262 Lab Notebook Assignment 2: Time

March 4, 2021

2/27/21

We met to discuss the assignment in order to tentatively establish code structure and division of labor. Discussion of design decisions covered the following questions:

- how to initialize the "agents" to connect to one another and begin their communication run,
- how to structure each of the distinct agent processes,
- how to implement the wire protocol for inter-process communication,
- how to implement logging for each agent.

We explored a few different ideas on the first front. The simplest approach would be to spawn each agent via multithreading or multiprocessing, and to simulate IPC through shared memory. However this seems against the spirit of the assignment. To be distributed, it seems that instead agents should at least communicate via sockets and listen on distinct ports. The program which starts these processes will therefore be a bash script which starts each agent on a distinct port. Then in principle these could be hosted on different machines and the behavior of the agents and their interaction would remain unchanged (albeit with more latency between sending and receipt of messages between agents). If each agent listens on a port, the next question which arises is how to coordinate agents' transition from the "startup" phase of initializing sockets on their various ports to the "active" phase of communicating with one another.

On the second front, we opted that each "agent" will consist of two processes running in parallel; the first will own an object which is responsible for sending and receiving messages from other agents, and the second will perform the other specified tasks–making requests to send and receive messages, "doing work," waiting around, keeping (logical) time, and updating the agent's log.

On the third front, freed of writing our own wire protocol we are exploring the option of using RabbitMQ to manage IPC. We are hopeful that this will maximally simplify/abstract away the actual sending/receipt of messages between processes.

Finally, logging seems separable from the other design decisions here. Each agent will periodically write to a file which records its state and progress. This will presumably be a .txt or .csv–to be decided later.

3/1/21

Logging

We have chosen to implement logging by writing rows to .csv files. This is simple to implement, and easily supports any basic data visualization or analysis which we may choose to do after running experiments. Logs are (initially) named using system time, so that the logs of the various agents for a given run will appear lexicographically adjacent to one another. Log names also include the combination of clock speeds which are present in that run. The log files themselves contain some additional information about the parameters

of the run, in addition to the logging outlined in the assignment; this allows them to serve as a (more) self-contained record of the experimental trial.

As a minor note, we made the design decision to log queue length *before* the incoming message is received, rather than after. An advantage of this choice appears in the figures in our analysis below: in a given round, the queue variable is nonzero if and only if the agent receives a message.

Logical Clock Updates

The agents' logical clocks are impelemnted as described in Lamport's "Time, Clocks, and the Ordering of Events in a Distributed System." Specifically, all agents' clocks begin at 0 and are incremented by 1 for each round of agent action. Agents' actions fall into two (or three) types, and the logical clock is updated differentially depending on type:

- Message Receipt: the agent takes a message off of the incoming message queue and checks the sender's logical clock value. If sender's clock is ahead of agent's, agent sets logical clock value to sender's, *then increments by 1* (and subsequently logs the receipt).
- Sending/Working: otherwise the agent chooses randomly between sending messages and "doing internal work". In this case, *the logical clock is incremented by 1 first*, then the agent sends messages or does work (and logs the activity).

Based on these rules, it's perhaps interesting to predict two phenomena. First, whichever agent(s) with the fastest processors (ticks per second values) should never have their clocks "fast-forwarded" by incoming messages–there is no way in which logical time can "speed up" though any combination of sending and receipt of messages. Second, if an agent's inbox is overflowing, their logical clock updates will also fall behind. If the slower agent receives messages without delay, then its clock will jump forward to meet the faster sending agent's. But if it has fallen behind, its time will only jump forward as far as the time of the faster agent when the stale message was sent.

Agent

The core class representing a virtual machine is the Agent class. The Agent is responsible for setting up the clock cycle, executing commands at the correct frequency, and handling the logic for the program. To run at a specific clock frequency, we decided to use python's sleep function to wait the appropriate amount of time between executions. We found that because of the communication approach we used (described below), the "work" at a given cycle actually took a small but measurable amount of time that added up over the 100s of iterations. To avoid the global time from drifting too much, we therefore improved the accuracy by capturing the system time before work was done and taking this into account when deciding how much time to wait.

The logic for the agent is simply exactly as described in the assignment, with the agent receiving a message if one was available and either working or sending a message if not. For logging, we decided to use a simple csv which the agent appends to. To make everything reproducible we initialize the agent with a controlled seed.

