Github repository: <https://github.com/akabcenell/Assignment2.git>

**Lab Notebook**

Assignment:

Create three virtual machines at 3 speeds randomly 1-6/sec

Each initialize communication with each other over sockets to send LC values

Each clock cycle:

If messages, receive messages and take maximum of own or sent LC value + 1

If no messages, 1/10 send to vm1, 1/10 send to vm2, 1/10 send to vm3, 7/10 do nothing, in all cases augment clock

In background: receive messages and add to message queue, not limited by individual clocks

VMs must act simultaneously, need to be on separate threads

Background message receiving also needs to be happening while VMs are in clock cycle loop, need separate threads for those

Looked at python socket documentation, made toy one send/one receive example to test

Using basic settings, code based on examples from python docs sockets howto: https://docs.python.org/2/howto/sockets.html

Arbitrarily use ports starting at 1000, should be out of range of anything else happening

Implementation: receiver binds to predetermined port

Sender creates client socket on separate thread and tries to connect to that port

Receiver, which sits in infinite loop, detects connection and creates client socket to receive message

First try crashed due to bug, then second try complains about reusing sockets

Evidently long timeout, need to be careful about closing properly or will need to cycle ports

Success, will directly use code for VM communication

Start writing VMs, which have an internal state, so are a class that extends threading.Thread to allow simultaneous operation

In \_\_init\_\_, initialize logical clock, open receiving socket on given port

All three inits finish before run starts, ensuring that all VMs are listening on the right socket before messages are sent

Initialize empty Queue, which are thread safe, that will be used to pass/store messages received from socket client threads

Initially use infinite loop for simplicity in run

Implement operations as described in assignment, launching new threads for each message sending

Create log file to be able to debug easily: every clock cycle, writes current time, LC value, operation for clock cycle, size of message queue, vm message was received from if that was the operation

Get loop to occur at appropriate vm clock cycle speed:

Try just sleeping after every cycle

Too slow, drops ~20ms per clock cycle due to execution time of code in loop

Use cronos package

Is actually global, so takes only last frequency/all ran at last assigned frequency

Sleep for normal wait in clock cycle speed minus execution time of loop calculated using datetime arithmetic

Only drops 1-2 ms/s, good enough

Use datetime to get initial and current time, and use this to terminate loop after 60 seconds

Text log files very challenging to parse in a way that offers reasonable insight, especially when trying to match up the times and LC values when clock speeds very different, need some visualization

Care about trend of LC values, nature of jumps

Size of message queue should also give insight into behavior if one lags significantly

Plot LC, dLC/dt, qsize over time

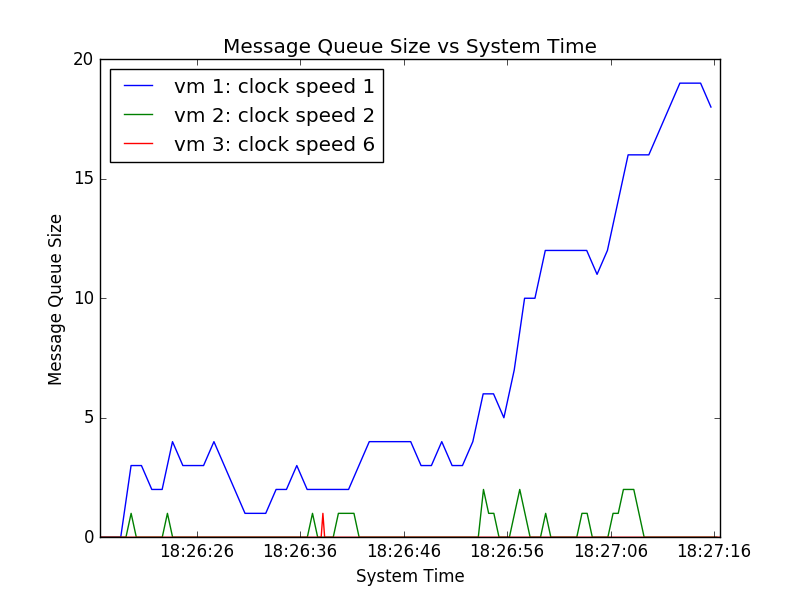
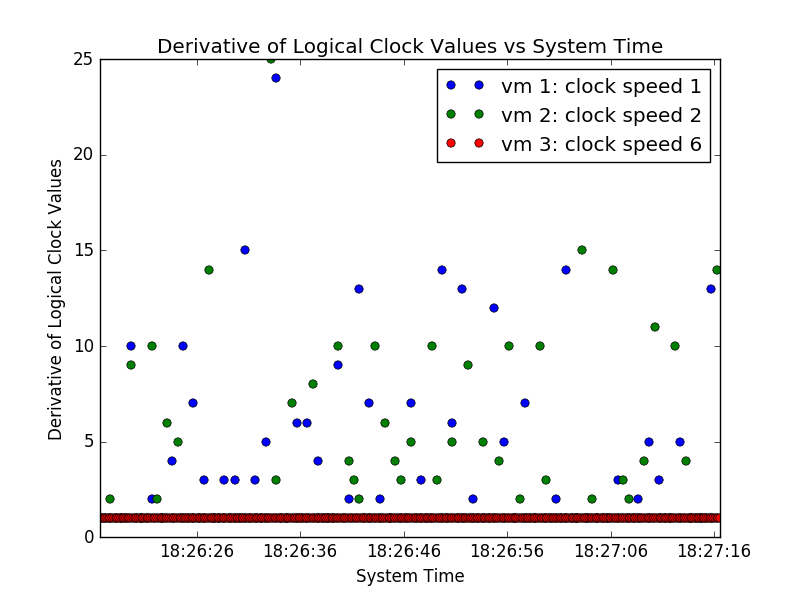
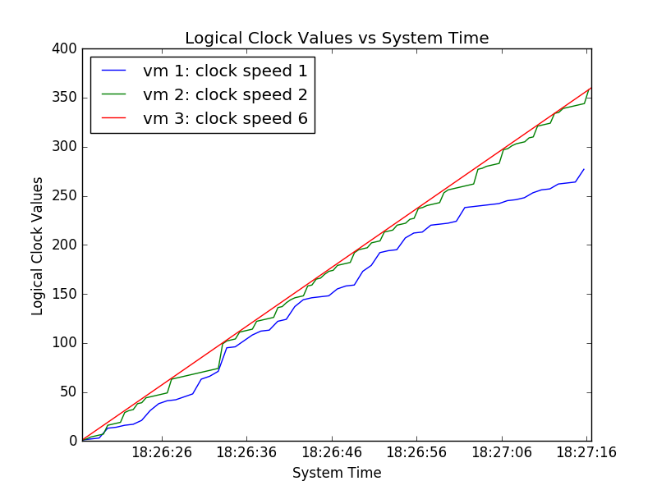
Wrote plotting code, straightforward

