Scale Models and Logical Clocks
5 March 2025
Ruben Valenzuela and Jesse James

Access the Repo: https://github.com/4CCESS/cs262-scale-models

DESIGN
Original plan of function:

1. Start a master process
2. Fork 3 processes to produce virtual machines, designate PID to current iterator value, and call the runProc method within each process
   following steps occur within runProc for each process
3. Open the log file and make first entry
4. Create a UNIX domain socket with address /tmp/smproc_[pid]
5. Bind socket and start listening
6. Connect to other two machines
    a. send connect requests to all
    b. accept as many requests as were not handled during previous step
7. Generate random clock rate
8. Create a listener thread to receive messages and place in queue in real-time
9. Call operationsRun method to process message from queue/handle operations if no messages are available
    a. generate random opcode
    b. check for messages in the queue, update clock and print to log if available
    c. send direct/broadcast or register internal event with clock update if not
10. After one minute, close all sockets and exit child processes
    main process
11. Wait for all child processes to finish, return success code

To implement the aforementioned plan, we used the fork() method to first create the VM processes. Then, inside runProc(), we used the ofstream data type to open the log file, and created an enterLog() helper function to standardize log entries with the current time. For our sockets, we used the built-in C++ socket library, but specifically opted for UNIX domain sockets rather than TCP sockets to reduce complexity, since all processes are local. After creating the socket address and starting listening, we then began the connections.

Issue 1: After using the approach in step 6 and continuing to create the rest of the functionality, we noticed that it appeared that the processes were connected and sending messages worked fine, but none of the processes were actually picking up messages to read from the queue. We attributed it to some sort of race condition tied to the sending of the connect requests to all others and subsequently accepting. Rather than investigating this further, we decided to simplify the process by ensuring that exactly one connect request was being sent between each process: all processes will attempt to connect only if the target PID is higher than the sending

PID, and they will subsequently try to accept connections from lower PIDs until they have two confirmed connections. This worked immediately upon implementation.

Issue 2: To help verify that connections were being established between each process every time, we created a handshake method to send and confirm reception of messages to between hosts as part of initialization. This made debugging faster and removed confounding variables like differing clock rates.

To generate the random values for both clock rate and operation we used the random library and specifically calculated clock rate by dividing one second by the clock rate and casting that to be a duration value. Duration and other time-based features leverage the chrono library.

In the listener thread, we use the select() method with a 1-second timeout to check the socket file descriptors associated with the other two processes and use the std::queue data type to store the messages.

In the operational loop (operationRun()) we parse incoming messages to extract the sender's clock, and use std::max to compare to our local logical clock, updating accordingly. If no messages are available, we used the randomly generated value to determine the course of action, which involves either socket writes for 1-3 or just a clock update and log record otherwise. To ensure no more operations are performed until the next tick, we then put the thread to sleep.

Issue 3: Not so much of an issue, but after designing, we went back and standardized all log entry messages to give a cleaner appearance, and used dashes to indicate breaks in phases (e.g. initialization to operation) and periods to indicate clock gaps following received messages.

Finally, to handle testing, we used the Catch2 library to create test or core connection and helper method functionality. Specifically, we made sure to validate the handshake, since the message selection and sending processes are identical to their use in the listening and operational methods.

OBSERVATIONS

Standard Experiments

In our standard experiments, we observed that the slower clock rates had more frequent gaps, and specifically a clock rate of 1 typically resulted in a pattern of gaps after nearly every message, regardless of the clock rate of the fastest machine. Additionally, we observed that after periods of no gaps, the gaps tended to be larger, and this can be attributed to a series of internal events (which are still faster with higher clock rates) followed by a sent message. Depending on the clock rate difference, we observed jumps of 4-10 on average, with some jumps up to just under 20. Additionally, we observed queue lengths of at most 1-2, except in the

case of systems with a VM of clock rate 1 and another VM of clock rate 6. The slower queues reached up to 27.

Low Clock Variation Experiments(range 1-6 to 1-3)

Since none of the processes in the system is moving significantly faster than the rest, that is, messages are sent and received at a similar frequency, at any point in time, their logical clocks will have very close values as they are updated when messages are exchanged. Consequently, for any process, for any logical clock update, the change will be small. Another observed benefit of this model is that queue sizes are significantly smaller on average. For instance, for a process with a clock rate of 2, the maximum queue size recorded was 1. Cf. processes with a clock rate of 2 but a low probability of internal events in the next section.

