Engineering Notebook #3

Kamryn Ohly + Grace Li

Link to Github: https://github.com/grxxce/logical-clocks

Link to Live Notebook:
https://docs.google.com/document/d/1DfPXOKEPJCDz2rVHp8GDPNpF8m5IQFF7w8mi_KFpdW8/edit?usp=sharing

Introduction
To begin this design challenge, we used our previous design exercise as a starting point. Since we needed to build a simulation of three communicating processes, we believed that it would be easiest and most beneficial to use our working chat application and remove unnecessary features. Thus, our architecture largely builds on our previous notebook, which you are welcome to reference here: https://docs.google.com/document/d/1kC1dU5g-jA3qQlwvC7pxrMSXKUD7kRxc6Kewmv5-JYc/edit?tab=t.0.

In our notebook, you will find our brief remarks on our approach to this design challenge, a few of the design decisions that we made, and our analyses and conclusions of varying simulations. We hope that you enjoy reading and testing out our code!

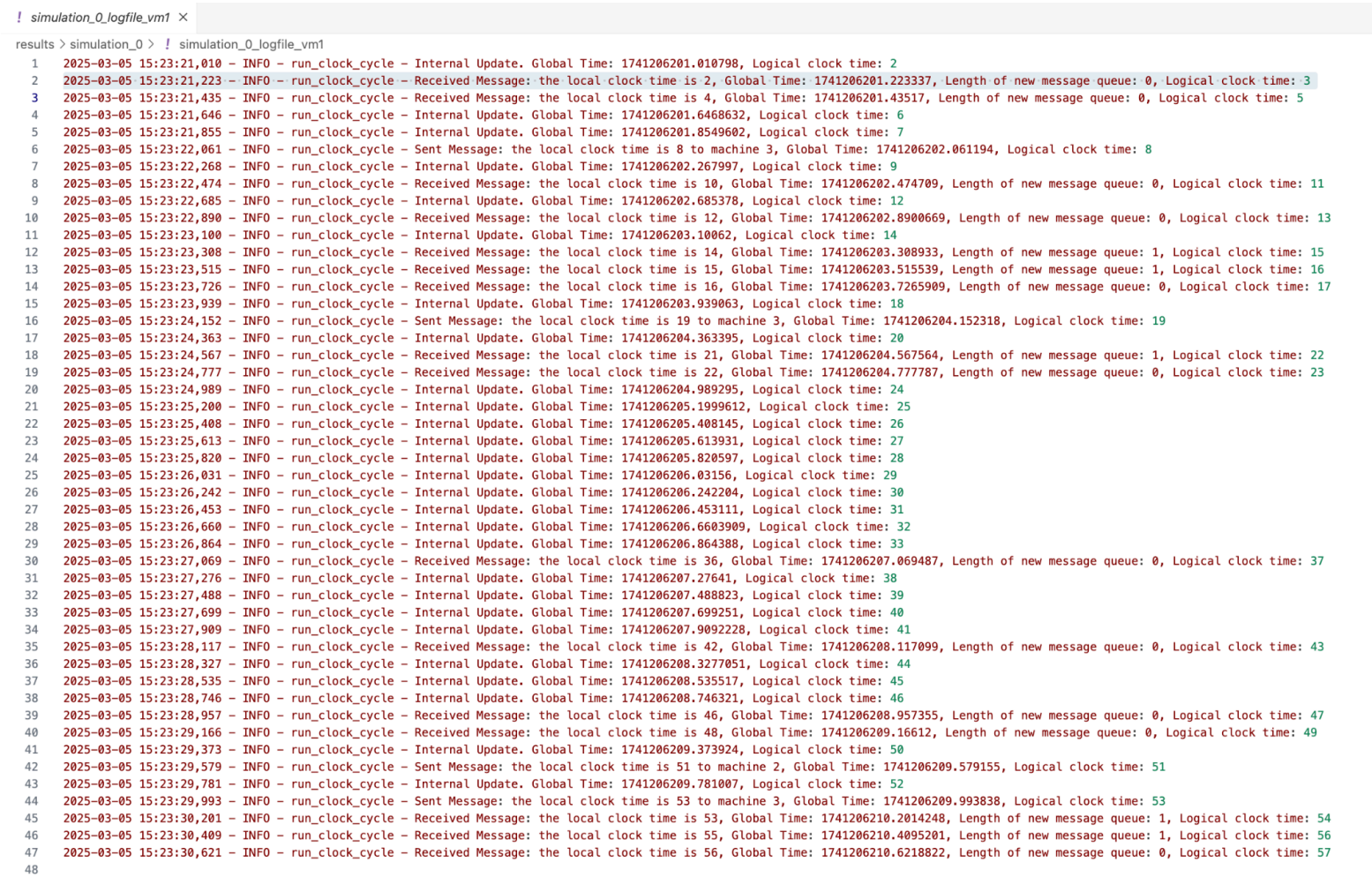
Approach & Architecture
We chose to stay with our previous design challenge foundation, as we felt confident in its architecture and reliability. We also knew that it could be easily adapted into allowing multiple processes to communicate with each other. We started by forking the repository and removing unnecessary components, such as the user authentication information, the UI, and features related to deleting messages and accounts. We used the gRPC approach and kept our protocols related to sending and monitoring messages. With this, we did not make many changes to the Server-side of our project besides deleting unused functionalities. Most of our work architecturally took place in the Client-side of our simulation, where we edited our Client class to become one to represent an individual process running. We started by representing the 3 clients and providing each with a clock-speed on initialization that would be randomly generated between 1 and 6 instructions per second. We then repurposed our message queue to become a network queue that would hold onto a client's pending messages.

With the creation of the randomized clock speed, we had each process independently keep its own logical clock. The way that we handle sending and receiving messages while updating the logical clock takes place in the `run_clock_cycle` function in main.py of our Client. When we receive a message from another client, we check the local logical clock of the other client, and if our logical clock is behind, then we jump to their clock. We then add one to the logical clock to symbolize our action. If we are performing an action that is not receiving a message, then we update our clock as usual. With this

approach, we are able to sequence events across our processes, yet we also can see interesting behaviors, especially when there is a vast difference between the clock speeds of the fastest and slowest process.

```python
def run_clock_cycle(self):
    '''
    (1) Update local clock
    '''
    print("running clock now")
    while self.running:
        self._handle_get_inbox()

        # If there are messages in the queue:
        if self.message_q:
            print("detected a message")
            message = self.message_q.pop(0)
            match = re.search(r"local clock time is (?P<local_clock>\d+)", message.message.message)
            if match:
                local_clock_time = match.group('local_clock')
                self.logical_clock = max(self.logical_clock, int(local_clock_time)) + 1

            self.logger.info(f"Received Message: {message.message.message}, Global Time: {time.time()}, Length of new message queue: {len(self.message_q)}, Logical clock time: {self.logical_clock}")

        # If no messages in queue, probabilistically take another action.
        else:
            print("Did not detect message. Creating an event.")
            self.logical_clock += 1
            # Default upper range is 10
            event = random.randint(1, self.event_probability_upper_range)
            match event:
                case 1:
                    message =  f"the local clock time is {self.logical_clock}"
                    self._handle_send_message(self.other_vms[0], message)
                    self.logger.info(f"Sent Message: {message} to machine {self.other_vms[0]}, Global Time: {time.time()}, Logical clock time: {self.logical_clock}")
                case 2:
                    message =  f"the local clock time is {self.logical_clock}"
                    self._handle_send_message(self.other_vms[1], message)
                    self.logger.info(f"Sent Message: {message} to machine {self.other_vms[1]}, Global Time: {time.time()}, Logical clock time: {self.logical_clock}")
                case 3:
                    message =  f"the local clock time is {self.logical_clock}"
                    self._handle_send_message(self.other_vms[0], message)
                    self._handle_send_message(self.other_vms[1], message)
                    self.logger.info(f"Sent Message: {message} to machine {self.other_vms[0]} and {self.other_vms[1]}, Global Time: {time.time()}, Logical clock time: {self.logical_clock}")
                case _:
                    self.logger.info(f"Internal Update. Global Time: {time.time()}, Logical clock time: {self.logical_clock}")

        time.sleep(self.sleep_time)
```