Begin analysis

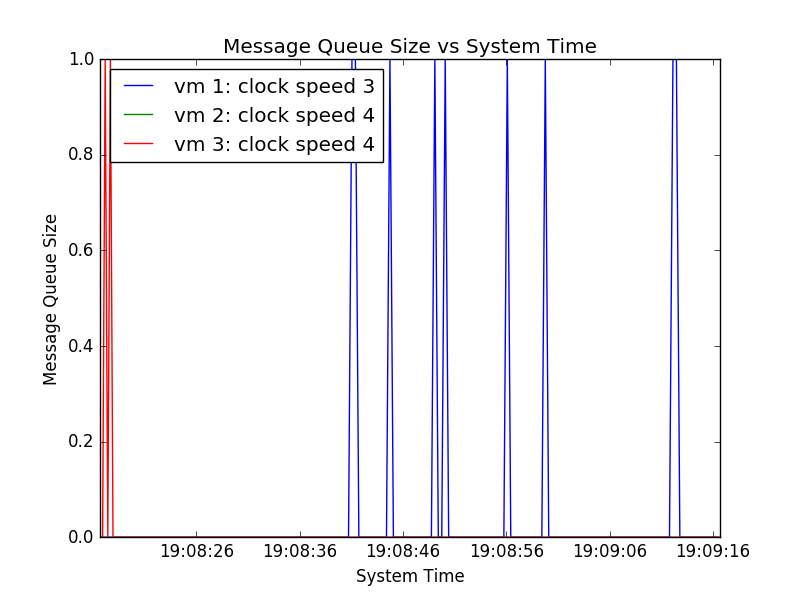
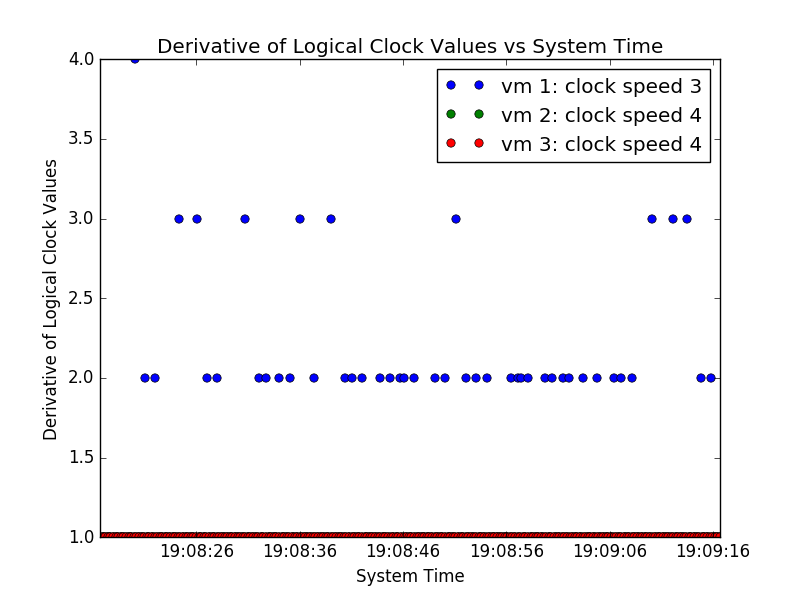
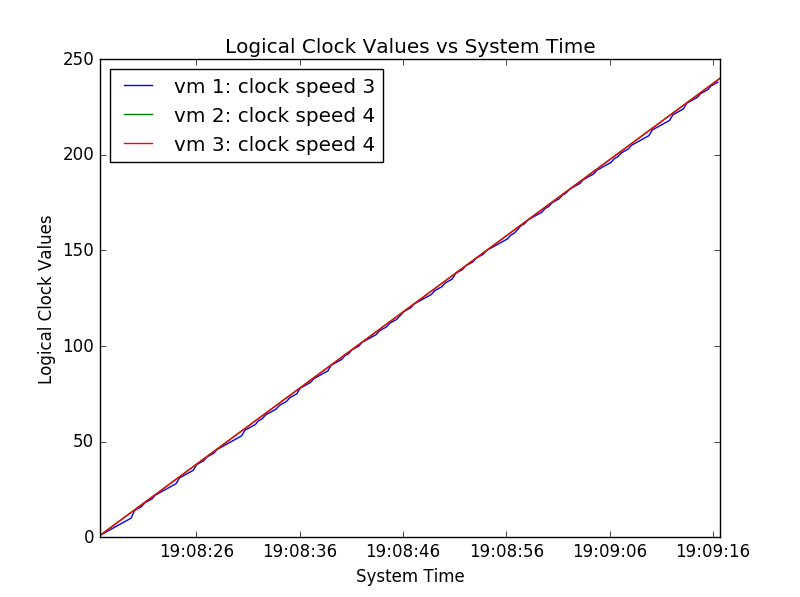
**Analysis:**

