Scale Models and Logical Clocks

Code Repository: https://github.com/acheng257/cs2620/tree/main/scale_model

Mar 2, 2025

We started by defining a base class for the machines used in our simulation. We defined functions such as reading and sending messages between machines and implemented the initialization of clock rates and randomly choosing what action to perform.

"*Each model machine will run at a clock rate determined during initialization. You will pick a random number between 1 and 6, and that will be the number of clock ticks per (real world) second for that machine*"

For that part of the specification, we followed by defining a **clock** and a **clock_rate** that will be used as a property of the Machine class.

```
class Machine:
    def __init__(self, id, host, port, neighbors):
        self.clock = 0
        self.clock_rate = random.randint(1, 6)
```

The **clock_rate** is used later on for managing the speed of the execution of the machine, as shown in the excerpt below:

```
    def main_loop(self):
        time_per_tick = 1.0 / self.clock_rate
        while self.running:
            time.sleep(time_per_tick)
```

"*The (virtual) machine should listen on **one or more sockets** for such messages.*"

For this part, we decided that each machine would have one socket for listening. As in this design exercise, we only have two machines, so having only one socket should suffice for all the needs we may have with regard to scale.

"*Finally, each machine should have a logical clock, which should be updated using the **rules for logical clocks**.*"

If e, e' are consecutive events in process p that are not message receive events, then

$$C_p(e') = C_p(e) + 1$$

If e, e' are consecutive events in process p and e' is a receive event with timestamp T, then

$$C_p(e') = \max(C_p e), T) + 1$$

Mar 3, 2025

We used a JSON message format with sockets for our communication mechanism for simplicity. So, each machine will communicate with the others through sockets.

"*Each machine will also have a **network queue** (which is not constrained to the n operations per second) in which it will hold incoming messages. The (virtual) machine should listen on one or more sockets for such messages.*"

For each machine queue, we decided to use the Python queue implementation, which allows us to take advantage of the queue properties through an optimized implementation. Since we are using a FIFO storing data structure, this makes more sense than using Python's default implementation of a list, which would give us a removal time of O(N).

We wrote a script to start three machines that communicate with each other and run for 60 seconds. We then ran a Python script to plot the content of these logs, precisely the values of each machine's clocks and the number of messages in each of their queues over time.

We did this by running a script that activated the three machines at the same time. We performed this action 5 times and recorded the logs, which will be analyzed in the sections below.

As asked in the exercise description, we also modified the variation in the clock cycles and changed the probability of the event being internal to a lower probability.

Results

In this section we explore and discuss the different results we obtained in this experiment.