Throughout the events handled by the processes, one of the most critical aspects of our approach was the choice to add robust logging of the system time and logical clock timings, as well as actions and messages where applicable. We also constantly log the state of the message queue to ensure that we would have enough information to analyze later on, as well as to assist in possible debugging. These log files are saved in a temporary folder called `/logs` on the same hierarchical level as the Client and Server repositories for easier convenience. Below, you will find an example of one of our log files:



simulation_0_logfile_vm1

results > simulation_0 > simulation_0_logfile_vm1

```
2025-03-05 15:23:21,010 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206201.010798, Logical clock time: 2
2025-03-05 15:23:21,223 - INFO - run_clock_cycle - Received Message: the local clock time is 2, Global Time: 1741206201.223337, Length of new message queue: 0, Logical clock time: 3
2025-03-05 15:23:21,435 - INFO - run_clock_cycle - Received Message: the local clock time is 4, Global Time: 1741206201.43517, Length of new message queue: 0, Logical clock time: 5
2025-03-05 15:23:21,646 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206201.6468632, Logical clock time: 6
2025-03-05 15:23:21,855 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206201.8549602, Logical clock time: 7
2025-03-05 15:23:22,061 - INFO - run_clock_cycle - Sent Message: the local clock time is 8 to machine 3, Global Time: 1741206202.061194, Logical clock time: 8
2025-03-05 15:23:22,268 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206202.267997, Logical clock time: 9
2025-03-05 15:23:22,474 - INFO - run_clock_cycle - Received Message: the local clock time is 10, Global Time: 1741206202.474709, Length of new message queue: 0, Logical clock time: 11
2025-03-05 15:23:22,685 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206202.685378, Logical clock time: 12
2025-03-05 15:23:22,890 - INFO - run_clock_cycle - Received Message: the local clock time is 12, Global Time: 1741206202.8900669, Length of new message queue: 0, Logical clock time: 13
2025-03-05 15:23:23,100 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206203.10062, Logical clock time: 14
2025-03-05 15:23:23,308 - INFO - run_clock_cycle - Received Message: the local clock time is 14, Global Time: 1741206203.308933, Length of new message queue: 1, Logical clock time: 15
2025-03-05 15:23:23,515 - INFO - run_clock_cycle - Received Message: the local clock time is 15, Global Time: 1741206203.515539, Length of new message queue: 1, Logical clock time: 16
2025-03-05 15:23:23,726 - INFO - run_clock_cycle - Received Message: the local clock time is 16, Global Time: 1741206203.7265909, Length of new message queue: 0, Logical clock time: 17
2025-03-05 15:23:23,939 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206203.939063, Logical clock time: 18
2025-03-05 15:23:24,152 - INFO - run_clock_cycle - Sent Message: the local clock time is 19 to machine 3, Global Time: 1741206204.152318, Logical clock time: 19
2025-03-05 15:23:24,363 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206204.363395, Logical clock time: 20
2025-03-05 15:23:24,567 - INFO - run_clock_cycle - Received Message: the local clock time is 21, Global Time: 1741206204.567564, Length of new message queue: 1, Logical clock time: 22
2025-03-05 15:23:24,777 - INFO - run_clock_cycle - Received Message: the local clock time is 22, Global Time: 1741206204.777787, Length of new message queue: 0, Logical clock time: 23
2025-03-05 15:23:24,989 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206204.989295, Logical clock time: 24
2025-03-05 15:23:25,200 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206205.1999612, Logical clock time: 25
2025-03-05 15:23:25,408 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206205.408145, Logical clock time: 26
2025-03-05 15:23:25,613 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206205.613931, Logical clock time: 27
2025-03-05 15:23:25,820 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206205.820597, Logical clock time: 28
2025-03-05 15:23:26,031 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206206.03156, Logical clock time: 29
2025-03-05 15:23:26,242 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206206.242204, Logical clock time: 30
2025-03-05 15:23:26,453 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206206.453111, Logical clock time: 31
2025-03-05 15:23:26,660 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206206.6603909, Logical clock time: 32
2025-03-05 15:23:26,864 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206206.864388, Logical clock time: 33
2025-03-05 15:23:27,069 - INFO - run_clock_cycle - Received Message: the local clock time is 36, Global Time: 1741206207.069487, Length of new message queue: 0, Logical clock time: 37
2025-03-05 15:23:27,276 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206207.27641, Logical clock time: 38
2025-03-05 15:23:27,488 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206207.488823, Logical clock time: 39
2025-03-05 15:23:27,699 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206207.699251, Logical clock time: 40
2025-03-05 15:23:27,909 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206207.9092228, Logical clock time: 41
2025-03-05 15:23:28,117 - INFO - run_clock_cycle - Received Message: the local clock time is 42, Global Time: 1741206208.117099, Length of new message queue: 0, Logical clock time: 43
2025-03-05 15:23:28,327 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206208.3277051, Logical clock time: 44
2025-03-05 15:23:28,535 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206208.535517, Logical clock time: 45
2025-03-05 15:23:28,746 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206208.746321, Logical clock time: 46
2025-03-05 15:23:28,957 - INFO - run_clock_cycle - Received Message: the local clock time is 46, Global Time: 1741206208.957355, Length of new message queue: 0, Logical clock time: 47
2025-03-05 15:23:29,166 - INFO - run_clock_cycle - Received Message: the local clock time is 48, Global Time: 1741206209.16612, Length of new message queue: 0, Logical clock time: 49
2025-03-05 15:23:29,373 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206209.373924, Logical clock time: 50
2025-03-05 15:23:29,579 - INFO - run_clock_cycle - Sent Message: the local clock time is 51 to machine 2, Global Time: 1741206209.579155, Logical clock time: 51
2025-03-05 15:23:29,781 - INFO - run_clock_cycle - Internal Update. Global Time: 1741206209.781007, Logical clock time: 52
2025-03-05 15:23:29,993 - INFO - run_clock_cycle - Sent Message: the local clock time is 53 to machine 3, Global Time: 1741206209.993838, Logical clock time: 53
2025-03-05 15:23:30,201 - INFO - run_clock_cycle - Received Message: the local clock time is 53, Global Time: 1741206210.2014248, Length of new message queue: 1, Logical clock time: 54
2025-03-05 15:23:30,409 - INFO - run_clock_cycle - Received Message: the local clock time is 55, Global Time: 1741206210.4095201, Length of new message queue: 1, Logical clock time: 56
2025-03-05 15:23:30,621 - INFO - run_clock_cycle - Received Message: the local clock time is 56, Global Time: 1741206210.6218822, Length of new message queue: 0, Logical clock time: 57
```

After creating the communication architecture, proper logging, and required elements like the logical clock, we created our simulation through our `simulations.py` file. To ensure that we can start clients at the same time and end them at the same time, we created this file to spin off our separate processes. This file starts by launching the server before launching the three processes independently. The code allows for the user to input the duration in seconds of how long the simulation should last, as well as how many iterations of the simulation should take place. To meet the specifications, our main simulations last for 90 seconds and run 5 iterations. We wanted to create this simulation file separate from both the client and server primarily to allow for the client and server to be more modular in nature. We also created it out of convenience, as opening 4 tabs in iTerm2 and trying to start them at the same time is both challenging and quite time consuming! It also proved to be very useful in allowing us to test different variations of our simulations without requiring substantial changes to the code. One last component of the simulation file is that it also organizes the log data into a `results` folder after each simulation to ensure that all data is properly saved and easily accessible.