We run five trials with the default parameters (clock speeds between 1 and 6 operations/second, 10 possible operations). We then plot the logical clock values, the derivative of those values to better track discontinuities, and the message queue size, over the one minute duration of the virtual machine system to more easily visualize the time evolution of the system:

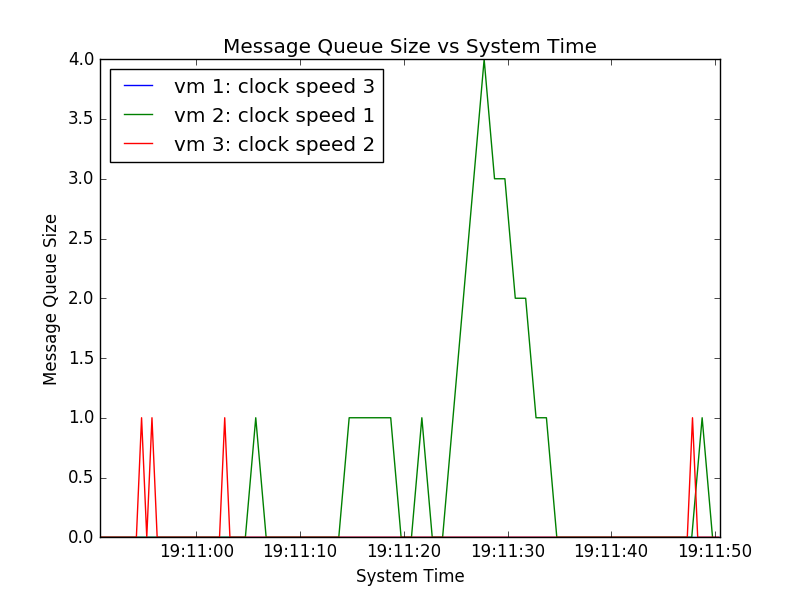
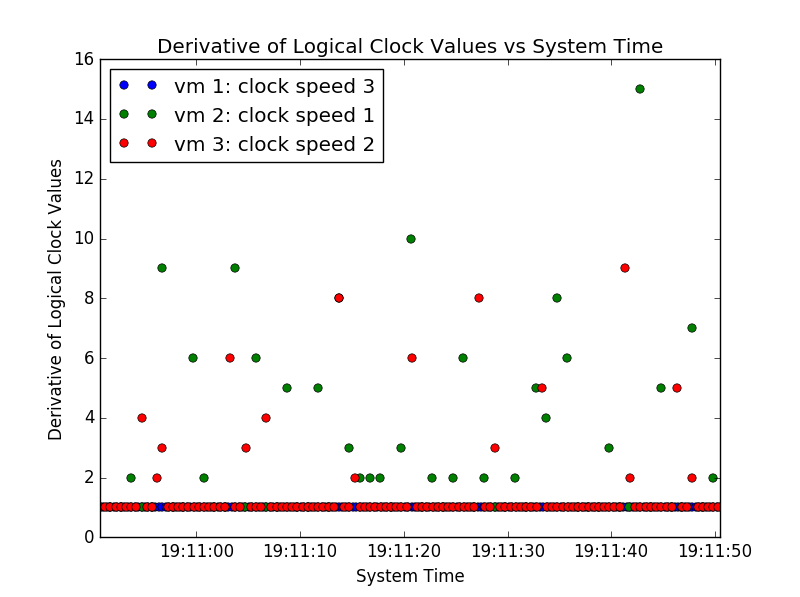
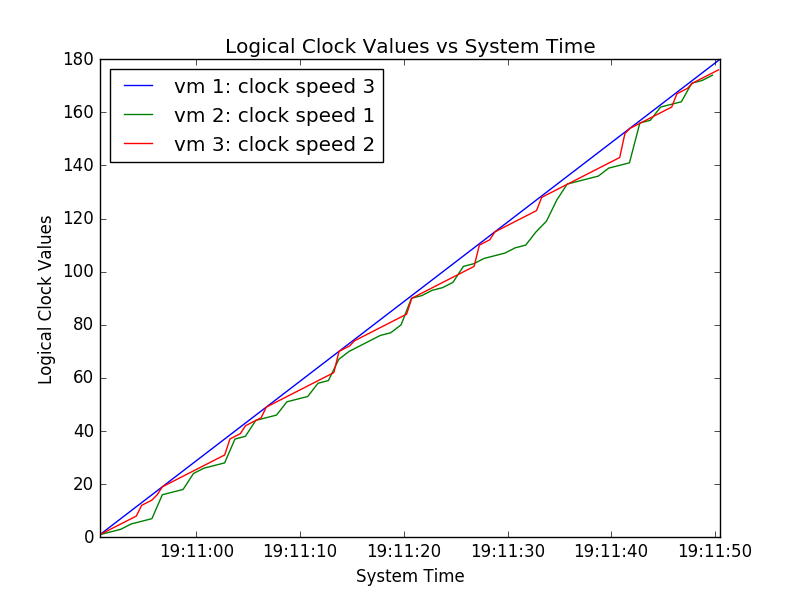
Trial 1:



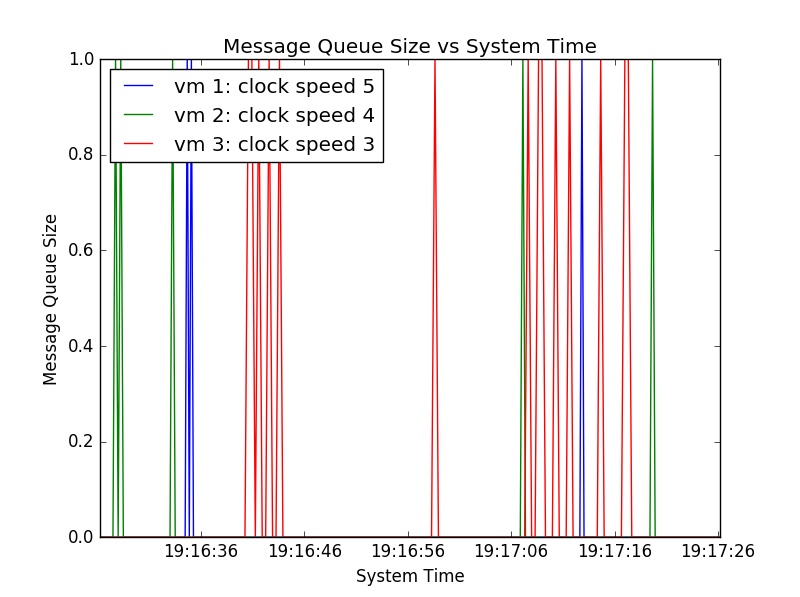
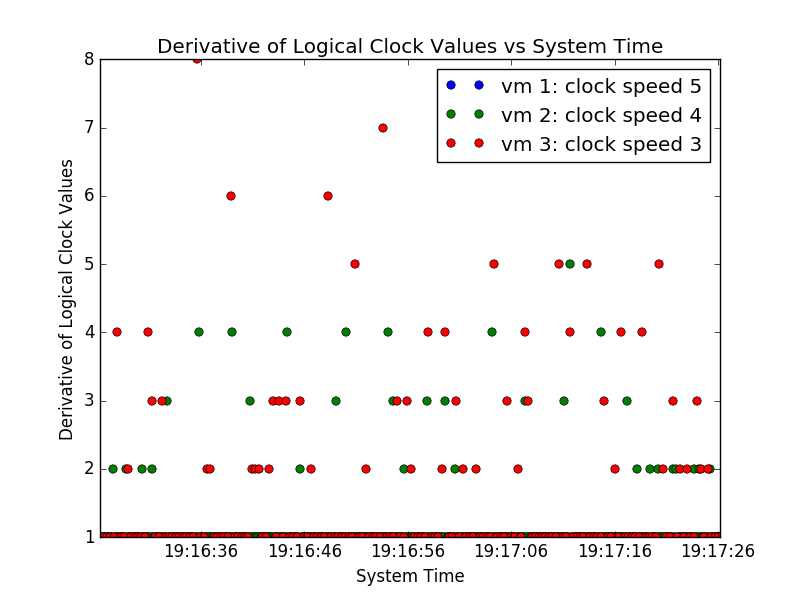
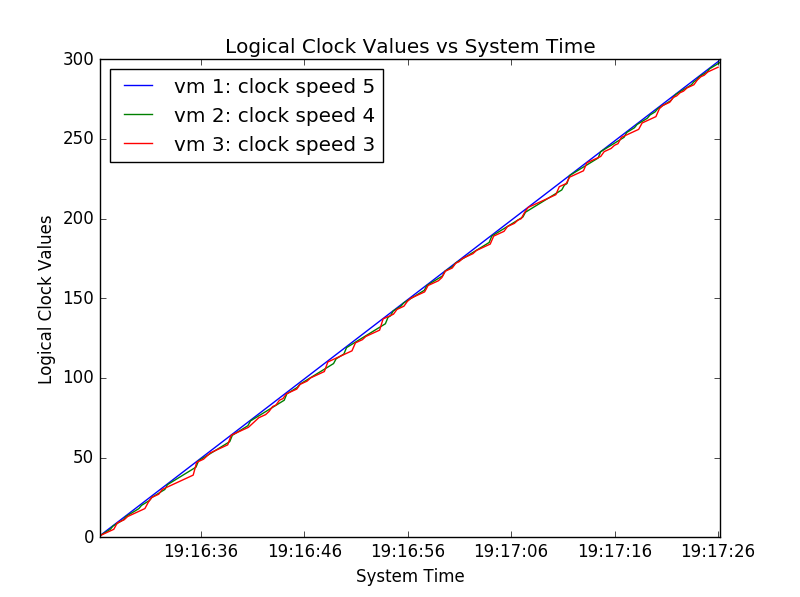
Trial 2:



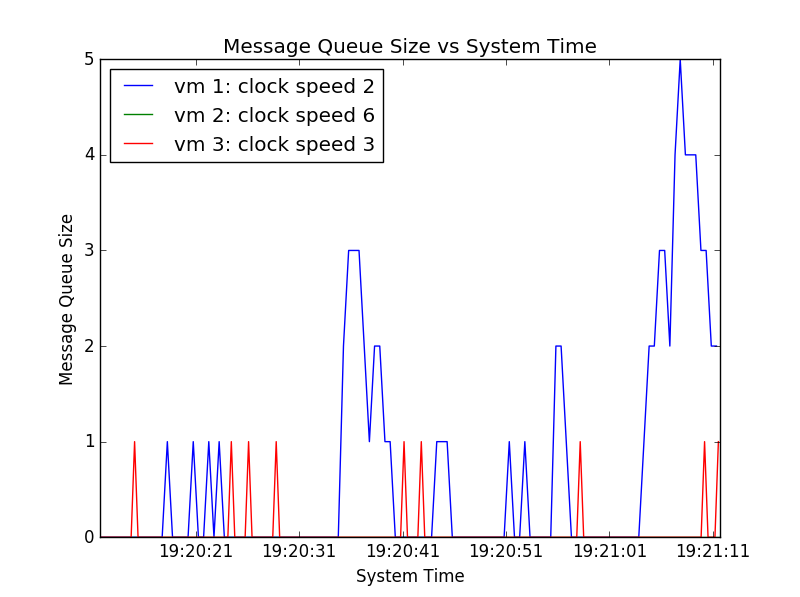
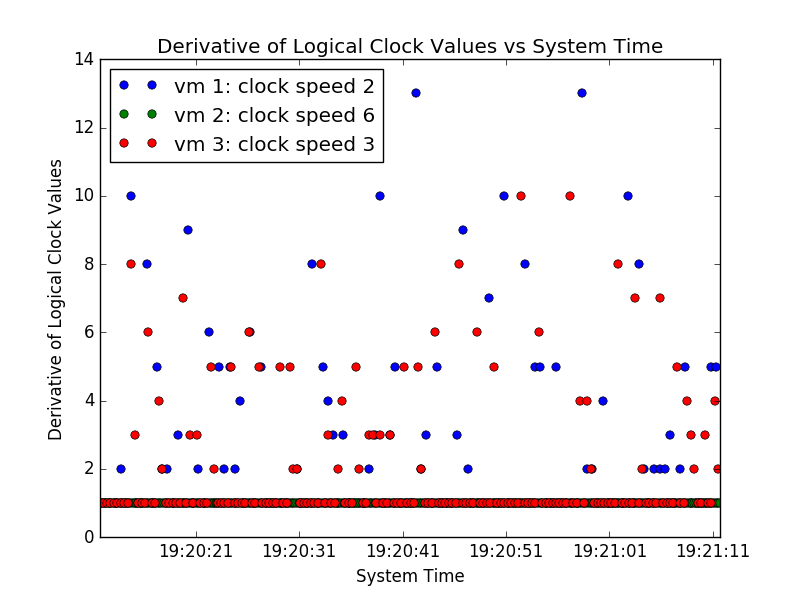
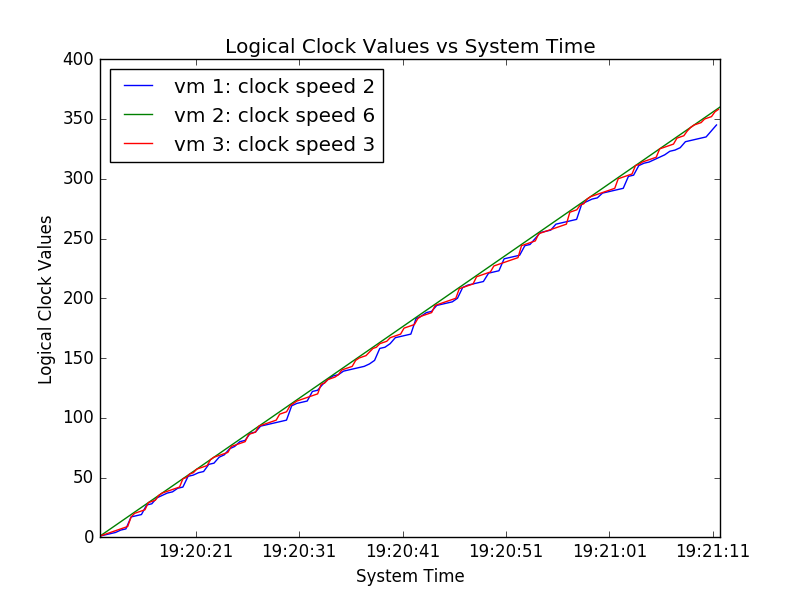
Trial 3:



Trial 4:



Trial 5:



Let us analyze the simplest case first, trial 2. Here, two clocks are running at 4 cycles/s and one is running one cycle/s slower. Since the two clocks running at the same speed are also the fastest clocks in the system, they will always have the same logical clock values, as if they receive a message from each other they will always read it on a later clock cycle with a higher LC value and thus always take their own, and the slower clock will always have an LC value no more than their internal value here so the LC value in the message can similarly never be greater than their internal LC value. Thus, neither of the clock speed 4 VMs ever have a jump, or drift away from one another. The speed 3 VM runs slower and thus occasionally drifts away. However, a message from either of the other two VMs brings it back to the same value, which happens quite often (2/5 fast clock cycles), and it is only running slightly slower so there is no time for a message queue to build up and for it to be reading old messages with much lower LC values. Thus, the largest jump is 4, and generally it just jumps 2 every few cycles, which keeps its LC value very close to the faster VMs LC value.