Launching

We wanted it to be easy to start multiple agents at almost exactly the same time. We considered using multiprocessing to handle this, but decided that added unneeded complication. We thought it would be much easier to simple have a bash script that starts the processing in the background, which turned into launch_agents.sh. We wanted the entire process to be reproducible, but we also wanted the agents to not all have the same seed so their clock rates would not all be the same. To accomplish this we defined a global seed which is passed into the bash script, which deterministically sets different values for each agent's

seed. To ease experimenting with different amounts of passive "work," launch_agents.sh also takes a second argument which is the maximum integer for the agent's random action selection process.

Additionally, to ease experimentation, we put in a little effort to make the bash script launch the agents at the same time but wait to exit until they are all finished. Furthermore, we handle SIGINT called with Ctrl+C to kill any remaining agent processes.

Communication

We considered a few options for communication between the agents; we wanted a form of communication that was easy to setup and relatively lightweight but would allow the agents to robustly send messages to each other. We settled on [RabbitMQ](https://www.rabbitmq.com/), which is the backend message broker. Running the system requires first running the broker in a separate process. Once this is running, we use the python client pika to interface with the broker to send and receive messages. RabbitMQ handles communication in a publish/subscribe model, where all communication is to and from queues. We adapt that to messages between agents by assigning each agent a queue based on the agents unique index. To send a message to the agent one sends a message to the corresponding queue, and each agent is continuously listening to events on their queue.

The communication is manifested through the Communicator class in communicator.py. In order to both listen for new messages and send messages we use blocking connections and multiprocessing. Specifically, a child process runs which listens for messages to the broker queue, adding them to a multiprocessing queue. It turned out that querying for the length of a standard multiprocessing.Queue is not implemented on Mac (which is crazy), so we had to make our own Queue class that handled querying for the length properly. To match the specification, Communicator implements a get_message() function which pops any incoming messages off the multiprocessing queue and returns it, or None otherwise.

Initially, we thought we might need to handle edge cases where messages are incoming to agents that don't exist yet. Because of the queue-based communication via the message broker, however, this problem is alleviated by declaring broker queues on both the sending and receiving ends of the communication. This way, if the receiving agent doesn't exist yet, the message is still added to the correct queue and the receiver receives the message as soon as it starts listening. This allowed us to simple start all the agents at roughly the same time without caring for the exact order or needed to verify that all agents were ready.

One last challenge we ran into was that originally, the queue names were exactly the machine indices. Because the broker was persistent and the machines weren't all ending at exactly the same time, often the broker queues were left with lingering messages from previous experiments, which jumped the logical clocks forward significantly. To get around this, we simply added a uuid portion to the queue name, with the uuid being the same for all three agents for a run, coming from the integer second since the unix epoch the agents were launched.

1 Experiments and Results

1.1 First Round of Experiments

For the first round of experiments, we ran our configuration of three intercommunicating agents for one minute. Each agent has a "clock speed" between one and six ticks per second, and each agent is configured to receive a message, should one be queued up. Failing that, each agent sends a message to one of the two other agents with probability 0.1 each, sends a message to both with probability 0.1, and otherwise completes a local task.

We analyze the behavior of this system under various combinations of clock speeds below. We do this by examining the logs directly and additionally plotting the logical clock values and queue lengths for each agent relative to local machine time.

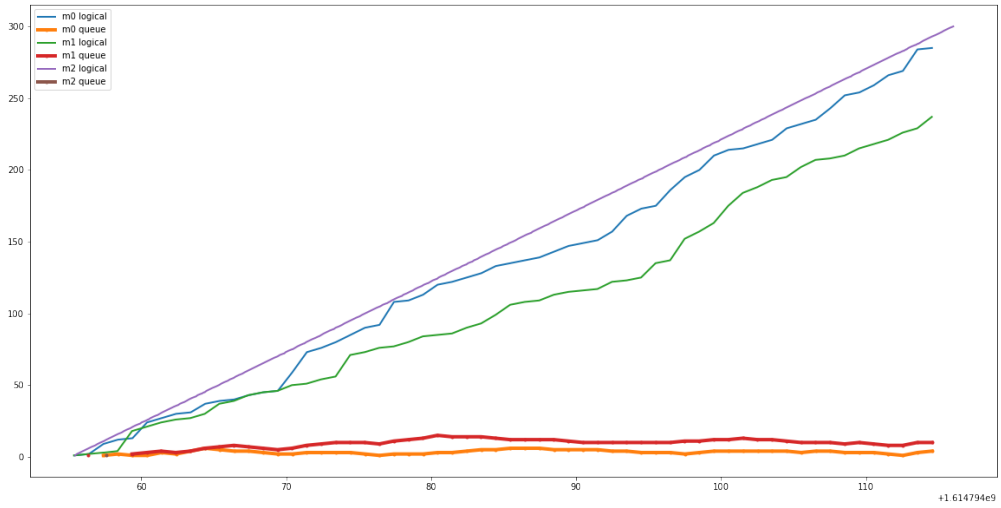

Figure 1: Logical clock and queue length evolution for speeds (1,1,5).

