CS262 - Design Exercise 2
Leticia Schettino - Adolfo Roquero

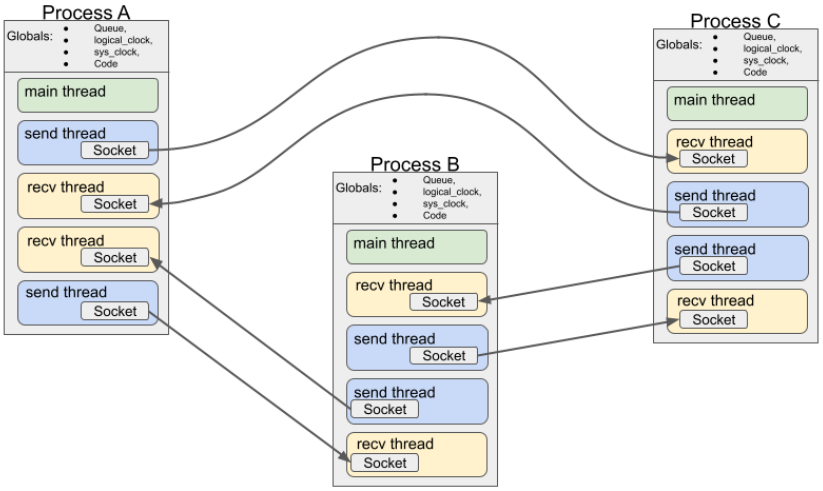
Engineering Notebook - Scale Models and Logical Clocks

Design Decisions

Technical decisions and details

Threading and Clock Rates: Use one-directional sockets with sender threads and receiver threads

The first design decision involved deciding how many sockets are used in the system and how each process handles receiving and sending messages simultaneously. The constraints on this design was to emulate a machine that is running at a given CLOCK_RATE. One consideration to simulate this was that socket reception of messages must happen immediately, while “receiving” a message in the simulated process must happen at the CLOCK_RATE. That is, only one “receive” can happen per clock cycle in the simulated machine, however real socket receives happen at a faster rate.



Our design includes a main thread per process that runs at CLOCK_RATE, then a send_thread for every Process that also run at CLOCK_RATE and finally a recv_thread for every Process that run at the real machine clock rate.
In this design, receiving threads append received messages to a global queue as soon as they receive them. However, we only count a message to be “received” when such messages are popped from the queue in the main thread. This way we can simulate that the system is receiving messages at CLOCK_RATE.

Easily implementing a system where reception of messages and sending of messages happen at different rates means having unidirectional sockets (that only send or only receive) such that the threads that run them can run at different clock rates.

Setting of the action code in MAIN thread and Read Before Write Error

Our system also uses an action code to signal sending threads when they need to send a message. The idea is that the main thread sets the code to be a list of thread ids (identified by port) that must send a message in this clock cycle. This way, multiple threads can send the message but all the clock cycle and logical clocks remain within the single MAIN thread.

One of the issues with this implementation is that it is prone to **read-after-write** error given that the MAIN thread as well as the multiple sending threads are all competing to acquire the same lock. However, there is an ordering requirement that all sending threads must have acquired the lock before the MAIN thread can get it again. This is because we want all sending threads to appear to be sending in the same clock cycle as the MAIN thread is on, before the MAIN thread is able to increase the value of the clock cycle counter again (of our simulated machine that runs at CLOCK RATE).

Unit testing

Given that this is not a classic input/output program but rather a multiprocess/multithreaded system it was harder to come up with simple unit tests.
As a first way to test correctness, we added several Assertions that check invariants at Runtime. Additionally, we moved the code assignment logic from the main thread to a helper function such that we can test input/output by looking at the input queue and checking the output code.

Problems Encountered/Debugging steps

The main realization that we had from this design exercise is that debugging a distributed system is HARD. Compared to the single-threaded single-process systems that we are used to, debugging deadlocks, ordering of events and synchronization are much harder tasks. This was particularly true given that in this pset we implemented a notion of ordering using simulated system clocks (going at fake clock rates).

Data races

We encountered problems with locking our global state to simultaneously update the queues and pop from the queue. We solved this issue by adding locks to avoid races.
We also had an issue with the read-after-write problem described above. We considered using condition_variables to wake up the MAIN thread only after all the SEND threads had sent their messages. However, we went instead for the polling solution described above, where the main thread waits until all sender threads have signaled that they have sent the message.

