
Mar 2 EA

- read spec, set up repo
- initial thoughts, shared by Eric: use python, sounds fun
- files pieces needed: process code, log files, analysis code for log files
- design considerations: should put a good bit of thought and discussion into log file format,
  that seems like most important piece, also get to set msg protocol (simple to self-roll)
- things that need to go into log:
        once: timescale (if that's what we want to call it)
        per step: system time, LC at end, operation, op details if message (sender/recipient, their LC?)
                - track message queue length (wrt LC or system time?)
                - other sender/recip info? their LC, their timescale?
                - is logging internal events a can, must, or should?
- analysis possibilities:
        - independent vars: range of timescale, specifically set timescales, sending probabilities
        - dependent vars: (prob histograms of) jump size, drift in LC values across processes, queue length, gaps
- plan for first steps: helloworld.py to brush up on python, chat app to see sockets,
                write skeleton for pieces, log spec to be discussed before going into detail
        - need to pick between python 2 and 3, 3 has been rec'd to me but should prob decide
          based on sockets (recall - our chat app had a py socket lib that meant it wouldn't run
          on windows machines, but googling suggests atexit? should ask that half of the group)
        - py3 seems to have decent socket library, will go with that for now, can change back if Eric
          strongly opposed.
- what does each process need to do:
        - on init, randomly set timescale, init LC and system time
        - per cycle:

        - wait a lot, according to time scale
        - send messages, possibly
        - update LC
    - always: be receiving messages
- actually, I think we need initializer to introduce them all to each other/give them each other's sockets, their own log file location, so each process' main should take in a timescale, filename, and 3 sockets and those will be in threads started by the initializing program? played around with threads, realized that to keep memory separate need to use multiprocessing rather than threading.
- log entries and messages seem like they might benefit from having version numbers (do messages need to do anything except contain an LC value?) --- this may change, but for now in code the message queue is just a list of LC values
- is anything going to be upset about force quitting the processes? do we need to close the log files?
 - very possible that I'm doing waiting wrong (not actually using wait())
- SUMMARY: played around, looked into threads and multiprocessing and writing to file, drafted skeleton code for processes and initializer with missing pieces (messaging and logging) in lines starting with ###
- NEXT STEPS: talk about options for log and message formats, design first pass analysis, figure out sockets

---
Mar 3 EB

Meeting Emma and Eric 3/3/16:
- Need to do socket connections
- how to format messages and log:
    - messages: only need logical clock, check that it's an int
    - log format: system time \t clock time \t some encoding for operation
    - header for log file?
- Eric does socket by sunday night
- code for analyzing gaps and all that (plot logical clocks for each process vs system clock): Emma
- log code: Emma
- next meeting: monday

Mar 5 EB

Read about python sockets at https://docs.python.org/2/library/socket.html and played around with the two examples in section 17.2.2
For our purposes, added 6 sockets, one for each (from i, to j) for each pair (i,j) of processes.
Needed to have each process sleep for 2 seconds after they start listening and before they connect so that they always connect to a socket that is already listening.

Mar 8 EA

Update: Eric added sockets, can start running tests
Turns out there's an issue with our listening and waiting threads plan, because only one thread can have the global interpreter lock at a time.
But just running code seems fine? Maybe because we have that if statement rather than sleeping.
I'm going to run it and look at the logs.
-> having some trouble, looks like regardless of time scale, each log entry is 1 sec apart
- put in counter, we check system time 10k times per op. So that's….worse design than I'd thought.
Definitely need to sleep, which means working out this GIL business.
-> online tutorials don't address this problem, hard to find info on it, but threads can .wait_for(predicate,timeout), http://jessenoller.com/blog/2009/02/01/python-threads-and-the-global-interpreter-lock night be helpful
-> should have a lock on the shared message queue, though

SUMMARY: took out global message queue, added logging, drafted analysis code (still want to do max(LC)-min(LC) hist), added comments, general debugging
NEXT STEPS: work out sleeping and listening, honestly super confused about what we are and aren't allowed to use for this assignment, thought it

was on the course message board but it isn't

Mar 11 EB

Trying to fix the problem with having two threads for each process where one of these two that needs to be timed and where they both share some resource (the queue).
Changing the code so that there is only one thread per process and the listening for incoming messages happens at the end of the main loop. This shouldn't be a problem for the timer since checking for messages is much faster than the time gap for any timescale. Additionally, performing operations also happens much faster than the time gaps between which messages are being sent, so each machine will still receive messages one at a time from other machines.
A few notes:
 - need to be careful in which order the setting up of the sockets happen to avoid deadlocks
 - made the sockets non-blocking so that when there is no message to be received, each process can proceed to another iteration of the main loop instead of waiting for a message.

Mar 13 EB

Adding markers to the end of a message because if a machine starts a bit later than an other, it could be that when it reads messages from a socket, there are several messages that have already been sent, in which case it just wants to read the last message sent to catch up on the LC. Once all machines are started, we don't really need markers since all machines will receive messages one at a time.

Analyzing data.

For the jump counts, removed all jumps of size 1 because they are just regular updates of the time clock and it's easier to read the histogram without them.
For the gaps plot, for each second where there is at least one logical clock

value for each machine, we consider the smallest logical clock recorded for that second for each machine.

Forgot to add length of the queues to the log files, restarting all experiments

Discussion of the data:

The plots are in the allFigures files, each row of plots correspond to a run of the model. For each run, we have four plots: the first is the logical clock of the different machines vs the system time, the second is the largest gap in the logical clocks of the machines vs the system time, the third is a histogram of the counts of the different sizes of jumps when a logical clock gets updated by receiving a message and the last is the length of the message queue for each machine vs system time.

We observe that the logical clock of the machine with the highest number of clock cycles per second is the largest logical clock at anytime, which is expected since it updates its logical clock faster than the other machines. The differences between the logical clock of this machine and the logical clock of the other machines at a specific system time varies differently depending on the settings.

When the number of clock cycles per second are close for the different machines, for example in the runs corresponding to the 3rd and 5th rows, the logical clock values mostly stay synchronize. They sometime fall a bit behind for the machines with the lowest number of clock cycles but regularly catch up. This behavior implies small gaps in the logical clock and small jumps when the logical clock gets updated after receiving a message.

When the number of clock cycles per second are very different for the different machines, for example in the run corresponding to the 4th row, and to a lesser extent the 1st and 2nd rows, the slower machines either rarely catch up the faster machines in terms of logical clocks, or even never catch up. Never catching up, for examples in the 4h row, is due to the length of the message queue always growing, so when the machine reads a message from the queue, these messages are increasingly out of date as

the system time increases and this machine then falls behind in logical clock value. Even if the messages accumulate in the message queue, a machine is sometime able to catch up due to the randomness involved in the frequencies in which messages are received, as we can observe in the 2nd row. In either case, when machines fall behind in logical clocks, we of course observe large gaps in the logical clock but also higher jumps in this logical clock.

A very slight change in the model would have immensely helped synchronizing the different machines and prevented some machines to fall behind. If instead of having a queue that was first in first out, we had a stack that was last in first out, then the slower machines would regularly be able to catch up since they would get the most up to date logical clocks of the other machines and not old messages.

We also did runs with smaller variation in the clock cycles and a smaller probability of the event being internal. The figDiff file are the plots obtained for a typical run observed in such a setting, in this specific example, the clock cycles are 3.1, 3.3, and 3.5 and we pick a random number between 1 and 5 instead of 1 and 10 to reduce the probability of an internal event. With these plots, we observe that the logical clocks are always almost synchronized, with very small gaps and jumps in the logical clocks.