Speeds = (1,1,5)

Figure 1 shows the fastest agent humming along and sending (but never receiving) messages, while the other two do nothing but receive its messages. It stands to reason that one agent running five times faster than the other two is capable of overwhelming both of them and inducing them to do nothing but read their emails, since it have five times as many ticks per second as they do, and in each tick it has probability 0.2 of sending a message to either individual agent. The slow agents' clocks drift behind the fast one because their incoming messages pile up, as evidenced by the continuous presence of the slow agents' queues (red and orange). Since neither gets the chance to send any messages throughout the (vast majority of) the run, we should interpret the first agent's relatively smaller lag (and corresponding shorter message queue) as attributable to the randomness of the fast agent's message recipients.

Speeds = (1,2,4)

Here Figure 2 shows that agent 2 is able to keep abreast of its inbox and keep up with agent 4, while agent 1 is just on the edge of being swamped by agent 1 and 2's incoming messages. Agent 2 even sends a few messages of its own, as evidenced by Agent 4 receiving while agent 1 has messages queued (and so cannot be the sender).

Since its clock progresses only half as fast as 4's, agent 2's clock is closest to 4's precisely when it is consistently receiving messages from 4 but its queue is not backed up.

Speeds = (1,3,6)

As contrasted with the $(1, 1, 5)$ and $(1, 2, 4)$ cases, here Figure 3 shows that the agent with speed 3 is able to keep up with the fast agent 6, while the slow agent 1 is most definitely not. Agent 3's clock is indeed more tightly in lockstep with 6's than is the case in Figure 2, because 6 sends more frequently than 4 and correspondingly 3 receives more frequently than 2. The main difference between these two runs is the slow agent is relatively slower, and indeed agent 1 here is totally unable to keep up, and its queue grows over the course of the run.

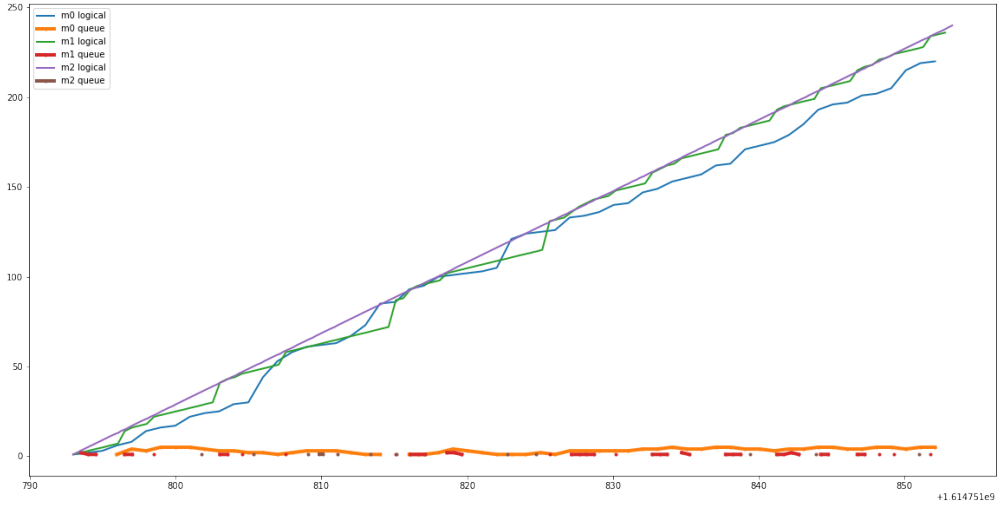

Figure 2: Logical clock and queue length evolution for speeds (1,2,4).