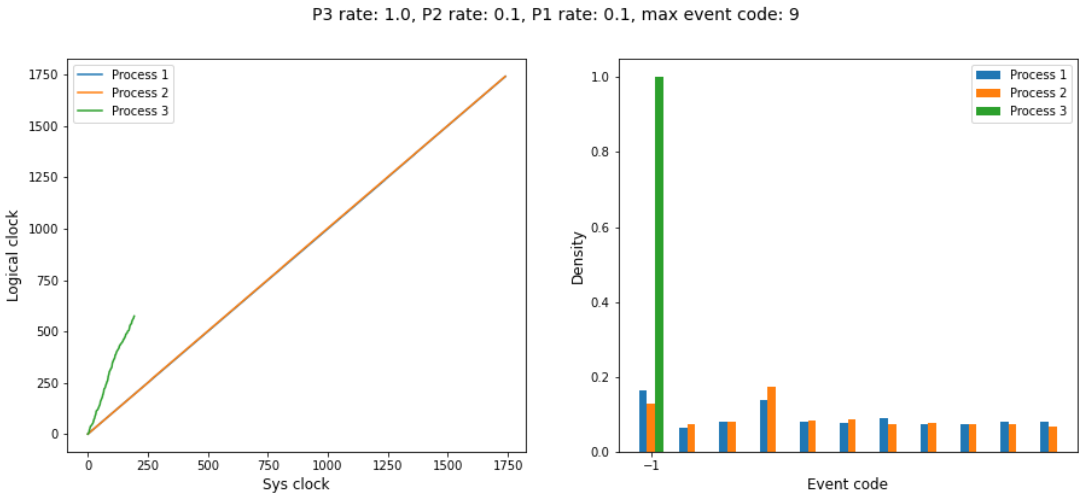
Queue Lengths

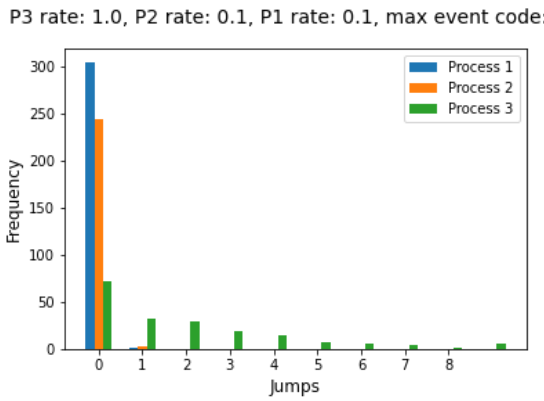
At some point during the development, we were very concerned about finding non-zero queue lengths on the logs of SEND statements (given that our implementation seemed to only SEND when the queue was empty). After headaches and unit testing, we realized that the RECV threads were acquiring the lock between the MAIN thread (that set the send codes) and the SEND threads, which caused the log of the SEND thread to have a non-empty queue. This behavior doesn't affect the correctness of the implementation since all SENDs are still happening in the same clock_cycle as the MAIN thread.

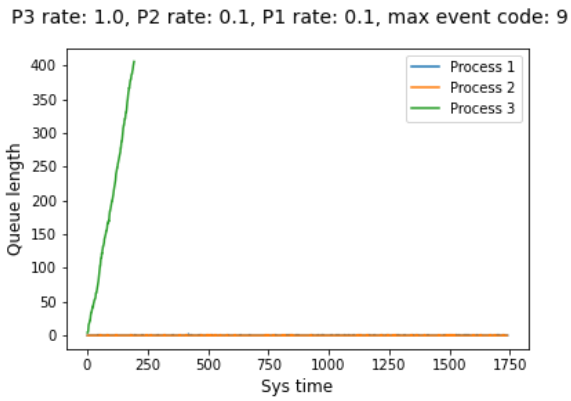
Main Experiments - Varying relative clock rate

We ran experiments using 5 different sets of clock rates for processes:
The main realization is that the clock rate of the process has a huge impact on the distributed system behavior.

We use as one illustration our experiments with a system of 2 fast processes (10 HZ) and 1 slower process 1HZ. The conclusions from these experiments were observed in all other experiments.







Actions Taken

First of all when it comes to the actions taken in each process, in the **top-right graph**, the -1 event code represents message reception while all other codes represent either send action (0-2 codes) or internal events (3-9).
We can see a clear difference in the actions taken by each process depending on their clock rate. In particular, the slow process P3 is only able to receive messages (code -1) and never has SENDS or INTERNAL actions. On the other hand, faster processes are able to run all the available actions. In some sense, the slower process is dominated by the faster ones in that it always has to receive messages and is never able to send them. This is due to the fact that given the higher sending rate of P1 And P2 as well as their higher dequeuing rate, both processes are able to keep their queues empty (and therefore perform other operations that are not only RECV). On the other hand, messages are queued in P3 at a faster rate than they are dequeued. Therefore, P3's queue is ever-growing and will never be empty. Evidence for this is provided by the **bottom-right graph** which shows the queue length for each process as a function of sys_time. We can see that P1 and P2 queues are almost always empty while P3's grows rapidly. P1 and P2 still receive messages from each other (never from P3) which is why we can observe that they still run dequeuing actions.

To conclude, faster processes **dominate** the distributed system.

In another experiment with only 1 fast process and 2 slow ones, we can see a similar pattern in that the fast process only has SENDS and INTERNAL events (and never RECVs from the other process) while the 2 slow processes only RECV.
Similarly, in a scenario where all processes run at the same rate, we can see that they all perform all actions equally.