After creating the simulation controller and testing its reliability, we created a python program that parsed, analyzed, and visualized our findings in `analysis.py.` These results and analysis can be found below in more detail. This file aims to loop over each iteration of the simulation, use regex pattern matching to extract data from log files, and piece back together information about the jumps, gaps, and sequencing of events that occurred. This also includes analyzing the size of message queues and the drifting that can occur between processes of different clock speeds.

Testing

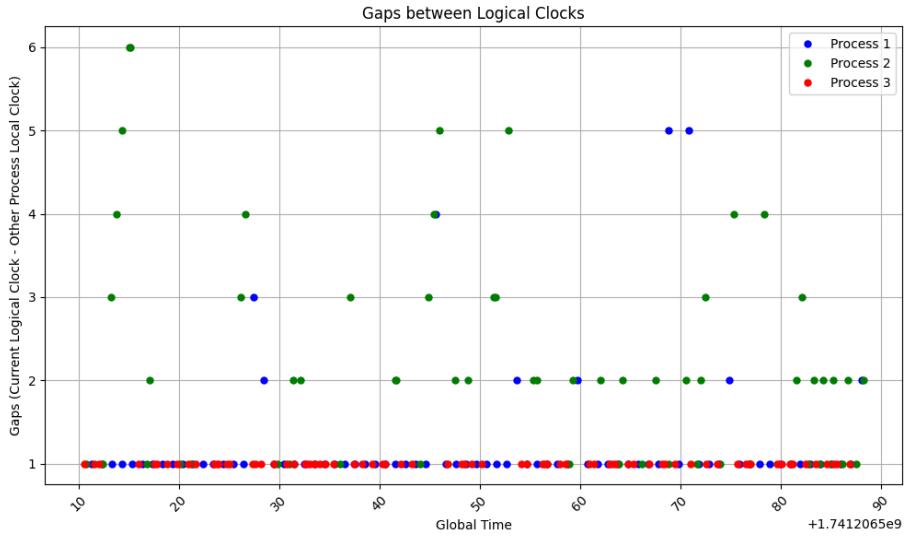
In order to test our files, we wrote unit tests with pytest to ensure full client and server functionality. Specifically, on the client side, we verified that the logical clock starts correctly, correctly picks the max between its own and the incoming message's logical clock time, and increments the count by one. We ensured that the messaging queue's execution was not impacted by the machine's logical clock, and we wrote test statements to ensure that the logging of the events was done correctly. We made temporary directories for log file testing to ensure that they would not interfere with our actual logging. We also made sure that messages were never sent if the queue was not empty. On the server-side, we implemented all of the relevant tests from the previous assignment, including (but not limited to) testing correct message sending between clients, monitoring for new clients, and streaming back the message array. We implemented this response as a stream since it returns a list of messages of unknown length. We implemented 17 tests in total for this system.

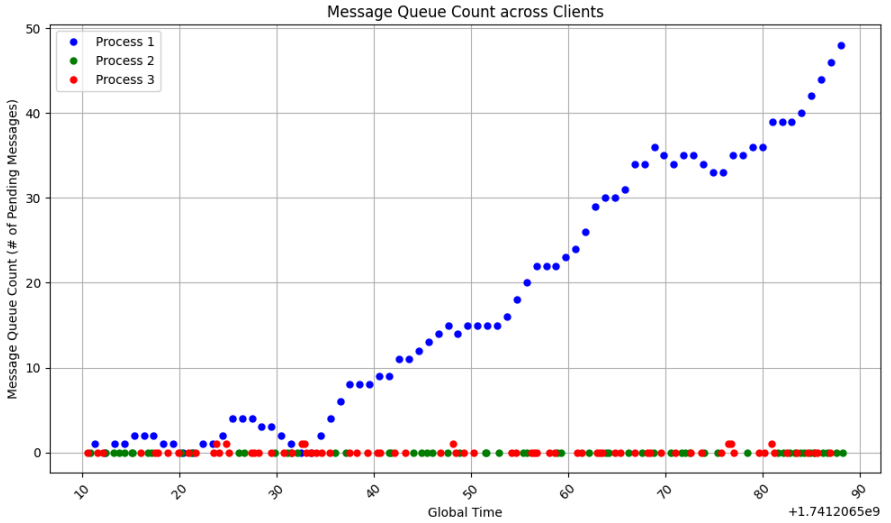
Our Analysis - Original Simulation:

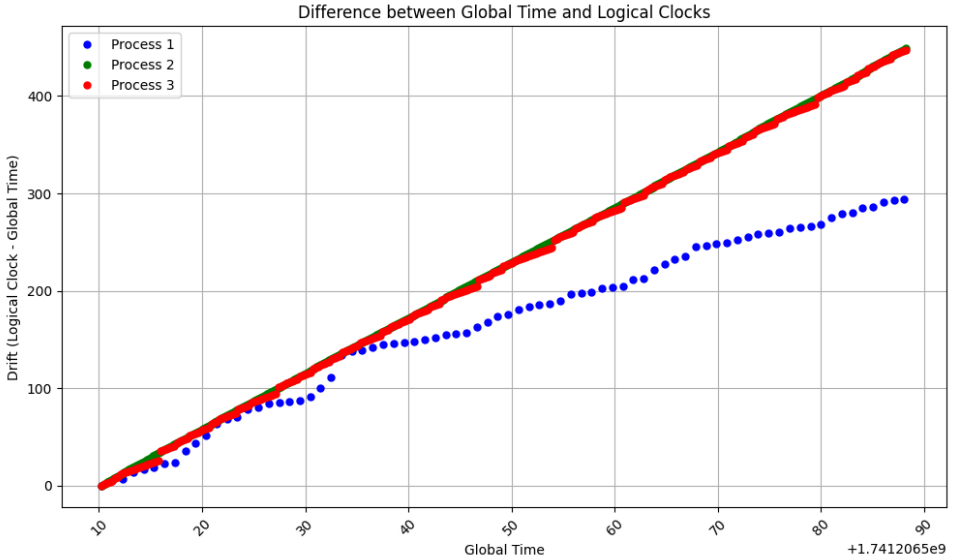
To begin our analysis, we started with **a few constants** for our first set of simulations:

- We have 3 processes running, and we perform the simulation 5 times with 90 second interval durations.
- Each process is randomly initialized with a clock speed that can run anywhere between **1 instruction per second to 6 instructions per second.**
- When a process is able to perform an instruction, it will always prioritize popping a message off of the queue first. If the queue is empty, then the following things can happen (assume that we are Process 1 and that these are generalizable):
    - There is a 1/10 probability that the process will send a message to Process 2
    - There is a 1/10 probability that the process will send a message to Process 3
    - There is a 1/10 probability that the process will send a message to both Process 2 and Process 3
    - If none of these events occur, an internal event will occur.

Simulation #1







Findings & Observations

Identifying speed & drift between processes

From our original simulation, we can clearly see how each process's unique clock cycle speeds led to different behaviors. In particular, in this example, by analyzing the logical clock cycles (the bottom plot), we can see that Process 2 and 3 are faster processes, due to their thick, almost continuous line and steeper slope. Every logging event is represented by a point. The thickness of the line indicates that over time (specifically over system time), there were more data points available to plot, highlighting that the process was generating more events, external or internal, per second than the other processes. This also provides reasoning for why Process 1's plot appears to be individual data points instead of a thicker line, as Process 1 completed fewer events per second and logged less information in the same period of time. Through this observation of the difference in the reported number of events and the speed of these events, we can understand more information about the behavior of the system as a whole. It also showcases the immediate drift of each process's logical clocks from each other, as each process ends at a very different point despite beginning and concluding at the same time.