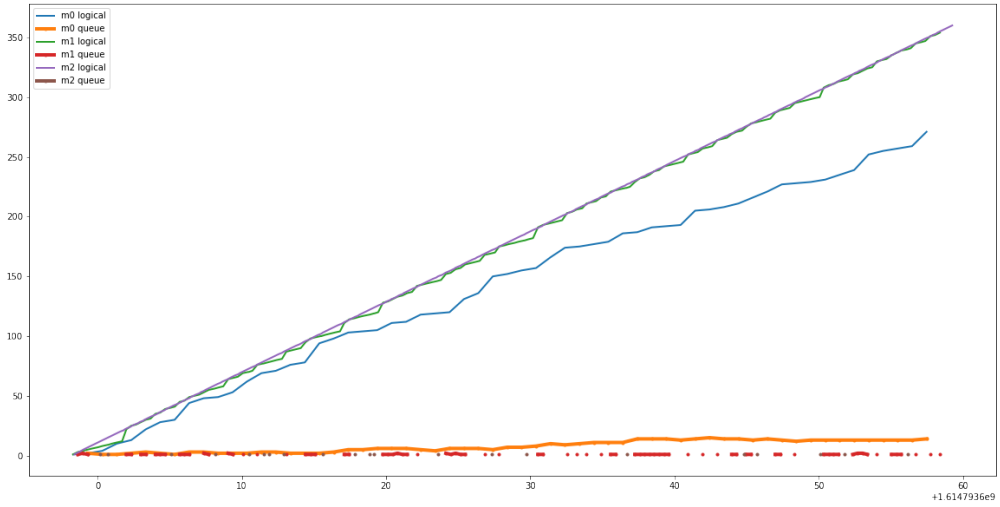

Figure 3: Logical clock and queue length evolution for speeds (1,3,6).

Speeds = (1,5,6)

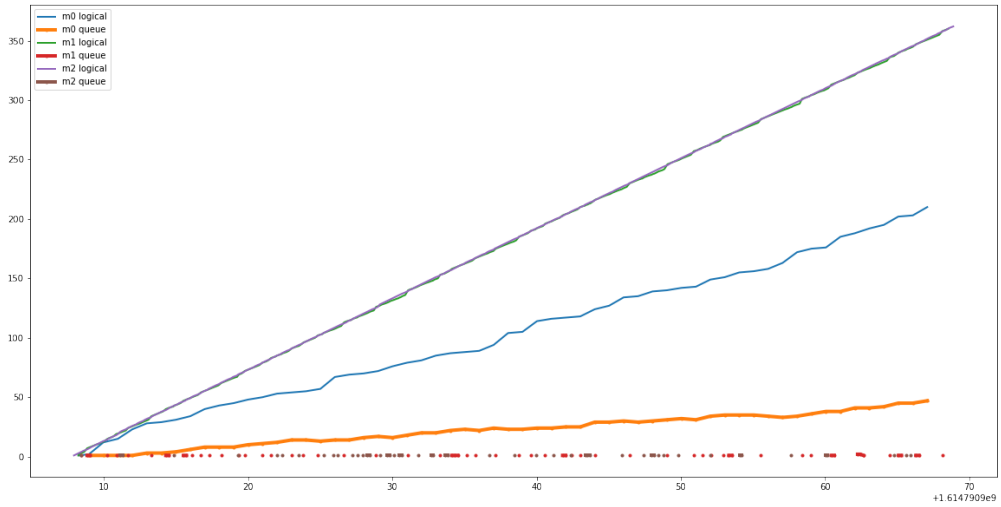

Figure 4: Logical clock and queue length evolution for speeds (1,5,6).

In Figure 4 have two fast agents and a slow agent. The two fast agents 5 and 6 stay very closely in lockstep, sending and receiving messages between themselves with uncongested queues. On the other hand the slower agent 1 falls even more drastically behind than the slow agent in Figure 3.

Speeds = (2,3,5)

As compared to the previous runs, for this combination of speeds Figure 5 suggests that the slower agents 2 and 3 are able to keep apace of the fast agent 5. Their relative speeds seem to predictably correspond to the proportion of the time during which they have a nonempty queue of incoming messages, though both are able to clear their queues.

Speeds = (5,5,5)

Finally, Figure 6 shows perhaps the most boring case: when all agents have the same speed. Here we should expect each one to have an incoming message in at most 2/5 of the rounds, and so no agent should become congested and fall behind. Furthermore, the clocks cannot fall out of sync because the are running at the same rate with respect to the system time!

In this case, it's interesting to think about the feedback loops which may arise between agents, for the following reason: the more messages an agent receives, the less it sends, and so the fewer messages other agents receive, and so the more messages they are free to send. For instance, if there were only two agents each of which sends to the other unless there is a message to receive, we might expect to attain an equilibrium where one agent only sends and the other only receives. Here any effects of this feedback loop should be more subtle, but the density of incoming messages over time for each agent does perhaps look a little "clumpy". This suggests that agents might naturally find themselves taking turns doing a disproportionate amount of sending vs receiving. We will consider this further in a later batch of experiments.