Log file for a low-clock variation process with a rate of 2.

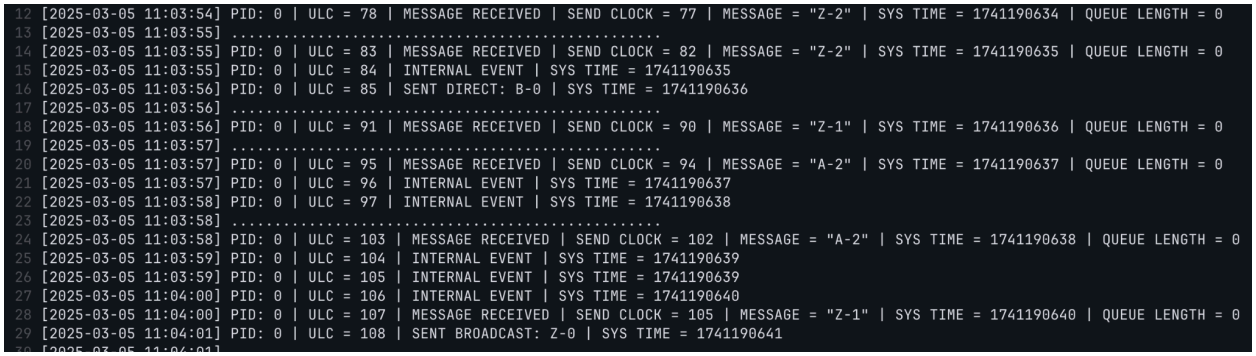
```
[2025-03-05 11:19:07] PID: 1 | ULC = 59 | INTERNAL EVENT | SYS TIME = 1741191547
[2025-03-05 11:19:08] PID: 1 | ULC = 60 | INTERNAL EVENT | SYS TIME = 1741191548
[2025-03-05 11:19:08] ..................................................
[2025-03-05 11:19:08] PID: 1 | ULC = 65 | MESSAGE RECEIVED | SEND CLOCK = 64 | MESSAGE = "A-2" | SYS TIME = 1741191548 | QUEUE LENGTH = 1
[2025-03-05 11:19:09] PID: 1 | ULC = 66 | MESSAGE RECEIVED | SEND CLOCK = 64 | MESSAGE = "A-0" | SYS TIME = 1741191549 | QUEUE LENGTH = 0
[2025-03-05 11:19:09] PID: 1 | ULC = 67 | INTERNAL EVENT | SYS TIME = 1741191549
[2025-03-05 11:19:10] PID: 1 | ULC = 68 | INTERNAL EVENT | SYS TIME = 1741191550
[2025-03-05 11:19:10] PID: 1 | ULC = 69 | SENT DIRECT: B-1 | SYS TIME = 1741191550
[2025-03-05 11:19:11] PID: 1 | ULC = 70 | INTERNAL EVENT | SYS TIME = 1741191551
[2025-03-05 11:19:11] PID: 1 | ULC = 71 | INTERNAL EVENT | SYS TIME = 1741191551
[2025-03-05 11:19:12] ..................................................
[2025-03-05 11:19:12] PID: 1 | ULC = 75 | MESSAGE RECEIVED | SEND CLOCK = 74 | MESSAGE = "A-2" | SYS TIME = 1741191552 | QUEUE LENGTH = 0
[2025-03-05 11:19:12] PID: 1 | ULC = 76 | INTERNAL EVENT | SYS TIME = 1741191552
[2025-03-05 11:19:13] PID: 1 | ULC = 77 | SENT DIRECT: A-1 | SYS TIME = 1741191553
[2025-03-05 11:19:13] ..................................................
[2025-03-05 11:19:13] PID: 1 | ULC = 81 | MESSAGE RECEIVED | SEND CLOCK = 80 | MESSAGE = "A-0" | SYS TIME = 1741191553 | QUEUE LENGTH = 0
[2025-03-05 11:19:14] PID: 1 | ULC = 82 | MESSAGE RECEIVED | SEND CLOCK = 81 | MESSAGE = "A-0" | SYS TIME = 1741191554 | QUEUE LENGTH = 0
[2025-03-05 11:19:14] ..................................................
[2025-03-05 11:19:14] PID: 1 | ULC = 84 | MESSAGE RECEIVED | SEND CLOCK = 83 | MESSAGE = "A-0" | SYS TIME = 1741191554 | QUEUE LENGTH = 0
[2025-03-05 11:19:15] PID: 1 | ULC = 85 | MESSAGE RECEIVED | SEND CLOCK = 81 | MESSAGE = "Z-2" | SYS TIME = 1741191555 | QUEUE LENGTH = 0
[2025-03-05 11:19:15] PID: 1 | ULC = 86 | MESSAGE RECEIVED | SEND CLOCK = 85 | MESSAGE = "A-0" | SYS TIME = 1741191555 | QUEUE LENGTH = 0
[2025-03-05 11:19:16] PID: 1 | ULC = 87 | SENT DIRECT: B-1 | SYS TIME = 1741191556
```

