Michael Hla
Aayush Karan

Clock Experiments

System Design:

To simulate the connection of multiple machines with different clock cycles, a P2P socket based network among different processes was established, with each machine randomly generating a clock speed on initialization. Each machine has N + 1 threads: 1 thread that simulates the local machine with the specified clock speed, 1 thread to listen for and accept connections with newly initialized machines, and N-1 threads to listen for messages, where each thread is specified for a specific connection with a different machine. Some form of threading was a clear decision since each virtual machine needed to perform server and client type operations in addition to regular machine operations at the same time. Due to the low number of connections, one thread per connection was chosen, although this may not be the most efficient assignment as the number of machines scales.

Upon initialization, a machine first connects to all other machines in the network (specified in terminal input) and opens msg_listen threads for each machine to listen for messages from each connection. In essence, *each machine acts as a client for machines already in the network and a server for machines that join the network later*. Upon receiving a message, the listening thread adds the message to a thread safe global message queue for that process, which is pulse checked by the machine to see if any new messages have arrived during its execution. Then, a thread to run the machine and a thread to listen for new connections are spawned, where the running_machine thread sends messages to the rest of the network based on a RNG, updates its clock accordingly, and checks its global message queue periodically with pulsed lock acquisitions. This design also allows for the number of machines in the network to be unbounded, so if someone wanted to,

The speed of the machine is isolated to the running_machine thread, where the reading of the message queue and the sending of a clock value or the local state change are treated as single operations, with appropriate sleeps in between to simulate different clock speeds. All clock operations are implemented as specified. For logging, we record first the log type on the .txt file line, followed by the global datetime of the system, the logical clock time, and if needed, the queue length.

Usage

To run an experiment, open three terminals (one for each machine) and input

```
python3 machine.py 1 {listening_port1}
```

to start the first machine, where the user determines an open port on their machine to listen for new connections. For each machine, a port is specified for that machine to listen for new machines to connect to it. Then, on the second terminal, input

python3 machine.py 2 {listening_port2} {first_machine_IP} {listening_port1}

You should see output on the first machine that the second machine connected. Then, on the third terminal, run

python3 machine.py 2 {listening_port3} {first_machine_IP} {listening_port1} {second_machine_IP} {listening_port2}

The system will not start until all three machines are connected (which can be easily tweaked) and the system will allow for more machines to connect as the system runs, provided that when a machine joins the network, it knows the IP and port number of the in network machines.

While the terminal inputs can be unwieldy, it allows for variation of inputs such that if the user wishes to run this experiment across different machines or the user is already using some of their ports, they can precisely specify which port and IP to run on. With more machines, this process could easily be automated by retrieving the inputs from a file or database.

Testing:

We run some tests to confirm accuracy. Test001.py can run with command "**python3 test001.py**" which simply checks if three machines can run concurrently for a period of time and record any messages in their logbook. The next tests become more targeted, and test002 and test003 require running "**python3 test_machine.py 1 8080**" in a separate terminal, and then running "**python3 test002/test003.py**" in another. To refresh logbooks, **after each test case, these external servers must be killed with keyboard interrupt**. Tests 2 and 3 check for edge cases such as no log book messages when less than 3 machines are connected (which we choose to do for sanity sake to uphold specifications), and if the logical clock updating due to a sent message is adequately logged. Finally, test 4 requires both this previous machine running as well as "**python3 test_machine.py 2 8081 TARGET_IP 8080**" on a separate terminal, whence we can finally run "**python3 test004.py**", which tests to see if double sending works and is received and sent and logged properly, with the right logical clock update everywhere as needed. For all tests, broken pipe/connection and malloc errors on the other machines are expected when the test ends, this appears to be inherent to the unit test package. All terminal usage errors are caught before any machine is run.

Remember to check machine IP addresses across the tests!