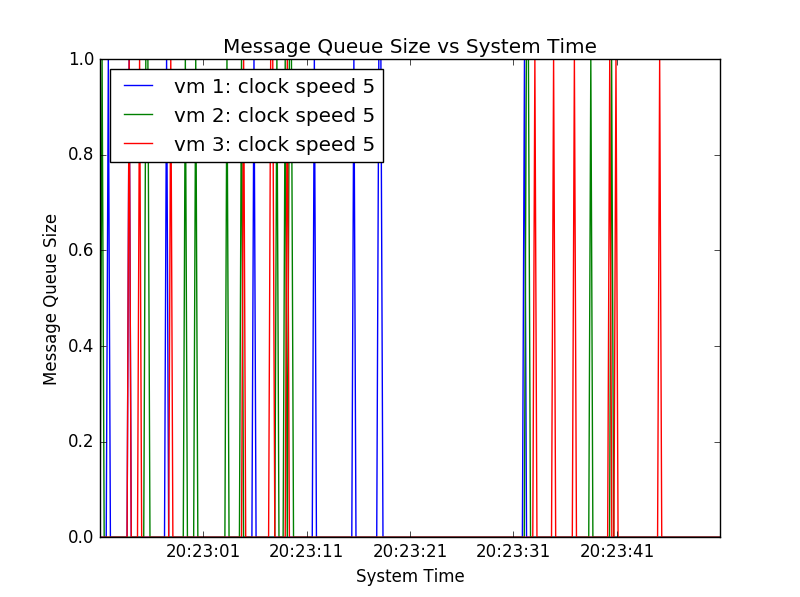
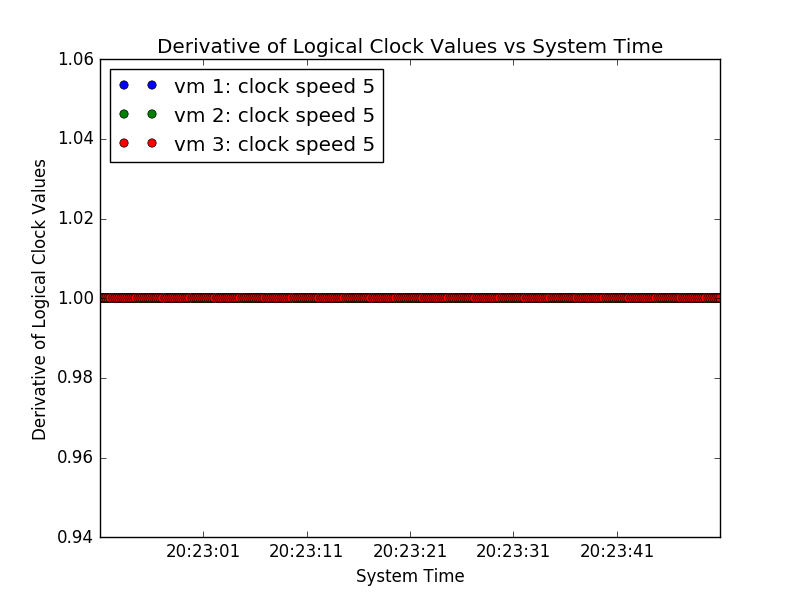
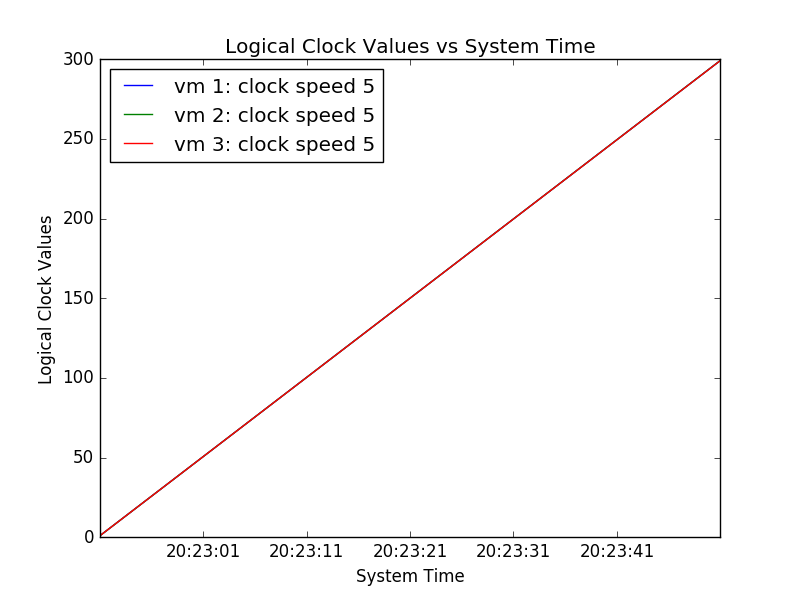
Trial 3 shows similar behavior. Here, the VM with the fastest clock at speed 5 has a linear LC as expected, and the one with a speed 4 gets updated sufficiently often that it never needs to jump more than 5 to catch up. The VM with speed 3 does undergo slightly larger jumps, up to 8, but this isn’t particularly large, and because the total difference in speed isn’t larger than a factor of 2 between the slowest and the fastest, messages are read quickly and there aren’t enough sends between the clock cycles of the slowest VM to build up a queue.

Trial 3 is somewhat similar, with all of the clock speeds spaced by 1, but here the slowest VM runs 3x slower than the fastest. Thus, the other VMs can have enough message send events to it in between one of its slow clock cycles that it can build up a message queue. When this happens, it reads off the messages in a FIFO manner, thus reading messages that were sent a few clock cycles ago. It thus can fall behind the faster clocks, and have to make larger jumps to catch up. In this case, the largest jump that it makes is 15, larger than in the other cases.

Trial 5 is approximately the same system at twice the speed (one clock speed is 3 instead of 4) and thus it shows very similar behavior in terms of jumps and few small message queues forming in the slowest VM.