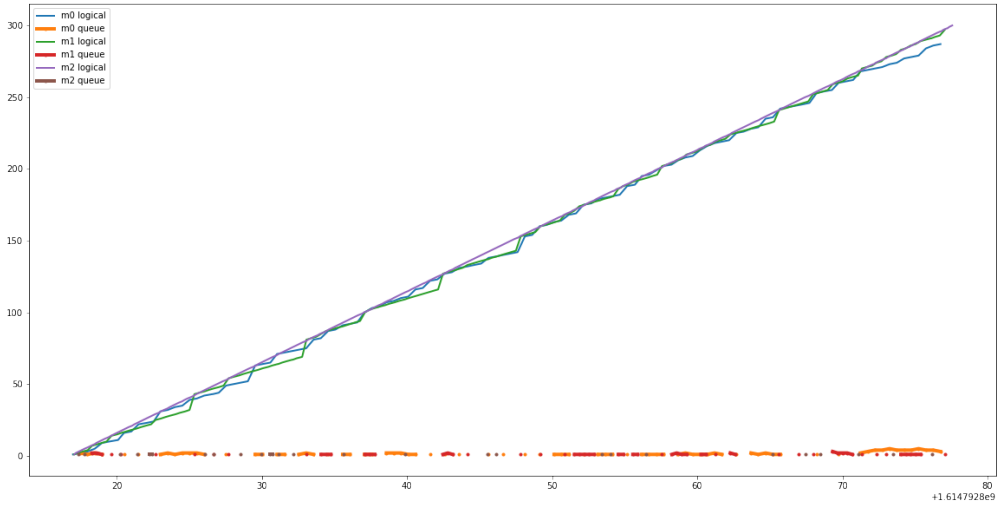

Figure 5: Logical clock and queue length evolution for speeds (2,3,5).

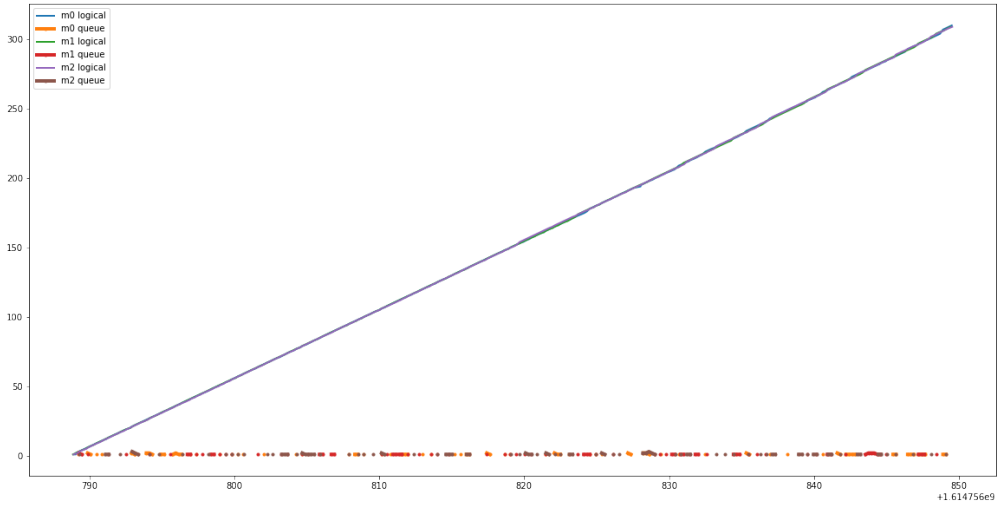

Figure 6: Logical clock and queue length evolution for speeds (5,5,5).

1.2 Increasing Probability of Broadcast

A number of natural questions follow from the above experiments. One such question is: "How does each agent's likelihood of broadcast affect the likelihood that slower machines become overwhelmed with incoming messages?" Or, put differently, "How might changing broadcast likelihood cause logical clocks fall out of synchrony?" Let's refer to the condition of an agents' queue length steadily increasing as "inbox infinity"

As a tentative answer, we hypothesize that increasing the likelihood of broadcast functionally amplifies the difference between agents' clock speeds *from the perspective of message processing*, thereby increasing the likelihood that agent(s) develop "inbox infinity" and subsequent asynchrony. Conversely, decreasing the likelihood of broadcast should decrease the likelihood of inbox infinity.

As a thought experiment, for any fixed number of agents and ratio of clock speeds, as the probability of broadcast approaches 0 we should eventually see every agent able to process its queue–the system will be free of inbox infinity. The other extreme (high-broadcast) is more complex reason about (since agents are broadcasting indiscriminately and) the more messages an agent receives, the fewer it is able to send. For this reason we might expect a sort of local negative feedback effect in the level of network traffic. However if we make the approximation that there is global probability of an agent receiving a message in some fixed interval of (system) time, then agents with higher clock speeds will (of course) be better equipped to handle this background level of incoming messages, and the degree to which a unit increase in this background level decreases their frequency of outgoing messages will be proportional to the inverse of their clock speed: their broadcasts per unit time will decrease by less. Therefore as all agents' broadcast likelihood increases, we might reasonably expect agents with faster clock speeds to remain able to handle the incoming traffic and continue sending at levels for which slower clocked agents reach inbox infinity (if in fact this higher level of effective broadcast can be attained without the slower agents).