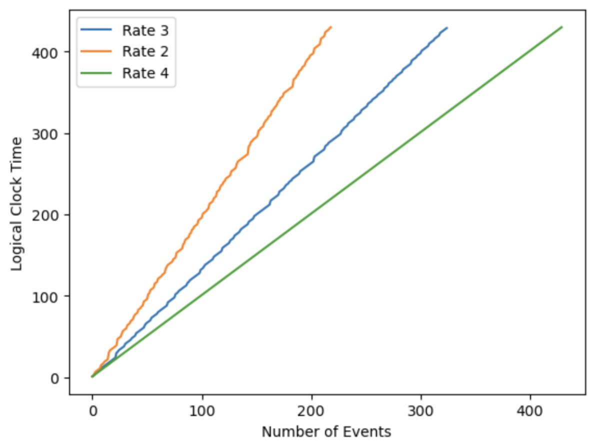
Experimental Design:

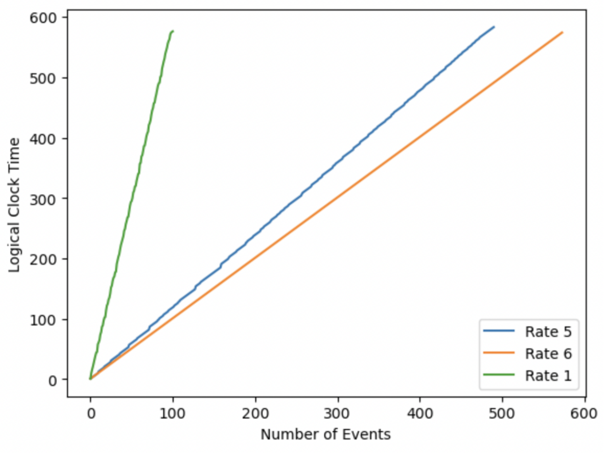
Three virtual machines were initialized in succession, with each machine connecting to the listening port of the previous machines in the network. Our design allows for an arbitrary amount of machines to connect and join the network, but as specified, only three were necessary. Clock speeds were randomly generated from 1-6 ticks per second, and actions were determined by the RNG described in the specification. Machines were run for around 100 seconds in 5 different experiments, with each machine writing to a log file system time, logical clock value, and the length of the message queue. Follow up experiments with varying differences in clock speeds and different internal event probabilities were also run.

Experimental Results:

First observe that the fastest clock has a contiguous rate for its logical clock, with no jumps in updating. Indeed, otherwise, there must be a clock that has a higher event count than the fastest clock from initialization, which would indicate it was faster, or, gets a higher event count after equalizing its timestamp with this clock, which would again indicate a faster accumulation of events.

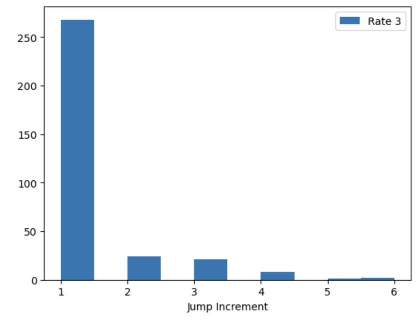
Hence, we expect a graph of event number versus logical clock number to be smooth with consistent +1 increments of the fastest rate clock. Let us observe this graph in some example trial runs:

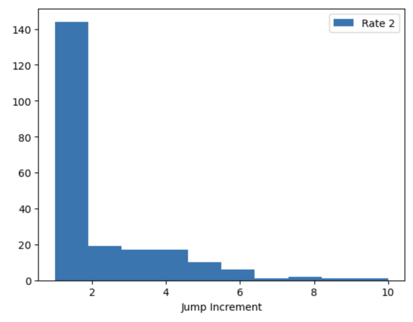


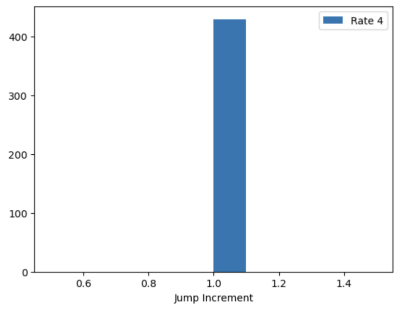


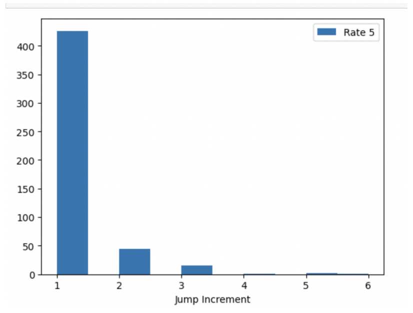
As is evident from these graphs, each clock at rate r will have 100*r events, but the logical time at the end is roughly fixed by the fastest clock, and messaging will lead this logical time to be roughly the same across the machines, hence the different number of events but with the same y-axis height of logical clock time shown in the pictures. The steeper the slope of this line, the fewer events occur per unit of logical clock time, and hence the slower the overall rate is.