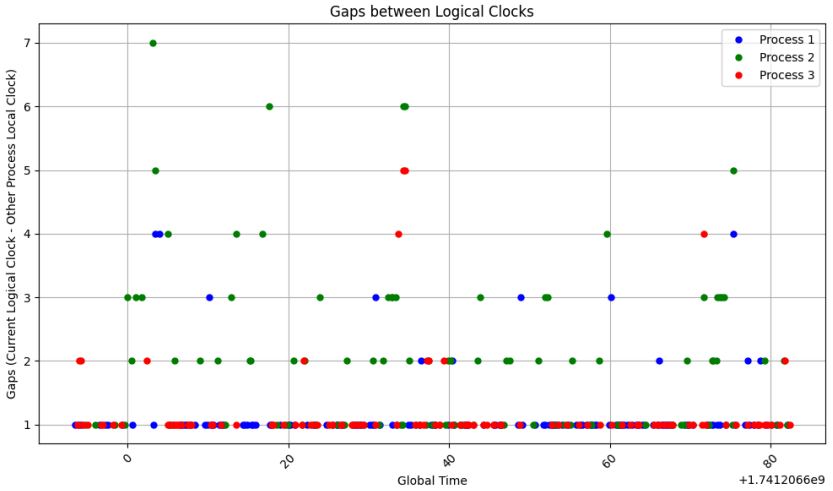
Now, our most important observation: **Paying attention to the behavior of the slowest process and how it interacts as a whole is critical to maintaining an accurate understanding of the state of a system.**

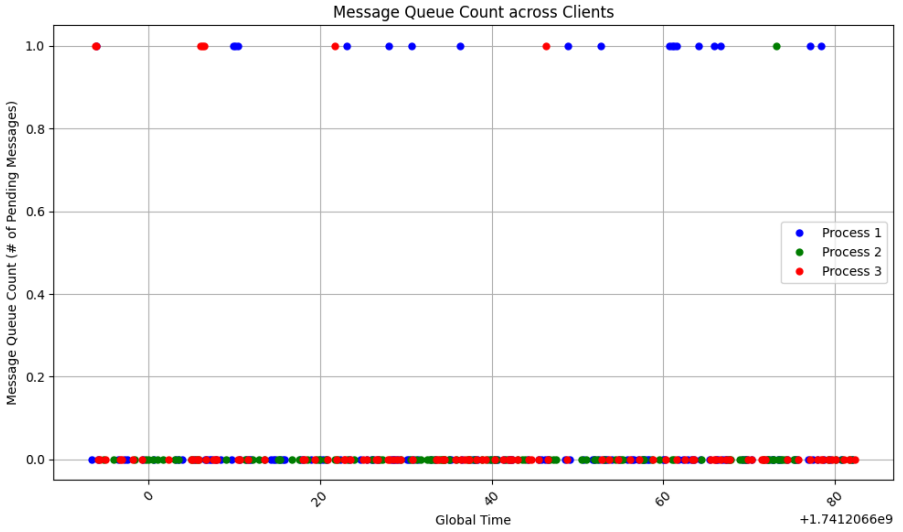
It is clear to see that the slower process queues can become overwhelmed with information, yet we can also see how this negatively affects the understanding of the overall distributed system.
One clear observation is that differences in clock cycles can lead to an overwhelm of messages in the queues of slower processes, which brings down the entire system's state of understanding regarding time. We can see from the plot in the top right that Process 1 (the slowest process) is unable to keep up with the messages being shared from Process 2 and Process 3, leading to a constant battle of Process 1 to clear its queue, which is unsuccessful. While Process 2 and Process 3 can easily clear their queue or maintain an empty queue, Process 1 finds itself entirely focused on that event. This is not a surprising result, yet it is more important to consider what this means for the rest of the system. **This also leads to Process 1 being unable to send updates or messages to Process 2 and 3, which prevents Process 2 and Process 3 from having an accurate understanding of the logical clock of Process 1.**

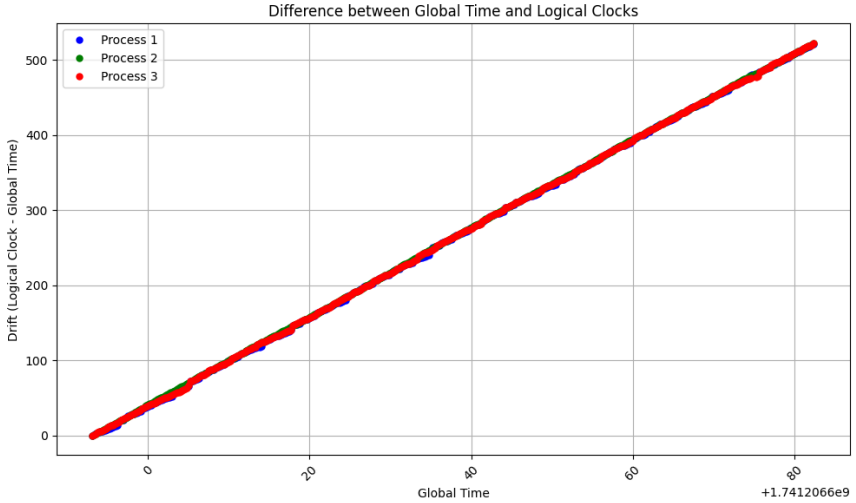
We can see in our plot of the gaps between the logical clocks that a peculiar trend emerges. Process 1 can document both the gaps between its logical clock with Process 2 and Process 3 respectively, yet Process 2 and Process 3 can only identify the drifting between their respective logical clocks. Why can we not identify Process 1's logical clock, you may ask? It took us a moment of reflection to figure it out before realizing that Process 1 is dedicating 100% of its resources to clearing its queue, meaning that it can't send updates on its state to the other processes. As a result, the distributed system loses the ability to understand its overall state from Process 2 and 3's perspectives.

Originally, while designing this system, we expected the slowest process to be a bit of a "lost cause," yet this realization and finding emphasize that a system needs to be able to support and handle all of its clients' and their sense of timing in order to have a clear picture of the system state. While the drift between logical clocks can be determined in the first few data points between the three processes, once the drift has reached a certain point, such as how Process 1 can no longer send updates, it becomes very difficult to recover without digging a deeper hole.