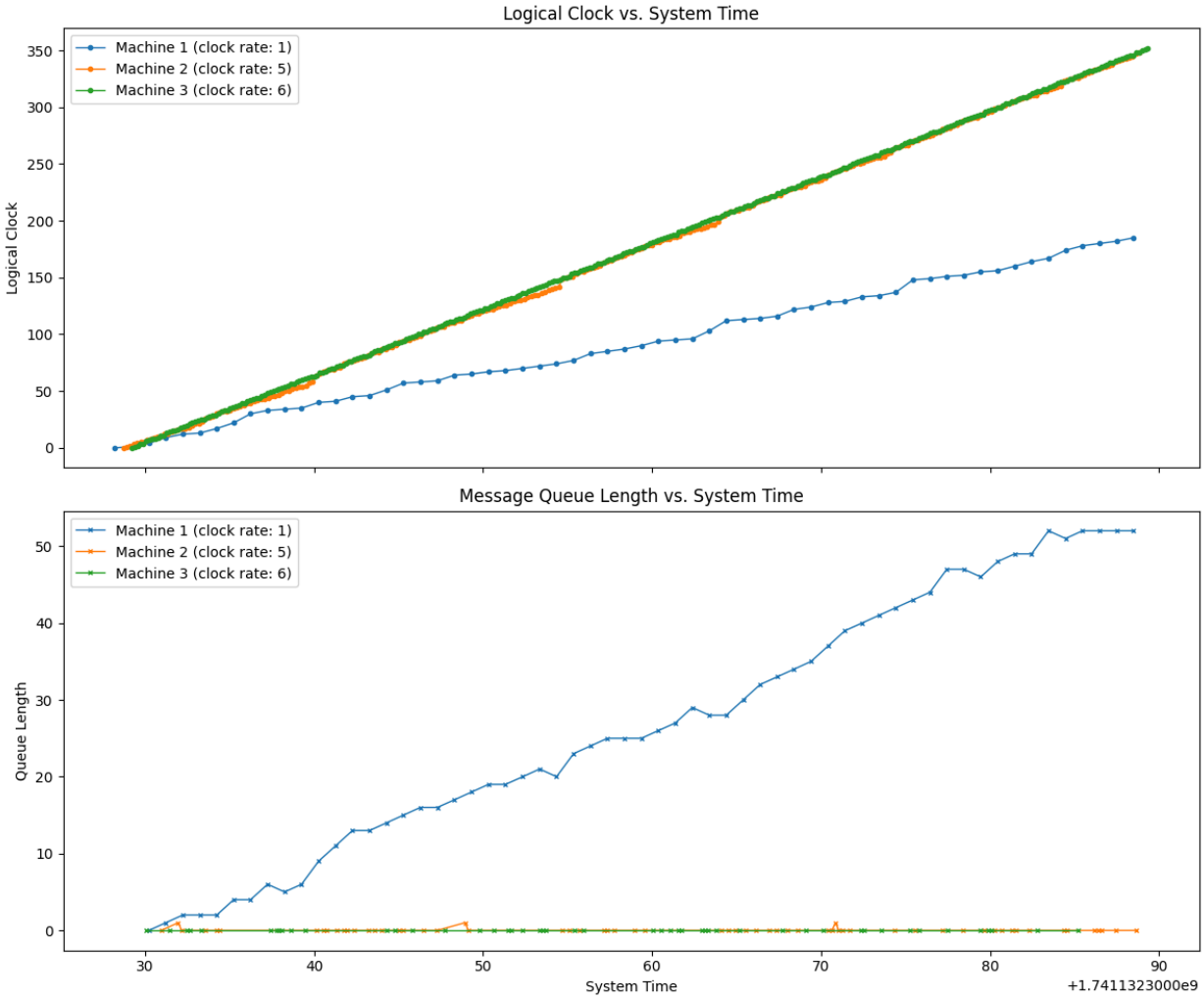
Clock Rate Between 1-6 and Internal event with 0.7 probability

For the first run, we had the initial configurations with the clock rate being a number between 1 and 6 and the internal event with 0.7 probability. We conducted the experiment of running the machines for 1 minute and registering the logs, which gave us the following results:

Single Slow Machine

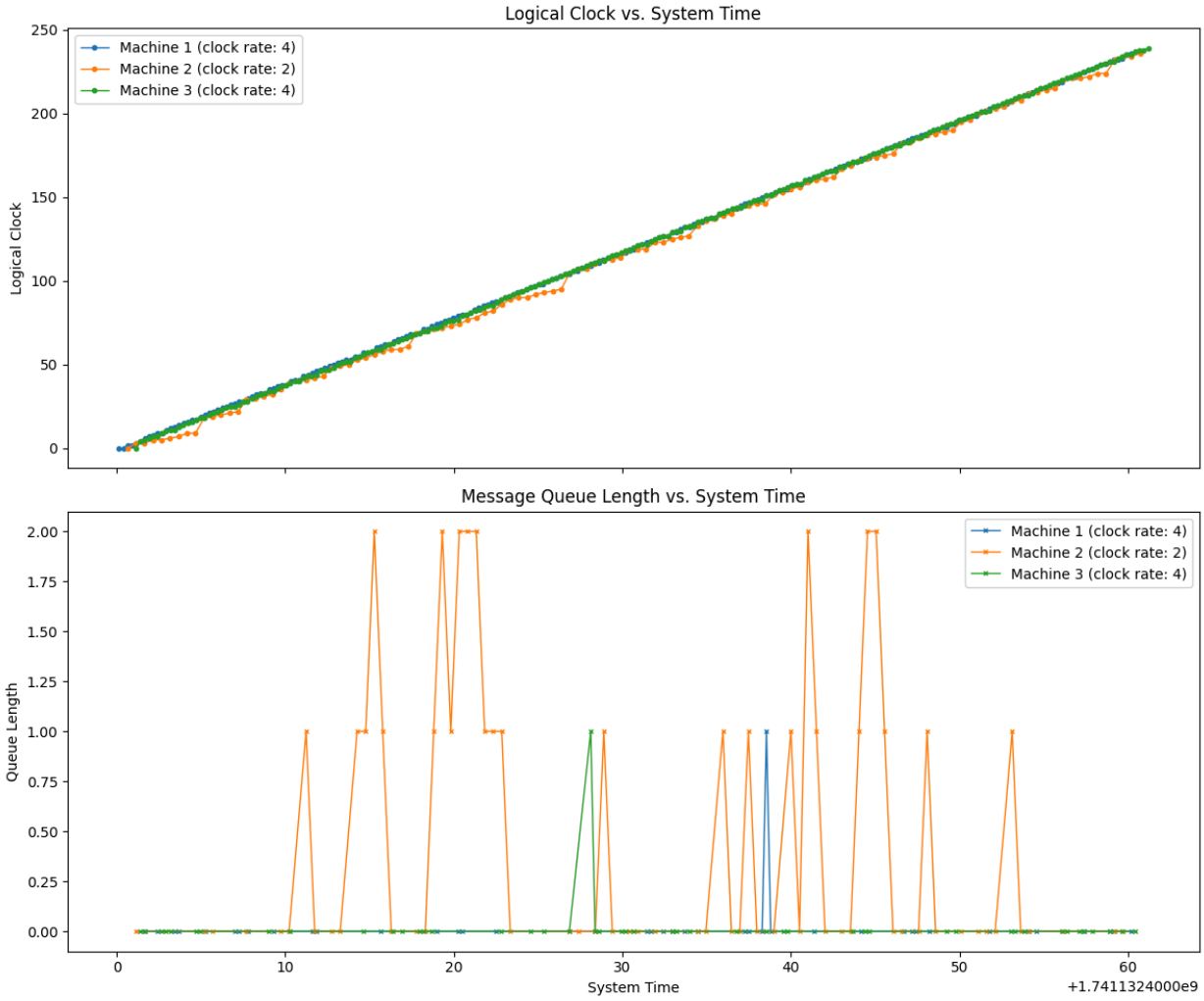
A **jump** in a machine's logical clock occurs when the machine receives a message with a timestamp higher than its local clock. We can observe that in the **Logical Clock vs System Time** plot below, where we have one Machine with a clock rate of 1 and the other two machines with clock rates of 5 and 6. These jumps usually happen in the slower clock machine.

Drift refers to the gradual divergence of one machine's logical clock from another's over real (system) time. Unlike jumps (which happen in a single update step), drift is more continuous and results from each machine's distinct clock rate. In this example, we can observe that the queue of Machine 1 grows indefinitely (**Message Queue Length vs. System Time** plot), which also explains the drift in its logical clock.


Logical Clock vs. System Time
Machine 1 (clock rate: 1)
Machine 2 (clock rate: 5)
Machine 3 (clock rate: 6)
Logical Clock
0
50
100
150
200
250
300
350
Message Queue Length vs. System Time
Machine 1 (clock rate: 1)
Machine 2 (clock rate: 5)
Machine 3 (clock rate: 6)
Queue Length
0
10
20
30
40
50
30
40
50
60
70
80
90
System Time
+1.7411323000e9