As a first step at testing this hypothesis, let us consider some of the combinations of clock speeds tested above, with likelihood of broadcast increased or decreased (and all other parameters unchanged). We may increase the likelihood of broadcast (for all agents jointly) by increasing the parameter maxrand_int, which (as per assignment instructions) is set to 10 in the experiments above.

Speeds = (5,3,2), maxrand_int = 4

With the maxrand_int parameter set to 4, the probability of a worker with an empty queue choosing to work instead of sending one or more messages is only 1/4, down from 7/10 for the default configuration. Figure 7 shows the logical clock values and queue lengths in this case. In Figure 5, showing the same configuration of clock speeds but maxrand_int=4, the set of clock speeds was such that no machine was swamped with messages and no logical clock drift occured. Here, with maxrand_int = 4, work happens much less frequently than sending, which leads to a lower threshold for starting the positive feedback loop. In Figure 7 this can be seen as the logical clock for m2 lags behind the other two logical clocks, and the queue length for m2 begins to grow.

Speeds = (1,5,6), maxrand_int = 20

We test this hypothesis in the other direction by taking an instance where inbox infinity clearly emerges and significantly decreasing the broadcast probability.

Figure 8 shows the results of re-running the experimental configuration of Figure 4 with maxrand_int = 20 rather than 10. This increases the probability that an agent with no incoming messages will perform an internal task (rather than send a message) from 7/10 up to 17/20. The probability that an agent with no incoming message will broadcast is therefore decreased by a factor of two.

As hypothesized, decreasing the likelihood of sending a message allowed the slowest agent m0 to 'catch up' with m1 and m2, fully clearing its queue on two occasions over the course of the run. This is in stark contrast with Figure 4, where it fell steadily behind m1 and m2.

From the perspective of maintaining global logical time, it is interesting to note that while decreasing message frequency has allowed m0 to catch up with the others, so that its clock is not experiencing drift in

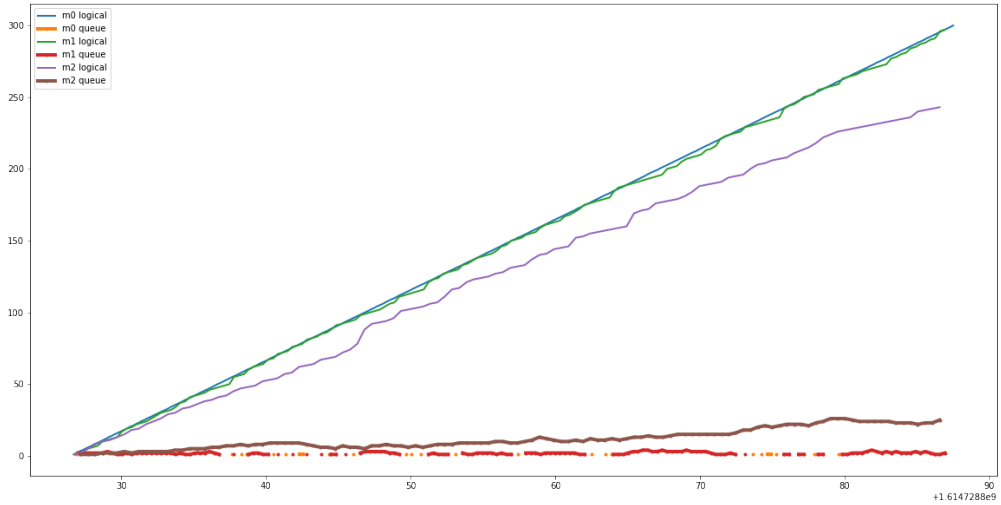

Figure 7: Logical clock and queue length evolution for speeds (5,3,2) with maxrand_int = 4.