Simulation #2



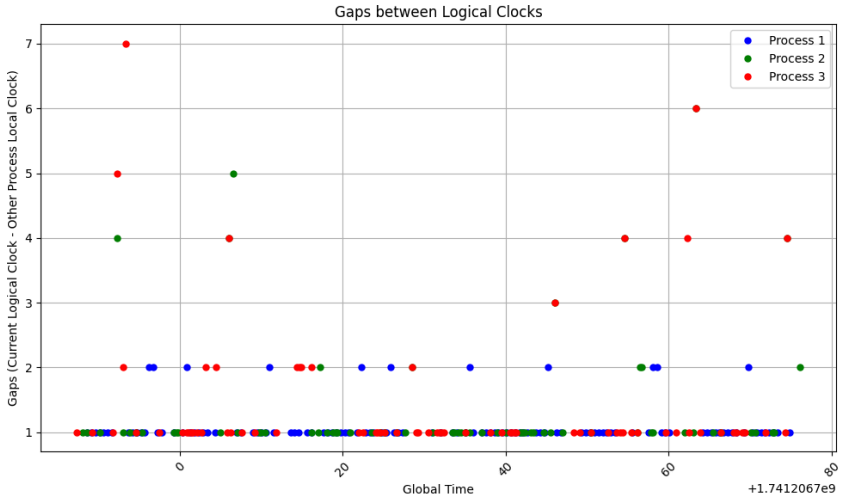


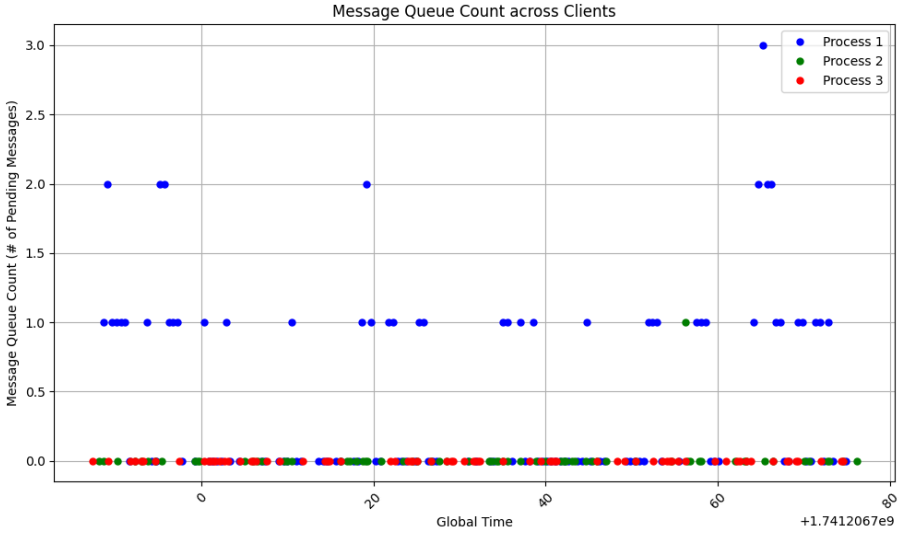


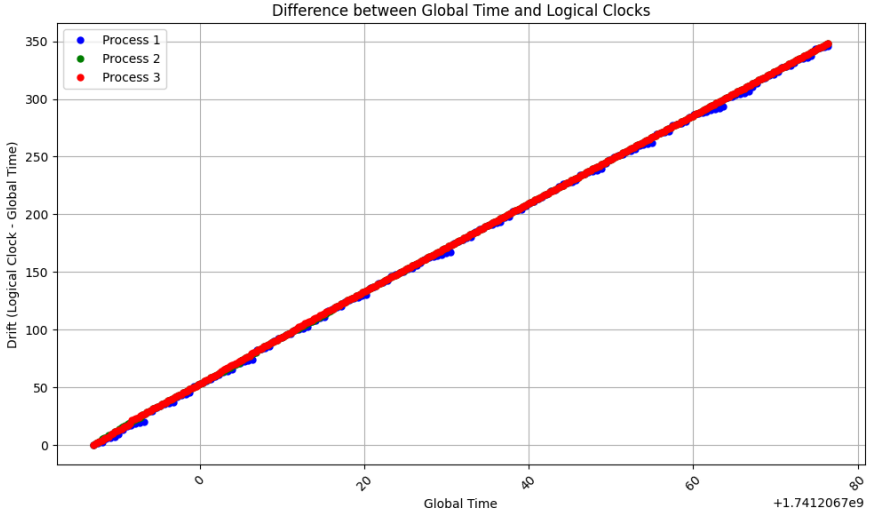
Findings & Observations

One interesting observation to note, in addition to the above simulation, is that when processes are operating at very similar clock cycles, we minimize the amount of drifting that occurs between processes, and our queue remains relatively empty. We saw similar observations in the next few simulations, so we will not provide additional explanation, as they support our initial findings.

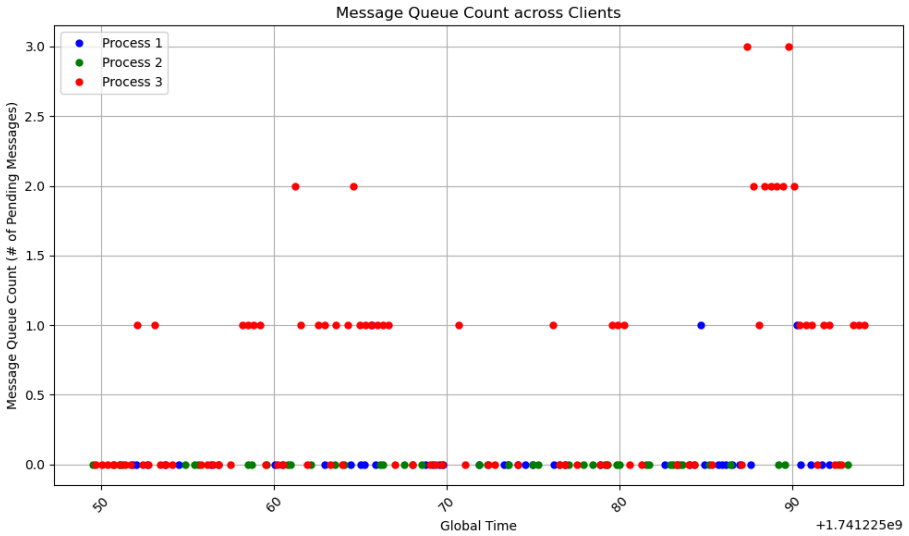
Simulation #3

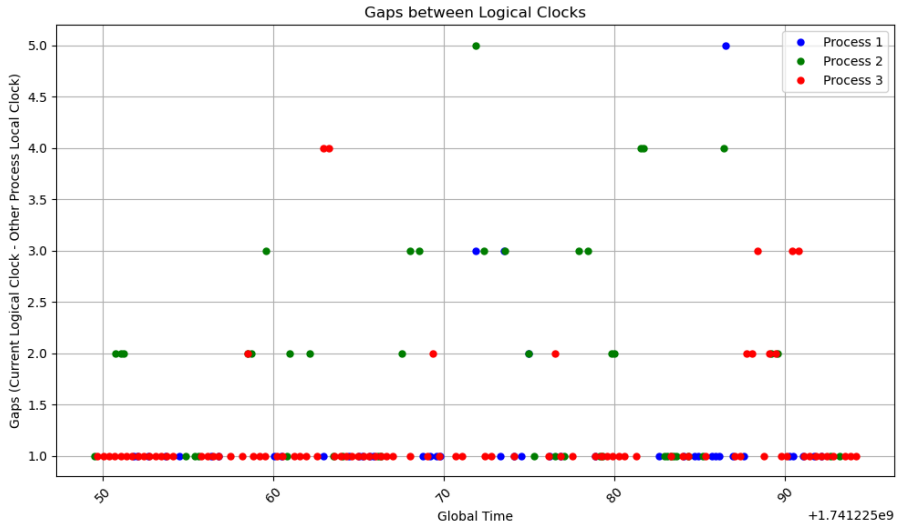

Gaps between Logical Clocks
Process 1
Process 2
Process 3
Gaps (Current Logical Clock - Other Process Local Clock)
Global Time
+1.7412067e9

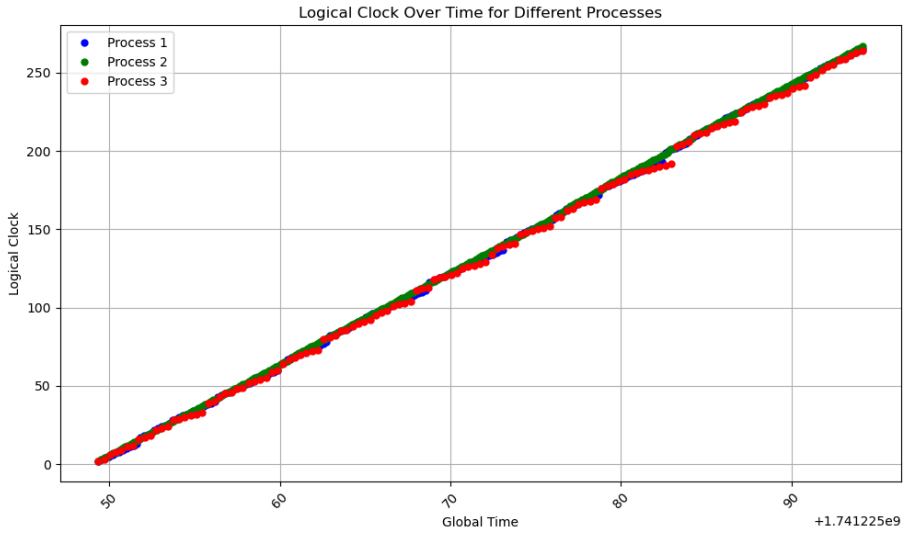

Message Queue Count across Clients
Process 1
Process 2
Process 3
Message Queue Count (# of Pending Messages)
Global Time
+1.7412067e9