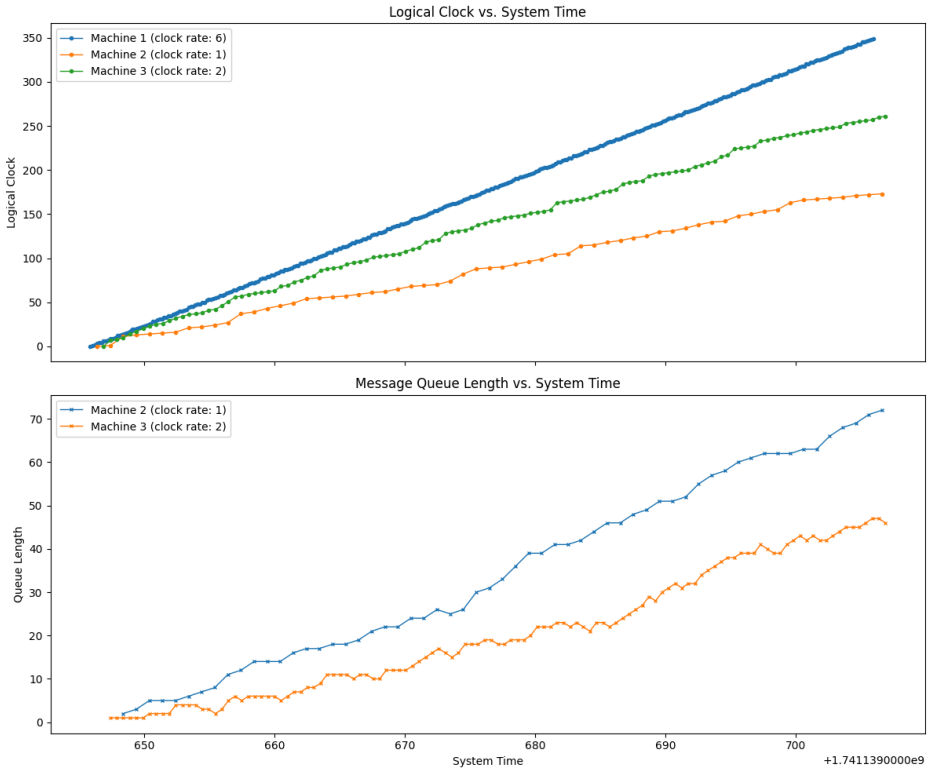
Similar Clock Rates

When the machines have similar clock rates, we still see that clock jumps continue to happen, but the drifting no longer occurs. The Queue Length also does not grow infinitely, and it varies and increases more for the lower clock rate machines.



Clock Rate Between 1-6 and Internal event with 0.4 probability

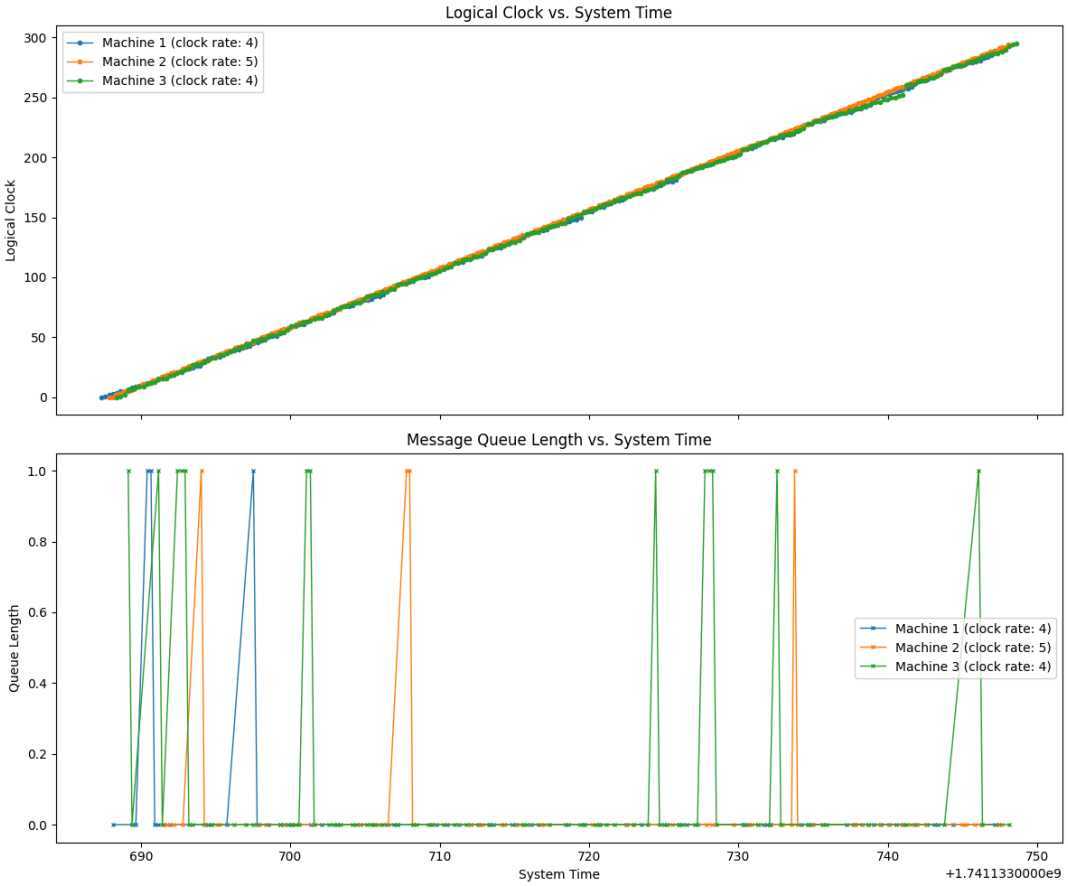
We modified our code such that the probability of machines performing internal events, rather than sending messages between each other, was 0.4 instead of 0.7. An example of the logical clocks over a period of 60 seconds in the real world is shown below.



