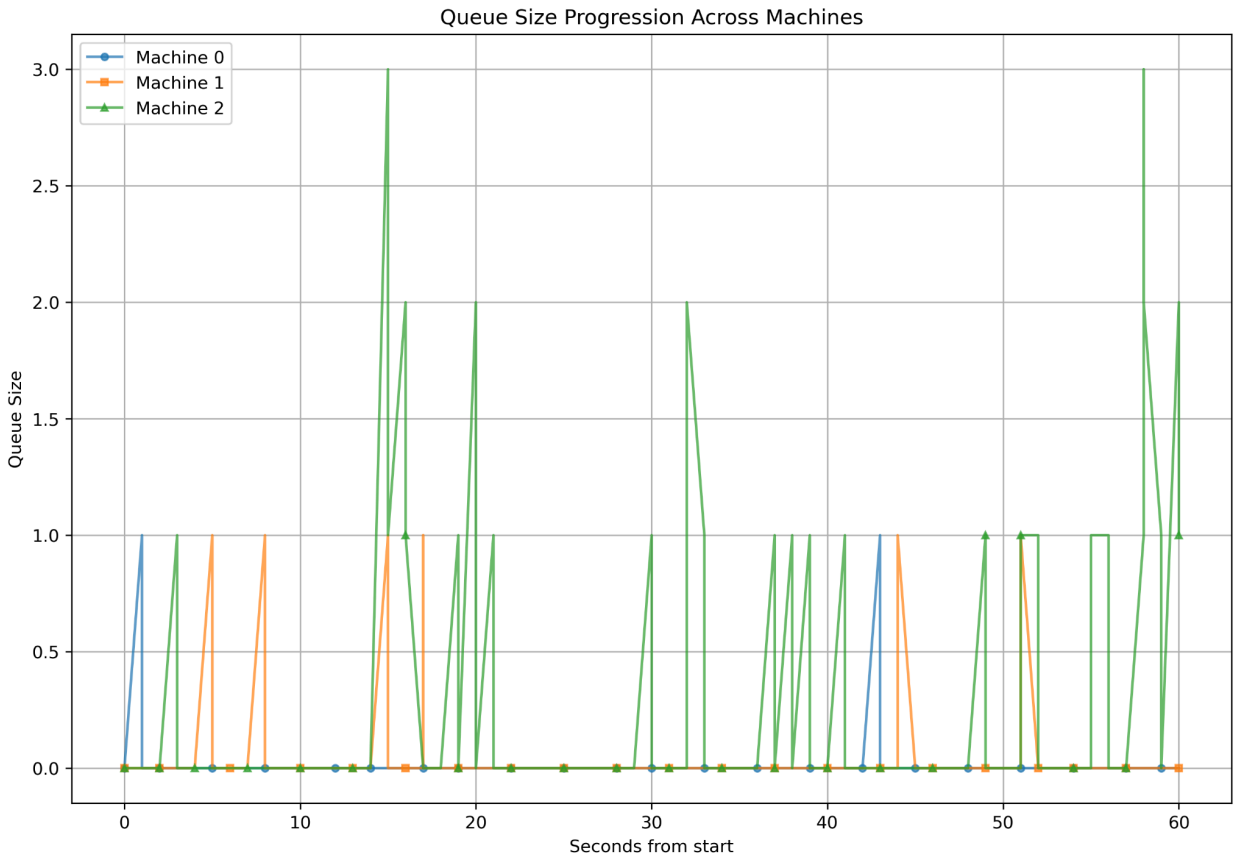


- IN the same simulation, it is clear to see that process 0 (having the slowest clock speed) constantly had an overflow of messages and only had “received messages” for most of the trial. This was clearly because the machine was overloaded by the two other machines and couldn’t empty its queue faster than it could receive messages. You can see machine 0 ended up with a queue size of 50 around the end!
- The two other machines could empty their queues more efficiently because they were never swamped with receives
- The event distribution also reflects this pattern, with slower machines experiencing predominantly receive events and faster machines rarely receiving messages.

These are now results from a 2-5-3 clock speed (respectively for machine 0/1/2)



- Whenever the clock speeds are not so far apart, we notice that the drift between the messages isn't so strong as when there's a significant disparity in clock speeds. So when machines operate at similar rates, their logical clocks advance more uniformly, leading to fewer dramatic clock jumps when messages are exchanged.



- The queue sizes fluctuate based on the balance between incoming messages and processing rate.
- It isn’t particularly interesting, but it shows the dramatic difference between the 1-3-6 graph.
- It appears that adding more clocks/machines will cause increased message inflow to the relatively lower speed machine even if they have high speeds.
- In contrast, when machines operate at comparable speeds (like this test), the system demonstrates better synchronization with minimal queue growth, less logical clock drift, and smaller clock jumps between machines.
- Clock jumps can be significant even with similar machine speeds, often exceeding 10 time units, though machines with vastly different speeds don't necessarily show proportionally larger jumps than those with moderate speed differences.
- Simulations with fewer internal events show more frequent and extreme clock divergence since faster machines send more messages while slower machines, already primarily processing messages, become even more overwhelmed with incoming communications.
- Reducing internal events also decreases maximum clock jumps because the relative proportion of communication events increases, leading to more frequent clock synchronization across the system. This was noted earlier.