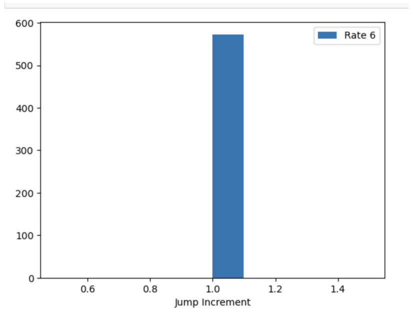
Notice further that the smoothest trajectories are with the fastest rates, indicating effectively no non-+1 jumps in logical clock value. Let us look at the distribution of logical jumps across 3 machines for a few trials (1 and 3) again:

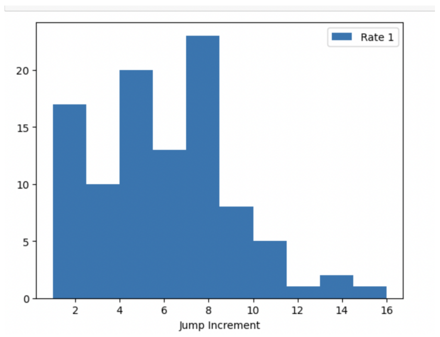








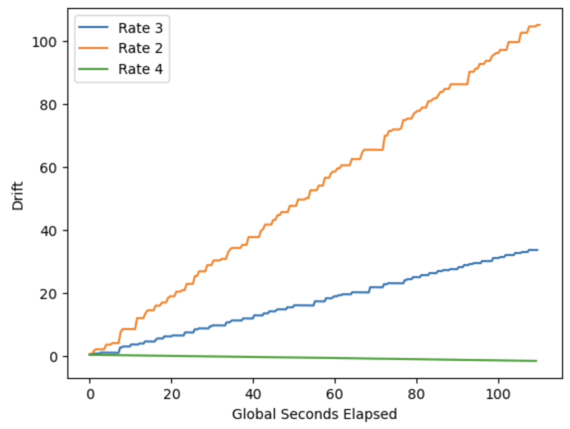


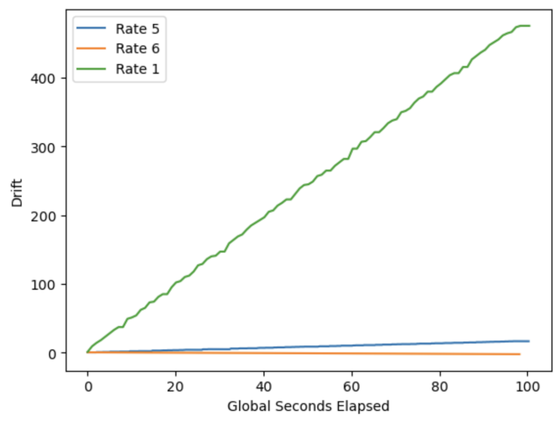


Indeed, these histograms of jump in logical clock values per machine confirm that slower rate machines will experience much more frequent and on average larger size jumps, since the local clock runs slow to the point where it must constantly catch up to faster processing rates in other machines. Another interesting aspect of this data is the apparent exponentially decaying distribution as the jump size increases. This is justified as a message timestamp update follows a kind of Poisson process/distribution, in the simplest case being that for the other machines, there is probability 0.8 of not sending an updated timestamp per cycle, so for larger cycle lengths this follows a probabilistic power law, which is exponential like. In general, for a rate like Rate 1 slower than the other two rates, the updates are similar to two independent Poisson processes superimposed with different rates of timestamp updating events occurring, owing to the different machine clock rates.

Again, what matters is relative clock rate between two machines in determining this distribution, as Rate 3 is to Rate 4 what Rate 5 is to Rate 6 in the bottom trial. The larger the gap, the higher jumps the slower machine will experience in the logical clock, with the rate 6 to rate 1 being the most pronounced and having the highest average and maximum jump stride.