We observe significant drift in machines 2 and 3. Machine 1 operates at a very high clock rate, and with a reduced probability of internal events, machines exchange more messages containing the sender's logical clock timestamp. As a result, the updates to the receivers' clocks are more frequent but less pronounced. However, the overall drift is greater than when the internal event probability was 0.7, because the increased message frequency leads to a backlog in the slower machines' queues—they cannot process incoming messages as quickly as machine 1 sends them. From the plot, we observe that the message queue in both machines 2 and 3 continue growing with time.

Clock Rate Between 3-5 and Internal event with 0.7 probability

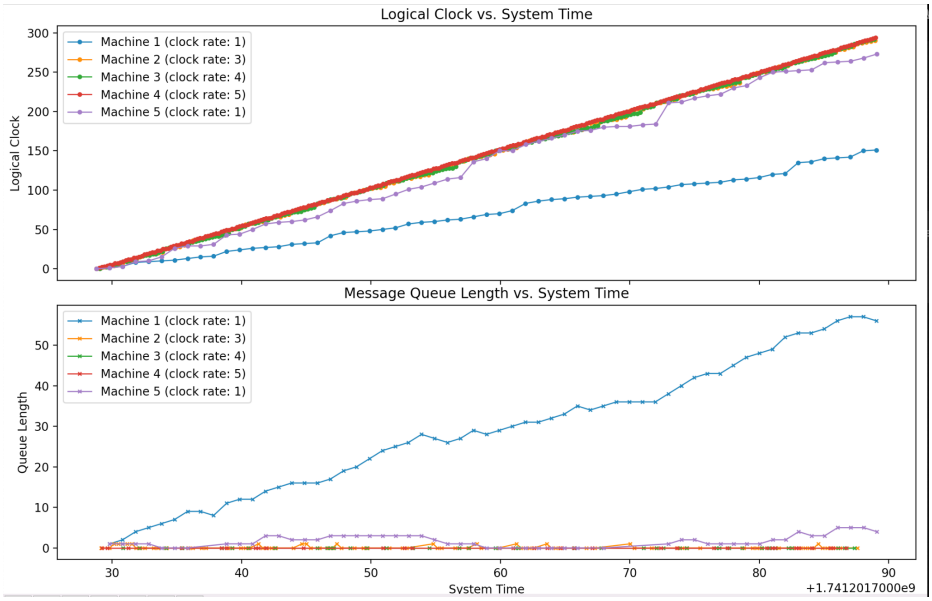
For our final experiment, we changed the clock rates of the clock to be between 3 and 5, instead of 1 and 6. An example of one of the trials is shown below:



With this configuration, the drift between machines' logical clocks is markedly reduced. Because the clock rates are more uniform, machines process incoming messages at roughly the same pace, preventing the message queues from building up—evidenced by the plot showing a maximum of one message per queue. Additionally, the logical clock increments remain small since messages are processed promptly, minimizing the difference between the receiver’s clock and the message timestamp.