Difference between Global Time and Logical Clocks
Process 1
Process 2
Process 3
Drift (Logical Clock - Global Time)
Global Time
+1.7412067e9

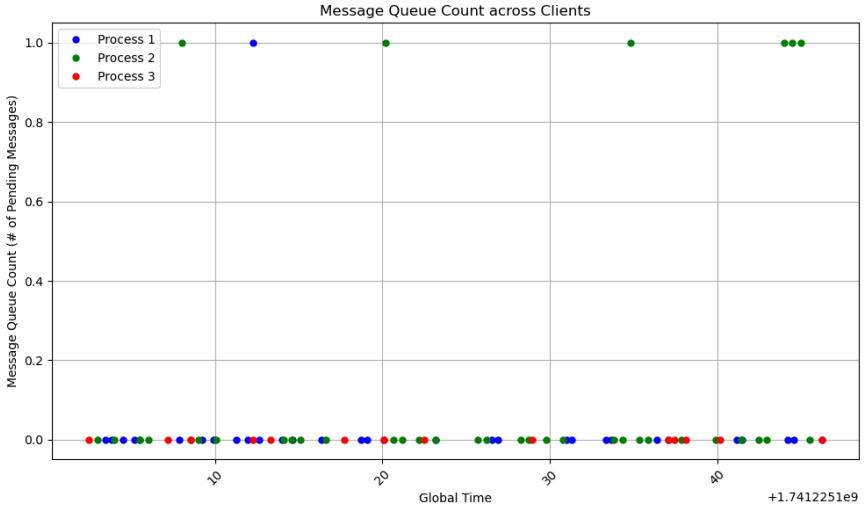
Simulation #4

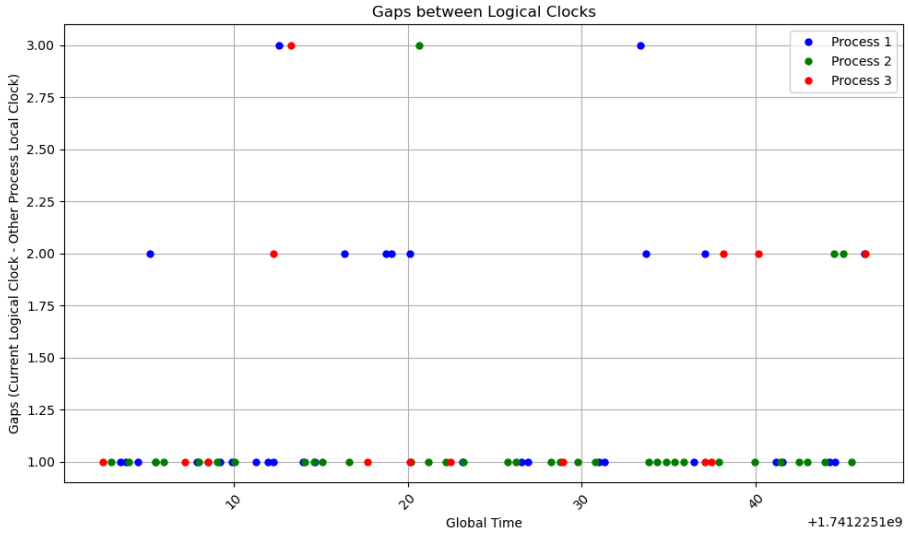


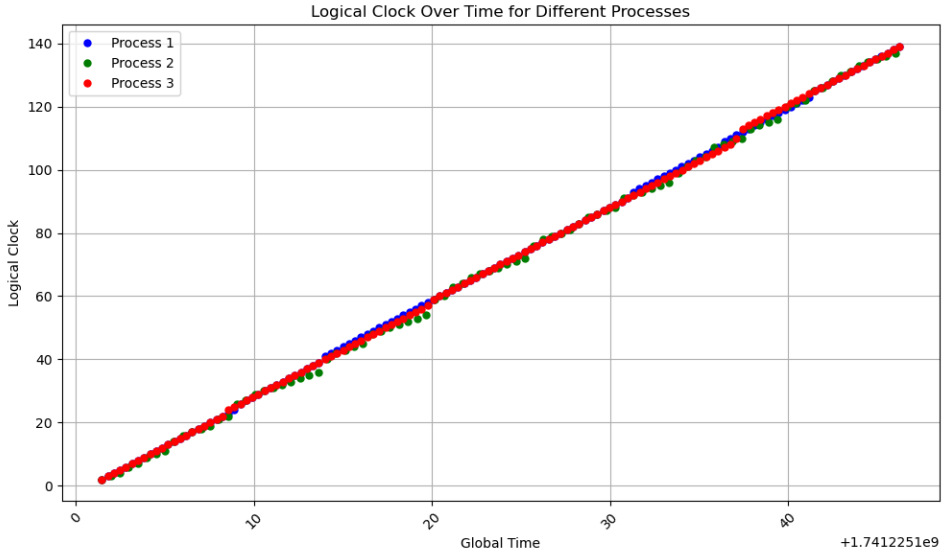




Simulation #5





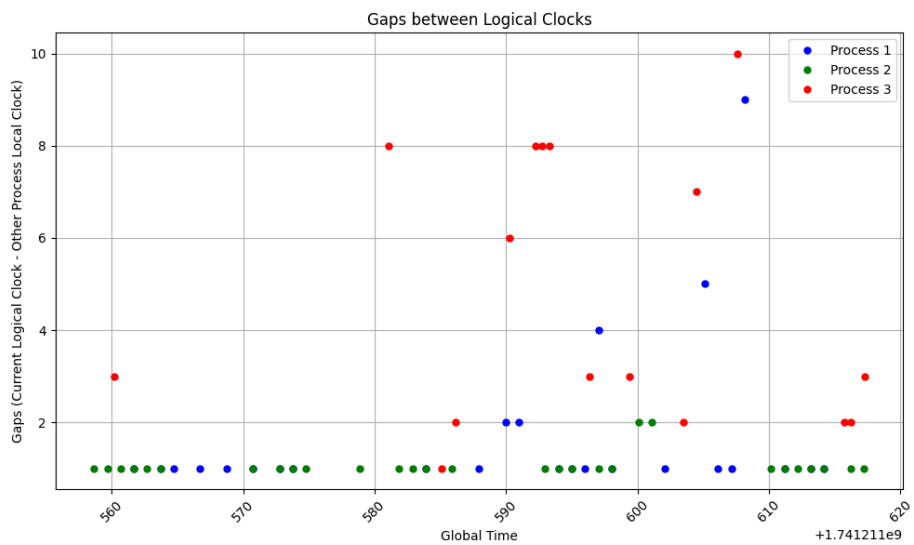


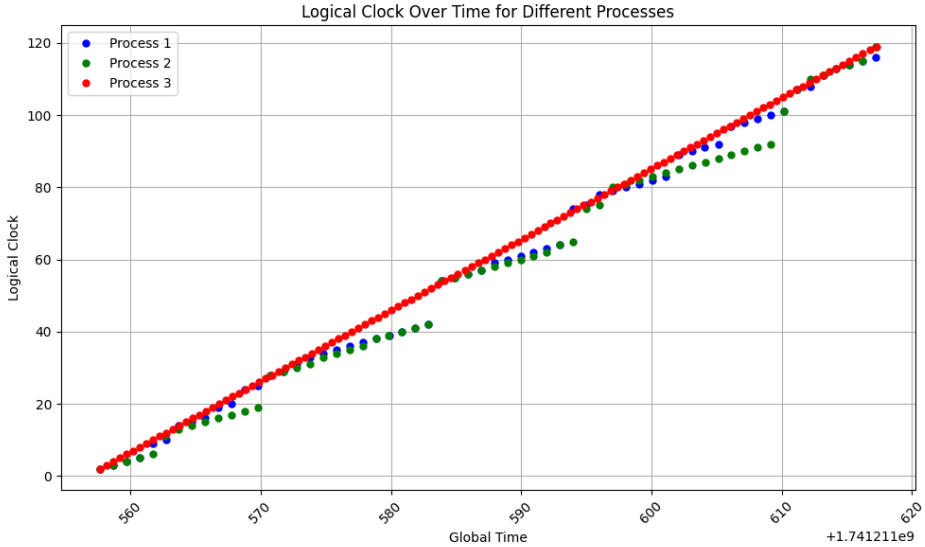
Our Analysis - Variation #1 (Smaller Variation in Clock Cycles):