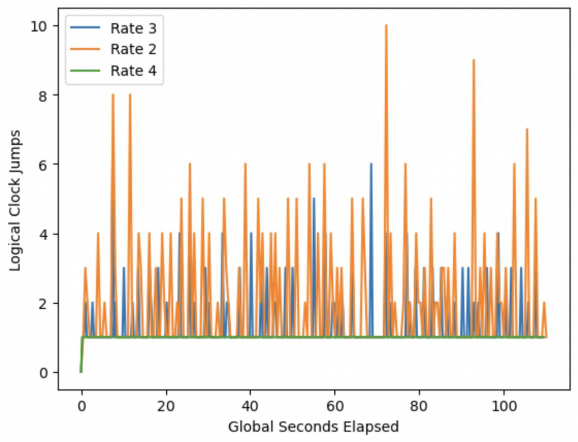
This analysis of relative clock rates defining properties of the system carries over to examining drift. Here, we define drift to be drift = 1/rate * (logical clock time) - (global seconds elapsed since initialization). This is a reasonable measure, as the fastest clock rate will have no drift since the logical clock time is never updated (slightly negative values are due to precision errors). Then the fastest clock rate defines a tempo of consistent events, and for slower machines, failing to tick as quickly means the synchronized clock time leads ahead, and the slower machine is continually readjusting to catch the fastest clock rate. As in the figures below, which graph drift vs. global seconds elapsed, we see that the fastest clock rates have no drift. Meanwhile, the drifts of the slower machines follow a nice linear pattern. Indeed, recall our equation drift = 1/rate * (logical clock time) - (global seconds elapsed since initialization). Now, let r_F denote the quickest rate, and notice logical clock time is (global seconds)*r_F, so drift is equal to (r_F/rate - 1)*(global seconds), depending on the multiplicative factor r_F/rate - 1. For example, if r_F = 6 and rate = 1, then the drift expansion factor is 5, which is confirmed by the right graph below. If r_F = 4 and rate = 3, then the factor is 1/3, which is confirmed by the left graph.



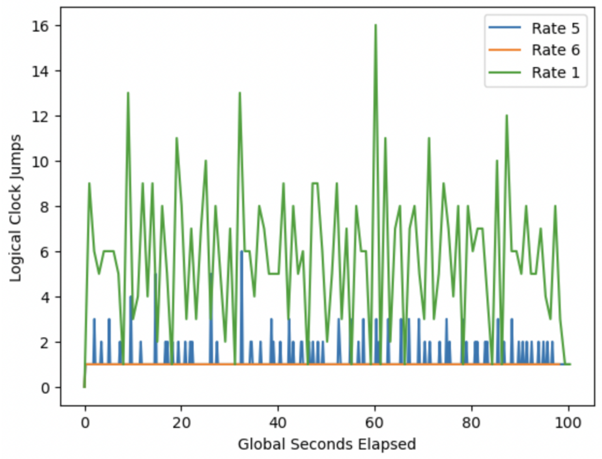


Now, we shall look at the values of logical clock jumps and message queue length over global time.

For jumps over time we have:

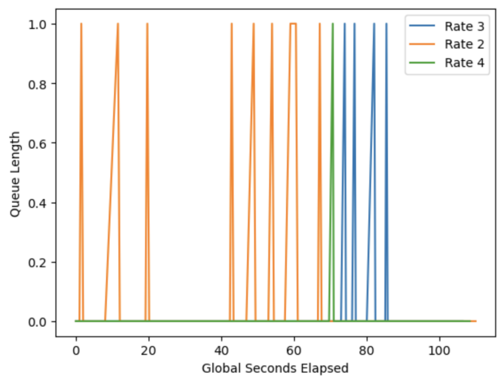


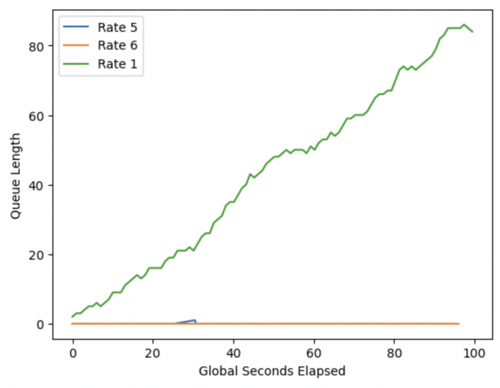
Here, we can



observe that there is no strong relation between global seconds elapsed and the jump experienced by a slower machine in its logical clock. In particular, we can once again turn to the Poisson process for intuition, as such a process is memoryless: the waiting times until a timestamp synchronizing update is sent are independent from previous/future events and are identically distributed.