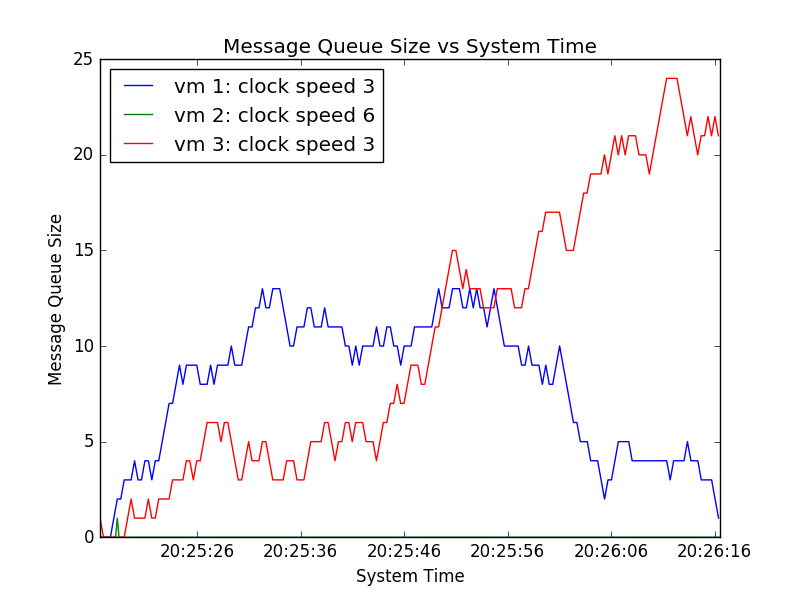
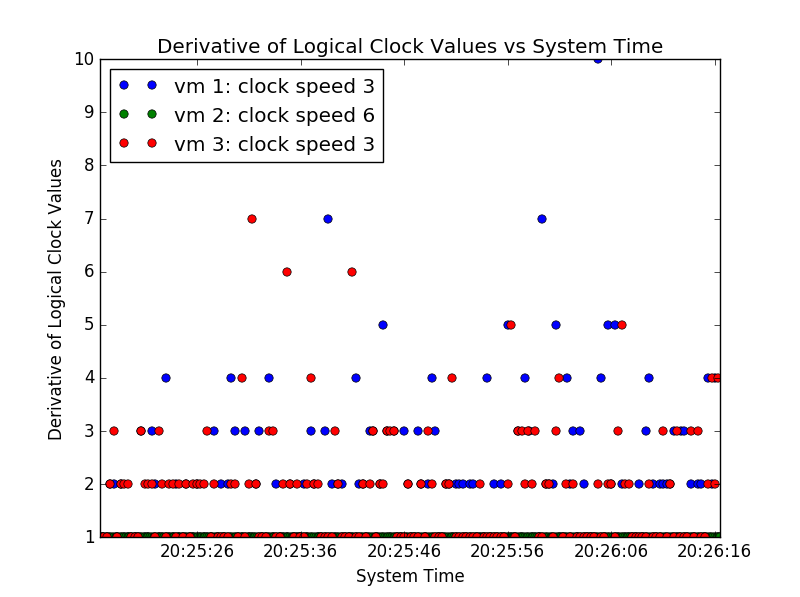
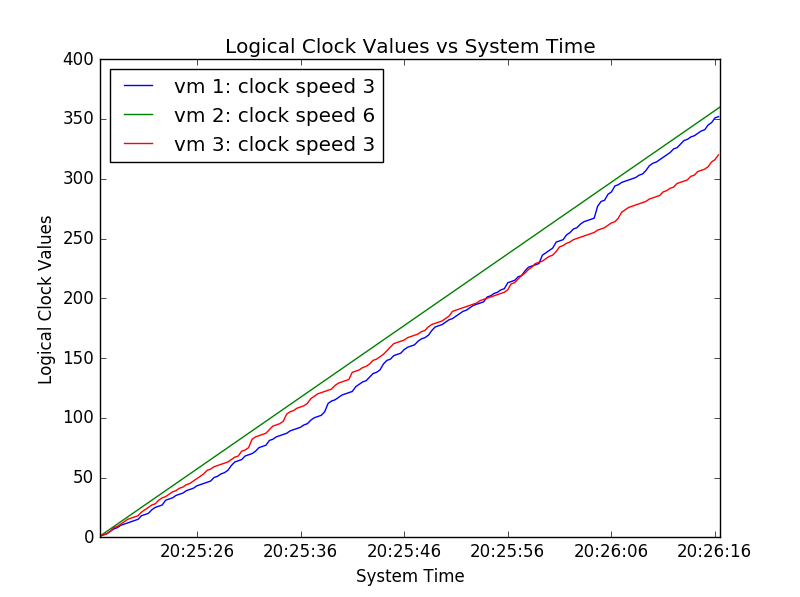
Finally, the most interesting case is trial 1. In all of the previous cases, the differences is speed were handled pretty well, with the clocks staying reasonably close to one another and the jumps never being greater than 15. However, in this case the fastest clock was 6x faster than the slowest. Thus, since messages are sent to a given VM on average every 5 clock cycles, the expected number of messages it sent to the slowest VM per slow VM clock cycle was 1.2 (actually slightly less since it also sometimes receives messages and can’t send on that cycle, but the other two VMs are much slower so receive events will be comparatively rare). The slow VM also gets messages from the middle VM, so the total expectation value of events added to the queue per slow clock cycle is greater than 1. The slow VM can only remove one event per clock cycle, so the queue grows faster than it can empty it. Since it always reads the oldest sent message off of the queue, when the queue is large it reads messages that were sent far more than one slow clock cycle ago in real time, and thus the messages it reads have an LC value far behind the current LC value of the sender. As the queue gets larger, these messages are even older, so the result is a runaway behavior in which the slow VM’s LC drifts farther and farther behind those of the other VMs, in this case leading to a gap of almost 100 in 60 seconds. The result is that, for sufficiently large speed differences, the system will fail to keep all of the VMs synched.