Extra Experiments

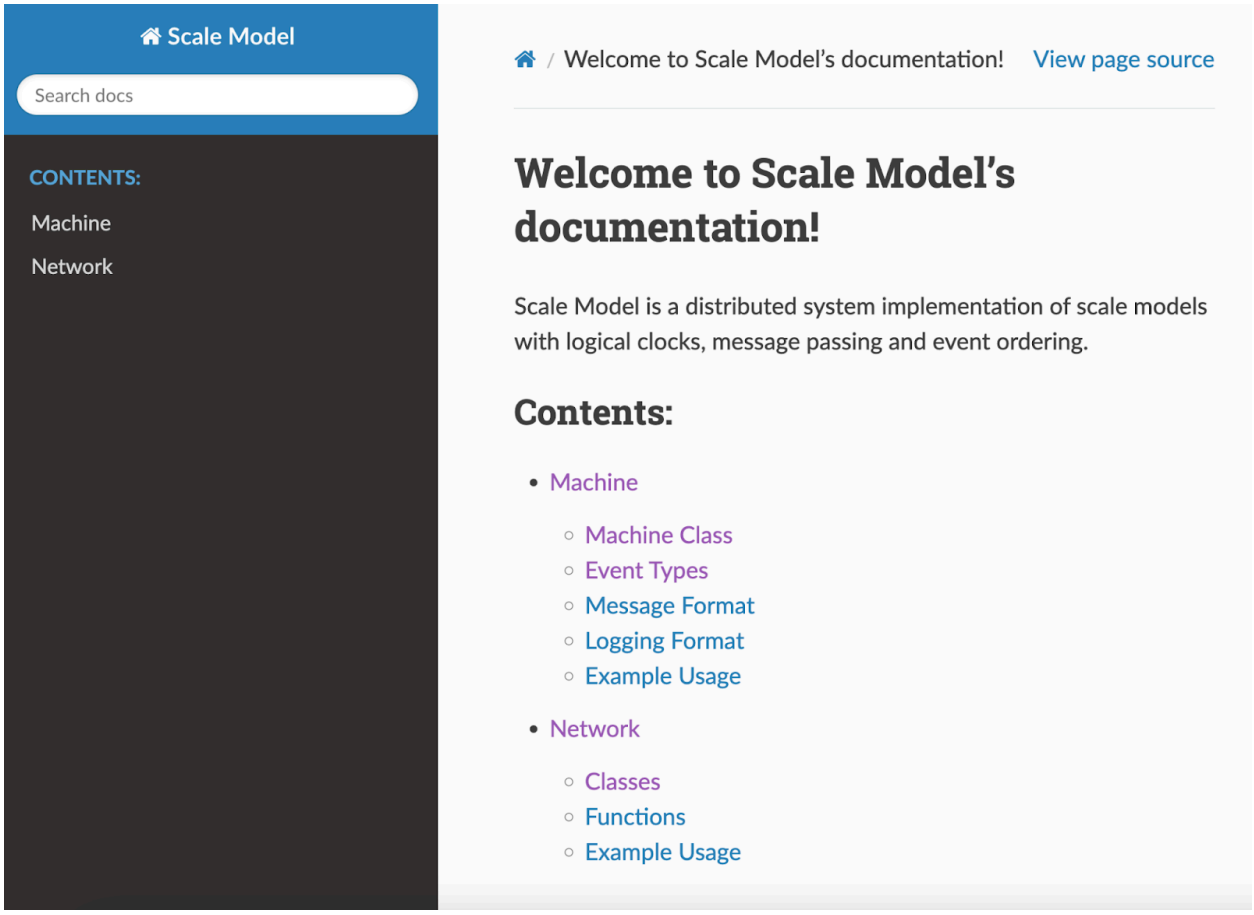
5 machines instead of 3



In this experiment, we increased the number of machines to 5, but each was initialized with the default parameters in our original experiment. An interesting note is that machine 5 showed larger jumps in the logical clock but little drift, while machine 1 showed significant drift and smaller jumps compared to machine 5 even though they both had the same clock rate. The logs show that machine 5 performed some internal work or sent messages, while machine 1 was constantly receiving messages, leading to a build up of the queue as shown in the message queue plot that resulted in smaller jumps and larger drift.

Docs

We also created some documentation for the project that can be accessed inside the repository, containing all the information about the code.



All this information is described in more detail in our README.md, but it is important to note that this documentation is generated from the docstrings present in the code and that it details every function and class available, as can be seen in the following image and example for the Machine class.