Logical Clock and Logical Clock Jumps

The same pattern as for the Actions Taken can be observed in the Logical Clock:

(top-left graph) Faster processes' logical clock is almost always matching their sys_time (identity function) whereas the logical clock of the slow process increases much faster than the sys_time. This is due to the fact that fast processes rarely get messages with a much higher clock time and therefore rarely update their logical clock from another process. Instead they just increment it by 1 at each clock cycle. On the other hand, the slow process always receives messages from the faster processes with a higher clock number and therefore are always updating their logical clock to be that of the received message.
The graph in the **bottom-left** provides evidence for this. We can see that the distribution (normalized per process) of the jump magnitudes is very different for the fast processes than the slow process: the fast processes jumps are almost always 0-magnitude with the exception of some 1-magnitude jumps (in the few cases where the other fast process has run an extra clock cycle). On the other hand, the slow process has jumps of all magnitudes (with the maximum magnitude jump being around the difference in clock rates between fast and slow process which in this case is a difference of 9 Hz)

Another interesting (obvious) observation from the top-left graph, is that after 200 seconds, the faster processes have reached a much higher sys_clock value (1750) than the slower process (240). We would expect this ratio (1750/240) to be equal to the ration of clock rates (10) but it is lower (7.6) This is due to the fact that our simulation of a machine running at a given clock rate is not exact and processes are not exactly running at the specified rate. I.e we implemented the clock_rate with time.sleep(clock_rate) but the code is running more instructions inbetween.

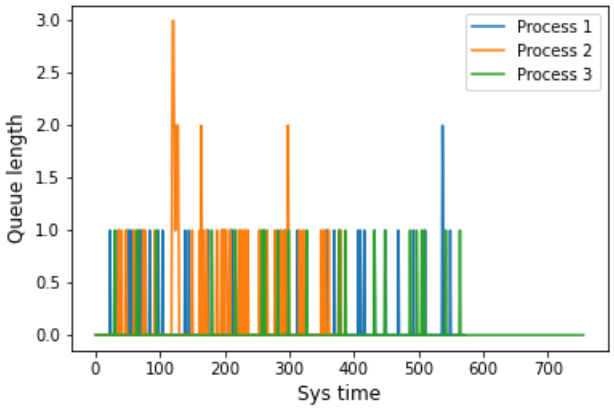
Explorational Experiments - Varying event distribution

Our approach to varying the probability of an event being internal was to run the same experiments as described above but decreasing the range of random numbers we select that map to internal events. Specifically, in the original experiment, when the queue is empty, we randomly pick from the inclusive range (0, 9) with numbers 3-9 mapping to an internal event. In the exploratory experiments, we changed this range to be (0,7), (0,5) and (0,3) while keeping the rule that if the number generated is greater than or equal to 3, then we log an internal event. The results of these experiments show that the decreasing the proportion of events that are internal (i.e. increasing communication between the processes) exacerbates significantly the same trends we have seen and discussed above. For instance, we can compare the two experiments which illustrates this point:

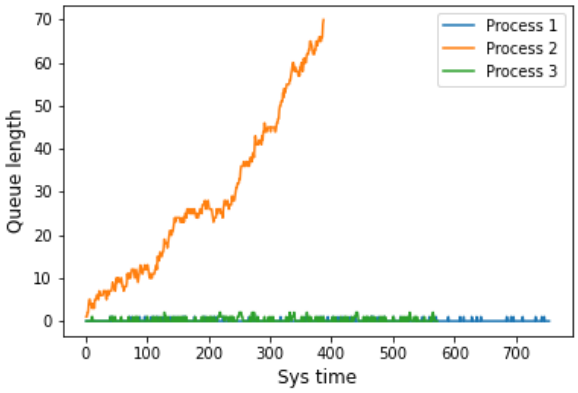
- Experiment A:
    - P1 clock rate 3 Hz
    - P2 clock rate 2 Hz
    - P3 clock rate 4Hz
    - **range of random numbers that map to send/internal events (0,9)**
- Experiment B:
    - P1 clock rate 3 Hz

- P2 clock rate 2 Hz
- P3 clock rate 4Hz
- **range of random numbers that map to send/internal events (0,3)**

Experiment A: Queue length vs system time



Experiment B: Queue length vs system time



In both plots above we had 3 processes with identical clock rates that ran for the same amount of time, the only distinction is as mentioned above, the proportion of internal events that take place in experiment B is much smaller than that in experiment A and we can see how much that yields to significant differences in the results. The intuition behind this difference is that the processes are communicating much more and therefore, the queue of messages for the slower process will increase much faster. For similar reasoning, we also expect this change from experiment A to experiment B to impact other metrics that we have analized such as the logical clock jumps that take place per process and the slope of the sys clock vs logical clock times. This is because the only way that one process can impact another process's logical clock time is when it sends a message to another process with a value that is greater than the current value logical clock of the process receiving the message. On the other hand, an internal event does not impact the logical clock of other processes. Therefore, the higher the probability of a message being sent to another process, the faster we expect to see the impact of the drift in the logical clocks and the different clock rates. All the results we obtain from this experiment agree with the reasoning above.