Additional Trial: Reduced clock speed variation



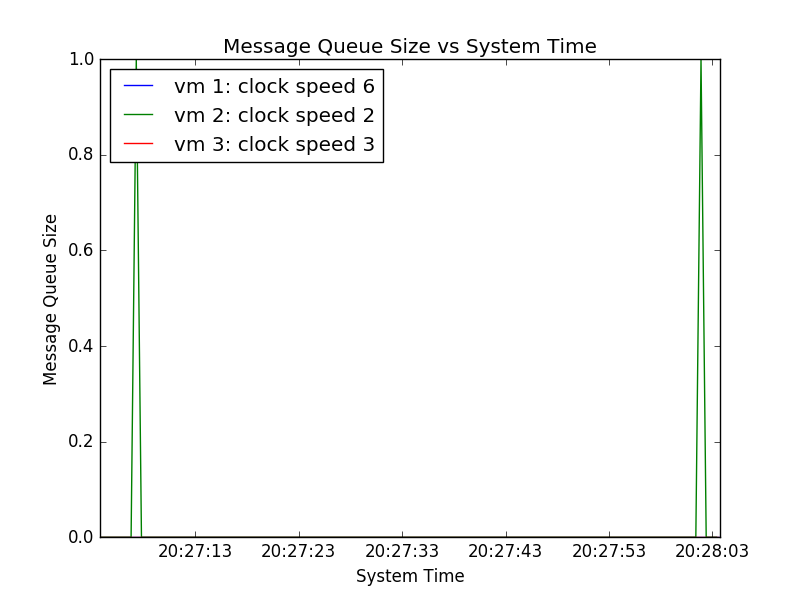
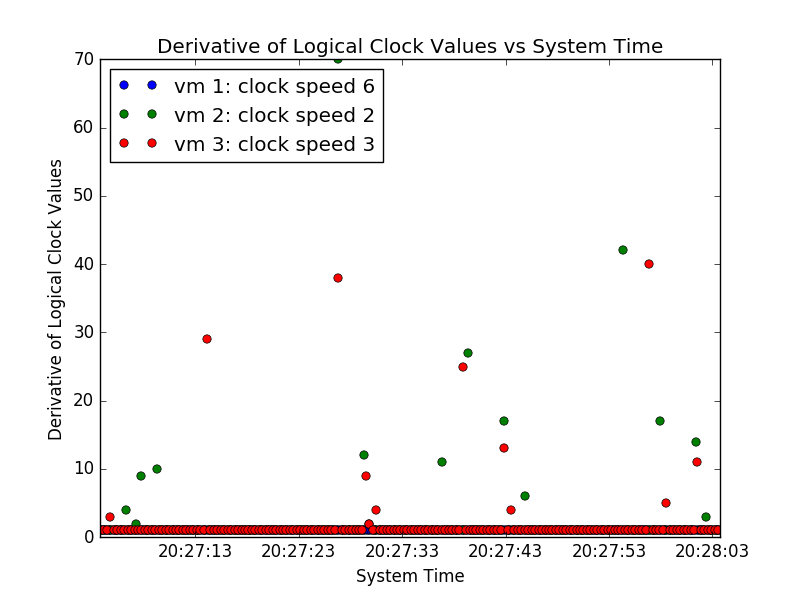
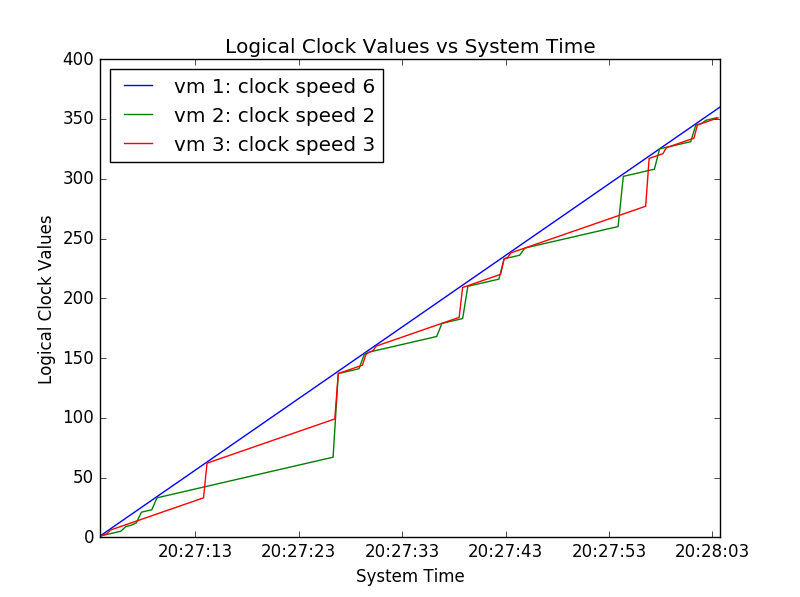
Here were look at the extreme case of reduced speed variation, in which every clock is running at the same speed. In this case, any message received, which must be from an earlier clock cycle, has a lower LC, so every VM only ever takes its own LC. The results is that the VMs LCs are always perfectly synched, and there are no jumps in LC values. This is the expected result, as we have recreated a synchronous system.

Additional Trial: Smaller probability of internal event (1/4 chance)



Consider the case of a smaller probability of an internal event, in this case a ¼ chance with an equal chance of sending to vm1, vm2, or both as in the previous cases. Here, one clock runs at twice the speed of the others and it sends on average one message to each other VM per clock cycle (generally lower than this due to receive events, but here both other VMs are tied up in their message queues and never send). Thus, we expect that sometimes, when due to randomness there are more than average send events, both slower VMs will build up a message queue as they do here. However, the size of this queue and of the jumps will vary based on probability. Here, one of the slow VMs is able to clear nearly the entire queue and resume internal and send operations while the other’s queue has grown to almost 25, despite the two running at the same clock speed. The result is that the former has nearly the same LC value as the fast clock, while the latter still has a gap similar to the one seen in trial 1. Also, note that when there are lots of sends and few internal events, the VMs are in almost constant contact. Thus, even though the slow VMs are reading a past state of the fast VM, the jumps are in general small, with no jump here exceeding 10 and mostly being 2-3.

Additional Trial: Larger probability of internal event (47/50 chance)



Consider this case of mostly internal events, which is the opposite of the previous case. Here, the VMs rarely talk. The result is that there are very few update messages, so in most cycles the LC will just tick up by 1, but once in a while a large jump, often greater than 10 and here as high as 70, will occur that resynchs the LC of that VM to the fastest one before it starts to drift again until the next message. Thus, the jumps are much larger but much less often than in the previous cases as we would expect.