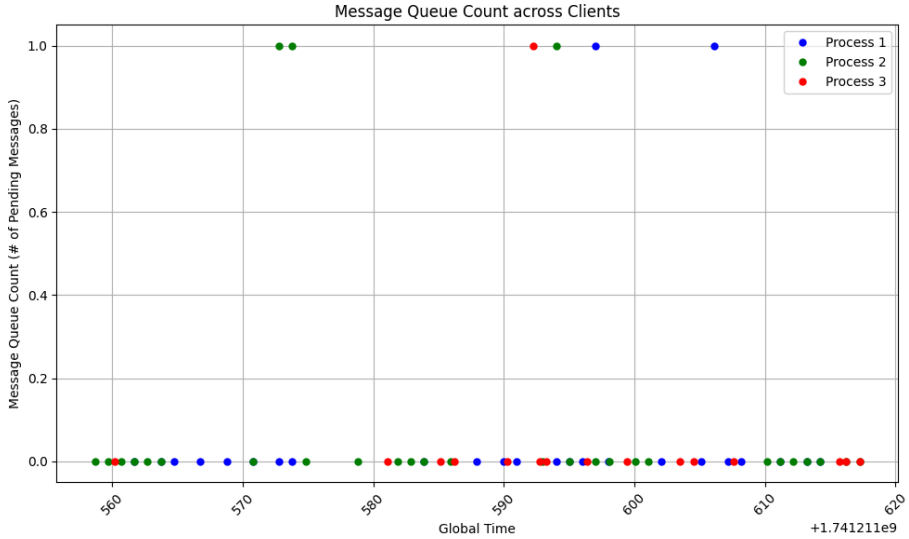
To begin our analysis, we started with **a few constants** for our set of simulations:

- We have 3 processes running, and we perform the simulation 5 times with 90 second interval durations. Here, we will only show a handful of interesting outcomes to avoid repetition (and to save readers time 👍).
- Each process is randomly initialized with a clock speed that can run anywhere between **1 instructions per second to 3 instructions per second.**
- When a process is able to perform an instruction, it will always prioritize popping a message off of the queue first. If the queue is empty, then the following things can happen (assume that we are Process 1 and that these are generalizable):
    - There is a 1/10 probability that the process will send a message to Process 2
    - There is a 1/10 probability that the process will send a message to Process 3
    - There is a 1/10 probability that the process will send a message to both Process 2 and Process 3
    - If none of these events occur, an internal event will occur.

Simulation 1







Findings & Observations

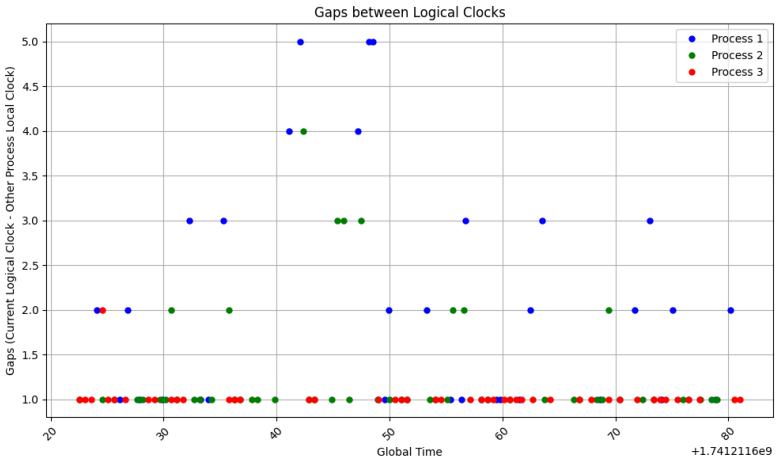
This was an exciting simulation where we noticed that all three processes are operating with very similar clock speeds, yet clock drift is clearly still occurring. Despite our short trial time, the jumps of Process 1 and Process 2 demonstrate their need to get back aligned with Process 3, highlighting that even small variations in clock cycles can lead to issues with synchronization.

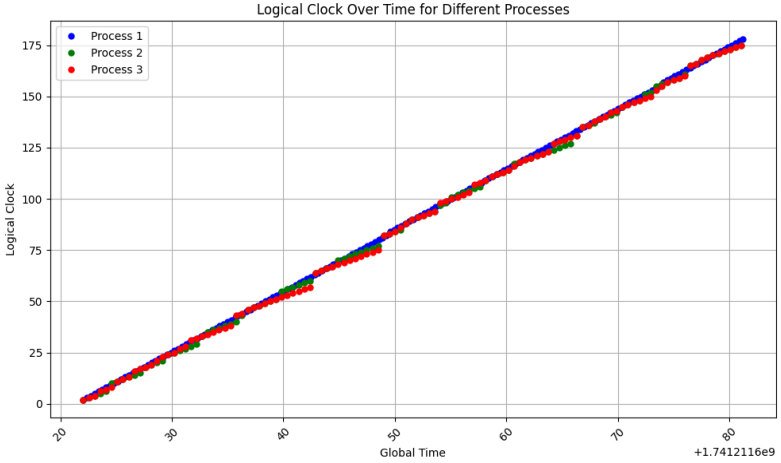
Another observation is that since all processes are processing around the same rate, there does not exist a gap bigger than 2 messages in the queue between processes. In this case, **it is actually not possible** for any process to have more than 2 messages in its queue. This is because by the time that any process executes one message from the queue, it can at most have received two other messages from other processes due to everyone's balanced clock rate.

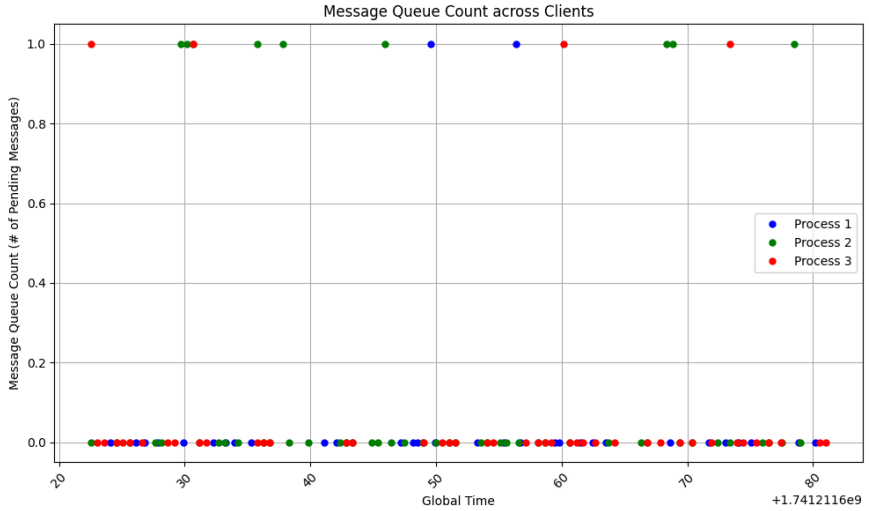
Additionally, the rate of divergence on average for the gaps between logical clocks is smaller on average than the previous simulation, just based on the fact that the probability of drastically different clock rates has decreased.

We see that these findings from our analysis are supported by additional simulations with these same conditions that we tested. We showcase another simulation below.