For queue length over time we have:

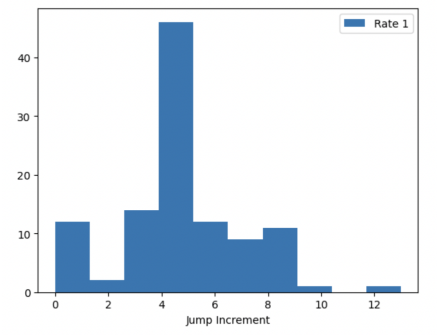




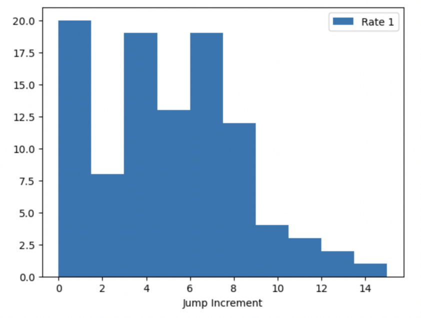
When rates are well stratified, the message queue accumulates on the slowest machine which is receiving messages much more quickly than it can pop them. The queue length then grows roughly linearly with time, with proportionality constant obviously depending on both probability for internal events as well as the rate of removal vs. rate of sending. For less stratified machine times, the queue length tends to be relatively small at all times, since slower machines are still relatively quicker than the gap between say rate 1 and rate 6 machines, so emptying queue occurs faster than others can send. In particular, since the send probability is 0.2 each time, if the rate of a slow machine is greater than that of 0.2 * fast rate, then the message queue will be typically empty. For rates 2, 3, 4, the effective send rate is far below the slowest machine, so the queues are empty. For rates 1, 5, 6, 0.2*6 = 1.2 > 1, so the queue of the machine with rate 1 grows steadily.

Finally, let us consider observing what happens when we run with smaller variation in clock cycles (i.e. reduced by an order of magnitude) or smaller probability of internal events. A smaller variation in clock cycles by order of magnitude would not affect the above results for example, if all rates were scaled down proportionally, since what is relevant is the r_F/rate factor.

A smaller probability of internal events implies that synchronization will occur more frequently by communicating time stamps, so it is expected that the distribution on jumps in logical clock timings will not reach as high values and will be more closely concentrated around +1 jumps: (Changed from 0.7 internal event probability to 0.5):

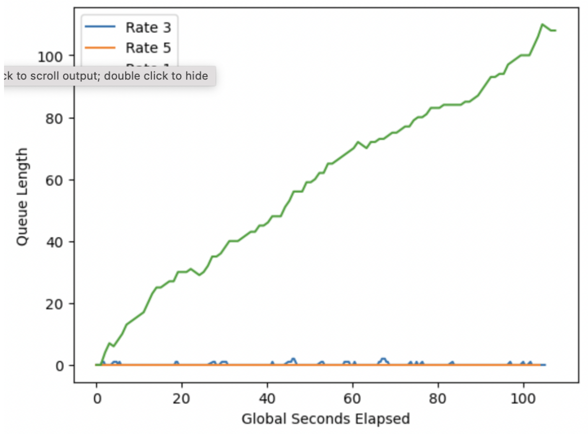


(Occurs when there are rate 5 machines, center more weighted towards the lower side of jumps.)



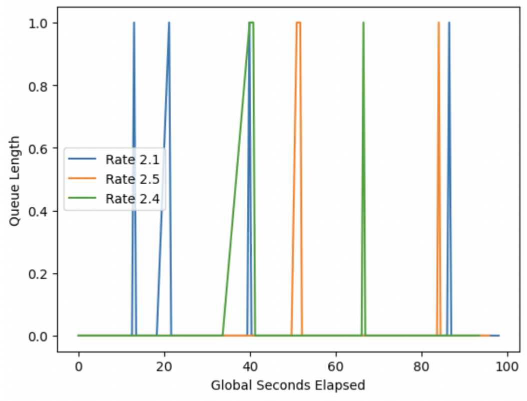
Finally, smaller overall relative gaps between times or higher internal event probability won't affect how time relates to when jumps are experienced by logical clocks. However, smaller internal event probability will make it more likely for message queues to begin to accumulate (see graph), while smaller relative gaps means rates are closer to one another, and hence it is unlikely for one rate to fall below 0.2*the other (see graph).

Smaller internal event probability implies message queues accumulate:



Indeed, 1 < 5*0.5 whereas before 1 = 0.2*5 so the messages would not have accumulated before but do now.

Small relative gaps in clock rates implies message queues less likely to accumulate:


Rate 2.1
Rate 2.5
Rate 2.4
Queue Length
Global Seconds Elapsed