Journal for Scale Model and Logical Clocks

Design

Overview

The entire system runs within the same process to minimize exposure to uncontrolled environment variables. The scale model is run by a controller that launches and manages the three virtual machines, as specified. The three virtual machines themselves are launched by the controller essentially simultaneously, each with their own individual threads.

Controller

The controller facilitates all communications between the virtual machines. All virtual machines, when wanting to send a message, send that message (along with the specified recipient) to the controller, which then passes it along to the appropriate receiving virtual machine. The design decision to use a controller for communication resulted from a desire to minimize complexity of the scale model and the necessary implementation code. It also reduces vulnerability to inherent uncertainties involving networks and socket communications.

Virtual Machines

Each virtual machine runs on a different thread since multi-threading most closely emulates a virtual machine when using a single process, just as a scale model is intended to do.

As specified for the virtual machine, it is initialized with a number representing the clock ticks per (real world) second for that machine. Multi-threading is beneficial here, since each machine running in its individual thread means that clock speed can be simulated by sleeping the thread for the specified time per tick. At each tick, either process the first message in the queue or, if there are no pending messages, a randomly generated value per the specifications determines the machine's actions on whether to send and who to send to. The logical clock is updated accordingly (see below for more details).

Also as specified, each machine has an open network queue for incoming messages that the controller can modify as appropriate. Again, design considerations are targeted towards a tangible model that is simple and clear to use, while deliberately overlooking concerns such as security that should be considered in real settings. Outbound communications as discussed above are handled by the controller.

Each virtual machine opens a distinct log corresponding to their unique IDs. These logs, as specified, track any communication event that happens for that machine (either sending a message or receiving one). Logical clock and real-world clock times are included, which are useful and insightful when performing analysis of the logical clocks.

Logical Clock

As already mentioned, each virtual machine operates based on its initialization clock speed value, while also maintaining its own logical clock. These follow the principle of logical clocks that have been studied before.

- If sending, `clock += 1`
- If receiving, `clock = max(clock + 1, message)` where `message` is conveniently the sender's logical clock value. Take the larger of the two possible values: either a basic increment, or the time of the sending machine. If the sending machine has a larger value, that means the recipient needs to "catch up" in terms of its logical clock time.
- Otherwise, do nothing. Based on our comprehension of logical clock rules, since nothing is happening to the virtual machine relative to the other machines in terms of communication, then the logical clock should not be aware of any liveness in the system and therefore should not update.

Experimentation and Observations

The scale model was run more than five times, each for a minute.

Jumps in Values

The size of the jumps in the values for the logical clocks appeared to be dependent on the operating speed of the virtual machine. When clock is relatively slow (e.g. ticks = 1), a decent amount of jumping occurred, and when clock is relatively fast (e.g. ticks = 6), it runs smoothly at almost perfect +1 increments.

According to the rules of the logical clock, this intuitively makes sense. The machines with faster clocks can process messages faster than other machines could send, allowing the faster machine to send out messages at its own pace and therefore rarely needing to catch up. The opposite is true for slower machines, which will more likely than not have messages continuously queued up faster than they can be processed. Since the other faster machines continue to send out several messages that increment the logical clock by 1, this slow machine would continuously need to catch up and see abrupt forward jumps.

There are even occasional duplicates, which resulted from when both the other machines sent a message simultaneously to this machines.

```
SEND: [global] 2018-03-04 23:30:58.077947; [recipient] 2; [queue length] 0; [logical clock time] 55
```

```
SEND: [global] 2018-03-04 23:30:58.078004; [recipient] 0; [queue length] 0; [logical clock time] 55
```

Drift

When examining drift in the values of the local logical clocks in the different machines, there is certainly drift between relatively slow and relatively fast clocks. In one instance, vm0 (ticks = 1) ended at logical time of 86 in trial0, whereas vm1 (ticks = 6) ended at logical 109. As mentioned before, the slower machines need to jump regularly to catch up, meaning that they perpetually need to catch up. In other words, the faster machines are the ones that "define the speed" by constantly sending messages, while slower ones are "speed takers" and frequently experience a drift from the "true value".

Gaps in Logical Clock and Message Queues

When relative clock speed is low (e.g. ticks = 1), machine is almost always receiving since other machines get ahead of it. This means message queue is almost never empty and usually grows as other (faster) machines fill it. When relative clock speed is high (e.g. ticks = 6), machine is mostly sending, since its clock is fast enough to handle its message queue and keep it near empty.

More Trials

Smaller Time Variations: Choosing a smaller range for times makes the differences more discrete (binary option between 1 clock speed for slow and 2 clock speed for fast). Furthermore, the faster machines will have mostly sends while slower machines are mostly receiving messages. The narrower range of only two speed options makes the aforementioned differences between the slower and faster machines more explicit and discrete.

Larger Time Variations: When machines have larger discrepencies between their clock speeds. With clock speeds of `vm0: 30, vm1: 53, vm2: 12`, vm2 has 117 cycles that are *all* receive entries. vm1 is clearly mostly all sends while vm0 is a very even split between sends and receives. These are more extreme and clearer instances of the previous discussion. The slow machines mostly receive and try to catch up while the faster machines are the dominant time setter with smoother time increments.

No Internal Events: Interestingly, when setting internal events to only send messages, (i.e. generate a value between 1 and 3 inclusive), the fastest machine *solely* sends machines while the two slower machines *entirely* receive messages, essentially exaggerating the logical clock effects that are being examined and have been discussed.

More Internal Events: Again interestingly, with a range of 100 values for the possible machine events, the chance of sending a message is very low, essentially countering the effect of large time variations since the probability of sending a message is low regardless of the clock speed. As a result, the ratio of sends to receives between the three machines despite large time variations of 100 possible clock speeds.