Simulation #2


Gaps between Logical Clocks
Process 1
Process 2
Process 3
Gaps (Current Logical Clock - Other Process Local Clock)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
20
30
40
50
60
70
80
Global Time
+1.7412116e9


Logical Clock Over Time for Different Processes
Process 1
Process 2
Process 3
Logical Clock
175
150
125
100
75
50
25
0
20
30
40
50
60
70
80
Global Time
+1.7412116e9

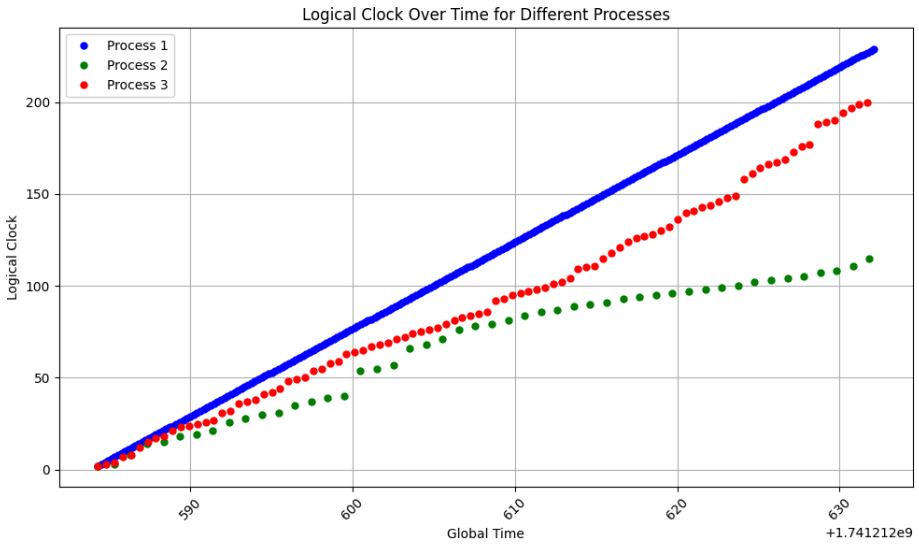

Message Queue Count across Clients
Process 1
Process 2
Process 3
Message Queue Count (# of Pending Messages)
1.0
0.8
0.6
0.4
0.2
0.0
20
30
40
50
60
70
80
Global Time
+1.7412116e9

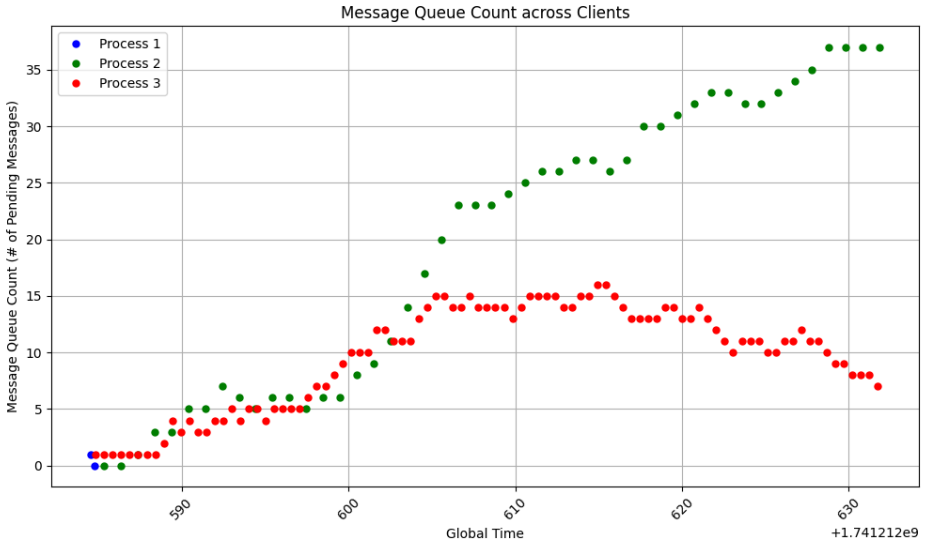
Our Analysis - Variation #2 (Smaller Probability of Internal Events):

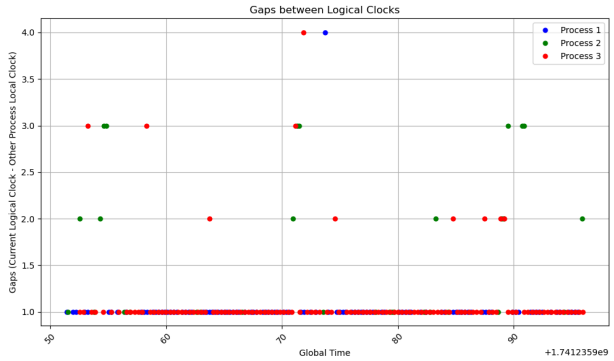
To begin our analysis, we started with **a few constants** for our set of simulations:

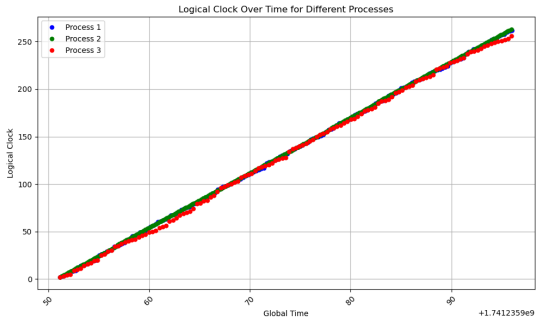
- We have 3 processes running, and we perform the simulation 5 times with 90 second interval durations. Here, we will only show our most interesting run, which happened in Simulation #2.
- Each process is randomly initialized with a clock speed that can run anywhere between **1 instruction per second to 6 instructions per second.**
- When a process is able to perform an instruction, it will always prioritize popping a message off of the queue first. If the queue is empty, then the following things can happen (assume that we are Process 1 and that these are generalizable):
    - There is a 1/5 probability that the process will send a message to Process 2
    - There is a 1/5 probability that the process will send a message to Process 3
    - There is a 1/5 probability that the process will send a message to both Process 2 and Process 3
    - If none of these events occur, an internal event will occur. Thus, the probability of an internal event is much lower.

Interesting Findings - Simulation #2









Findings & Observations

As you can see, since the probability of an internal event is lower, the process with the slowest clock speed will get overwhelmed much faster. This is because there is less time for the green process to "catch up" to its quickly filling message queue. As a result, we can see that the faster processes 3 and 1 have very few messages to process as they spend most of their time sending messages to process 2 and each other instead. This is shown through the

bottom examples very well as well, where the gaps between processes show that messages are being exchanged more often, leading to more often re-alignment of the logical clocks and smaller jumps. On the bottom, we also see that when the processes have similar speeds and a lower probability of an internal event, we can also experience momentary jumps in message queues that can also be resolved by processes as a bit of a push-and-pull. This highlights how similarities in cycle speed have a huge impact, even when messages are being exchanged more frequently, in holding a system together in sync (asynchronously of course).

Overall, our main observation here is that the increased likelihood of sending a message can exacerbate the queues of slow processes, yet it can be beneficial for allowing the system more opportunities to check the logical clocks of other processes.

ADDITIONAL EXPERIMENTATIONS :D

- We also combined these two above variations into one variation and found similar results
- Found that there was not a ton of drift after many trials (only variations 0.00001) after many minutes
- We also experimented on running 2-4 processes, instead of 3.

Conclusion

In conclusion, we were quite surprised at our key finding that the whole distributed system is only as strong as its weakest link — this is due to the now intuitive reason that, when slower systems are overwhelmed, they are to busy trying to keep up with their queues to be able to send a message out successfully. This means that any one system will never be able to receive a clear overall photo of the entire system's performance. Similarly, we found that the density of the dotted line is a clear (but unintentional) indicator of the number of events that could be processed by the target process per second, since we are plotting point plots per event and not a line graph directly. This helped provide us with richer data analysis. Analyzing our plots and testing our initial clock drift hypotheses was a surprisingly educational (and fun) assignment — excited for what's next!

AI Disclosure:

We used AI to assist us in the following:

- Regex parsing log files to be able to extract data properly
- Assistance in plotting on matplotlib and using pandas