Low Probability of Internal Events Experiments (range 1-10 [70%] to 1-5 [40%])

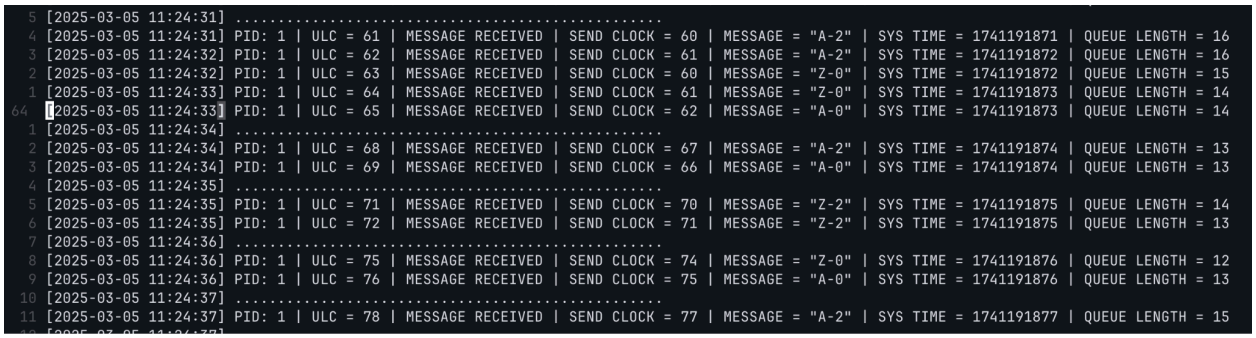
After the implementation of low internal probability, log files show that the size of the queue at any point, on average, is larger as opposed to that of the normal model with standard internal probability. For instance, for a process (virtual machine) with a clock rate of 2, when the internal probability was standard, the maximum queue size recorded was 1 whereas for a process with the same clock rate of 2 but with a low internal probability, the maximum queue size recorded was 37. The average queue size changed similarly. This is reasonable as a lower internal probability for a process means that such a process will take an element out of the queue less frequently.

Another thing to be noted is that with a lower internal probability, messages are exchanged between processes more often and logical clocks are updated more frequently on average. This explains why processes with similar clock rates, when given a lower internal probability, their log files show smaller steps between logical clock updates as opposed to their standard counterparts. See log file snippets.

Log file for a standard process with a clock rate of 2.



Log file for a low-internal-probability process with a clock rate of 2.



Moreover, logical clock drift is significantly smaller for systems with lower internal probabilities,. which can be explained by the fact that processes clocks are being updated more frequently on average.

Other Experiments

We tried two other experimental cases. The first involved low clock variation with low probability of internal events. We observed that this had compounded effects of both characteristics individually. For example, in the cases of clock rate distinction >1 (VMs of 1 and 3), the slower clock rate VM experienced much longer queue lengths, up to 30. However, the message frequency plus lack of variation resulted in much smaller jumps, typically 2-3 with some occurrences of 4-6 in slower VMs.

The second extra experiment involved extremely high clock variation (1-20). Interestingly, the largest jumps were comparable to those in the standard tests, but the frequency of midsize jumps was higher (i.e. more sequential jumps of 4-6). Additionally, the queues never exceeded a few messages. This makes sense, as the fastest VMs are certainly producing more events and are updating their clocks considerably faster, but the jumps are also constrained by the relative difference in clock rates, rather than their absolute values, and the fact that internal events have the highest probability balances keeps the queue lengths low. This reaffirms a conclusion that could be drawn from the standard tests: if clock rates are close together, it doesn't matter whether their values are high or low. The only difference will be the ending values of the logical clocks, due to overall higher event frequency.