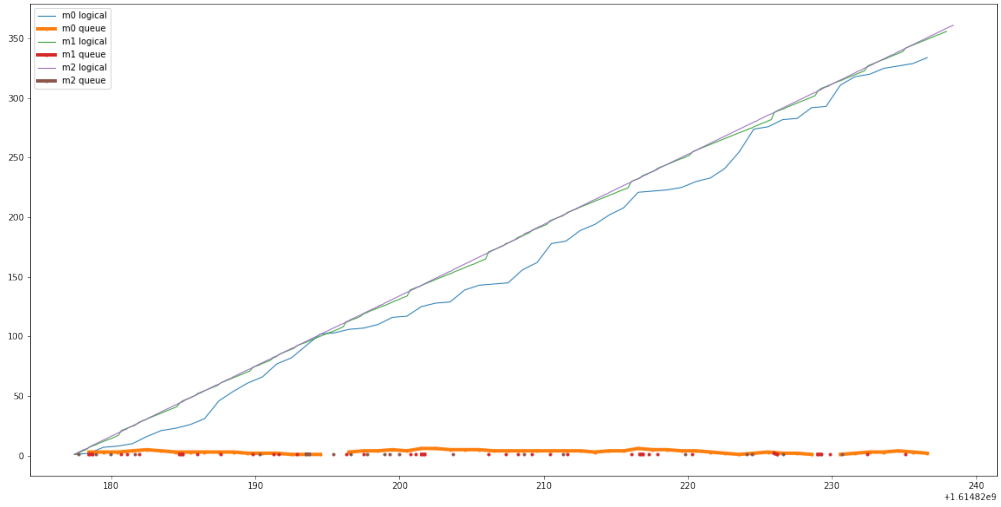

Figure 8: Logical clock and queue length evolution for speeds (1,5,6) with maxrand_int = 20.

the limit, this synchrony comes at a cost to the synchrony between m1 and m2. Whereas in Figure 4 their logical clocks proceed very nearly in lockstep, here their logical clocks are marginally more loosely bound, since m2 does not send a message to m1 as frequently and its clock speed is only 5/6 as fast.

These experiments in modifying the message sending probability have tentatively confirmed the hypothesis which we put forward above. This hypothesis concerned the inbox infinity phenomenon under increased probability of sending and different clock speeds. It leaves open the question of how changing the probability of outgoing messages affects the likelihood of inbox infinity/logical clock asynchrony when *all clock speeds are the same.*

1.3 Inbox Infinity and Agents with Equal Clock Speeds

The question we now ask is, "Do agents with (nearly) equal-speed clocks lose synchrony when the probability of broadcast becomes sufficiently large?" The answer is not a priori clear. This is because, even if agent clocks have exactly equal speeds, there is a negative feedback effect in terms of messages sent and received: the more messages are flying around the system, the more time agents spend reading messages, and the fewer messages they send. To make this more clear, it will be helpful to introduce a little bit of notation.

Here all agents' clocks are presumed to run at the same rate. We will additionally preserve the symmetry of the default configuration of the system that each agent is equally likely to send a message to any other agent.

Definition 1.1. For an agent a, let r_a denote the probability that a receives a message in a given round. Let s_a^f denote the expected number of messages that a sends in a given round, provided that a has no incoming messages. Finally, let s_a denote the expected number of messages that it sends in a given round.

Let's indulge in some informal calculations. In equilibrium (which may or may not exist/be stable), with no agents at inbox infinity, we should expect that for all agents the probability that there is a message waiting equals the expected number of messages sent. This is because agents are identical, and if agents are not overwhelmed, all messages sent are received. Therefore we would expect that $r = s$. But agents only send messages when they do not have one to receive. Therefore we find that

$$s = (1 - r)s^f,$$

where $(1 - r)$ is the probability that an agent is free to send messages. Combining this with $r = s$ we may derive that in steady state, $s = (1 - s)s^f$ and so

$$r = s = \frac{s^f}{1 + s^f}.$$

Finally, agents may only continue to send messages if $r < 1$.

Next consider the event that an agent a's queue develops a backlog. This is quite likely to happen at some point over the course of the run. At this point it does not emit messages, but continues to receive messages from the unstuck agents. If there are n of these other "unstuck" agents, and they are in equilibrium with one another, then we expect this stuck agent's queue to continue to grow if $r_a > 1$. If these other n agents are in equilibrium, then each is receiving from the $n - 1$ others; on the other hand, a (who is stuck) is receiving from all n of them. Therefore $r_a = \frac{n}{n-1}r$. What are the conditions under which $r < 1$ but $r_a > 1$? This occurs when $r = s \in (\frac{n-1}{n}, 1)$. Solving for s^f, we find $s^f \in (n - 1, \infty)$. Recalling that n were the unstuck agents, this suggests that the interval should be $s^f \in (N - 2, \infty)$, where N is the total number of agents. If the expected number of messages sent per free round approaches this range, we expect that it should become reasonably likely for one or more agents to reach inbox infinity.

How does this measure up in practice? The experiment shown in Figure 6 above has equal clocks and $n = 3$, and seems far from inbox infinity. What is its value of s^f? Based on the parameters of the initial configuration, each agent sends a message to each of the other two with probability 0.2. Therefore $s^f = 0.4$ for this run–far from the threshold of $N - 2 = 1$ at which we would expect to start seeing problems.

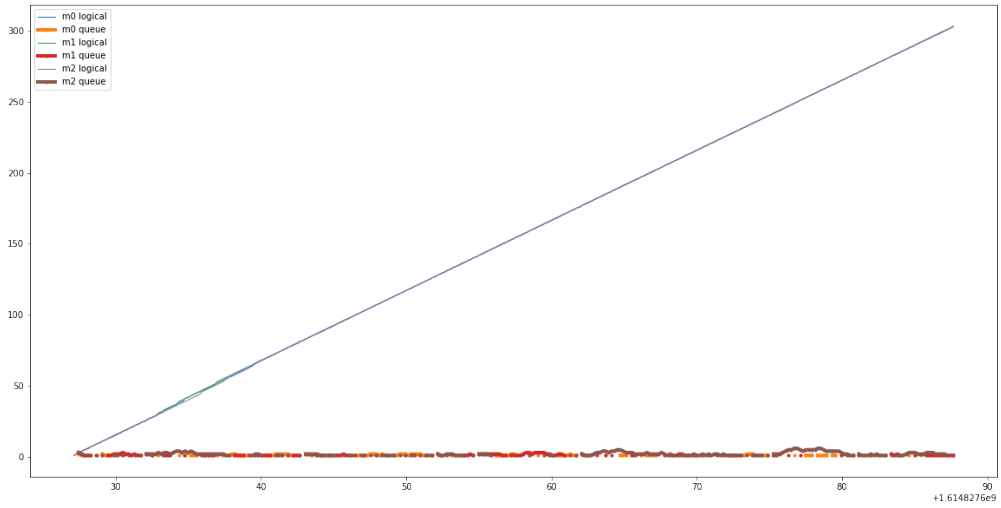

Figure 9: Logical clock and queue length evolution for speeds (5,5,5) with maxrand_int = 3 (yielding $s^f = 1.25$).

Choosing maxrand_int = 3, on the other hand, gives $s^f = 5/4 > 1$. The results of this run are shown in Figure 9. This graph may appear initially unpromising for our theory, since the logical clocks remain firmly in sync throughout the run. But of course they do! They are all running at the exact same speed, and update their logical clocks exactly once per action cycle. This is also apparent in Figure 6, though there it is misattributable to the fact that queues are uncongested.

In this run, by contrast, we see queue congestion as our model predicted! Agents take turns with congested queues for long segments of the run.

Therefore even despite the negative feedback forces present in our send/receive model, and when agents are unable to "inbox infinity" one another due to their superior clock speeds, when communication becomes sufficiently frequent with respect to the number of agents the dynamics of the system will naturally pick one or more scapegoats and flood their incoming message queue–in fact, the calculations above can be further informally extended to heuristically predict how many equal-clock-speed agents will avoid inbox infinity for a given value of s^f.

1.4 Reflections

Overall, this simple setup allowed us to see the features/conveniences of the logical clock design, and the failure modes. One thing that is quite striking is the simple combination of sending the local logical clock values and the max update rule is enough to achieve some form of synchronization among machines that run at different speeds and have no access to an accurate global clock. When the machines run at similar paces and don't send too many messages, no one machine is bogged down by receiving all the time and the local clocks do indeed roughly line up. For some applications it's easy to imagine this level of synchronization is not sufficient, but it's still very cool that there is any level of rough synchronization given the simplicity.

At the same time, given the specific problem setup with highly variable clock speeds, there is a clear failure mode that leads to a positive feedback loop, "inbox infinity," and subsequent asynchrony. As we've seen, when the machine speeds are different the slow machines spend all of their cycles receiving, leaving the faster machines to receive even less frequently and thus send more frequently. Even relatively modest changes

in send frequency can be sufficient to bog down the slower machines. Furthermore, we have demonstrated that this is even possible when all machines' speeds are equal! Clearly, this is undesirable.

To avoid "inbox infinity" and asynchrony, one could perhaps model the dynamics of the system and determine a range of values of message transmission probability which yield stable dynamics. We have made an informal first attempt to do so here in the case when all machines have equal speed. With a "safe zone" of transmission probabilities in hand, one could then operate and hope that even in the unlikely event that one machine's queue becomes backlogged it will clear in relatively short order.