Loggy — A logical time logger

Anton Bothin

September 12, 2019

1 Introduction

The assignment was to use logical time in order to print messages sent to a logger in their correct order. This included creating the logger as well as workers that sent messages to each other and to the logger, here a small delay was introduced to increase the amount of messages sent out of order. The main module of intreset is however the `time` module which allows the logger andworkers to use Lamport time.

2 Main problems, solutions, and evaluation

The implementation consisted of two steps. Firstly I implemented the logger and worker without regards to time. Secondly I introduced logical time via the `time` module.

2.1 Initial implementation

The logger was naively implemented by printing a message as soon as it was received. This ensures a FIFO order for messages if only one worker is involved, but if another worker becomes active this is no longer the case. If a worker A send a message to the logger then one to worker B, worker B could act on that message and send its own message to the logger before the logger has received the message from worker A.

The worker is implemented by waiting for a random amount of time, after this wait it sends a message to a a random worker following by again waiting a random amount of time and then sending a message, {`sending, Msg`}, to the logger. When the worker receives a message it also sends a message on the form {`received, Msg`} to the logger. Each message sent was "unique" in the sense that it contained a random integer between 1 and 100, this was during testing used to identifier matching `sending` and `received` messages.

When running a test (using the provided `test` module) I could easiliy see that messages were out of order by noticing when a `received` message was

printed before the sending message with the same identifier. I also noticed that when specifying a smaller jitter time (the time between sending a message to another process and to the logger) fewer messages are printed in the wrong order. When specifying a jitter time of 0 virtually no messages were printed out of order.

2.2 Implementation using logical time

Implementation of the `time` module used Lamport time, the timestamps were as simple as 0, 1, 2, etc. The module consists of seven simple functions: `zero/0, inc/2, merge/2, leq/2, clock/1, update/3, safe/2`. These are used for abstractions, allowing us to later change the representation of logical time without changing any of the other modules. Most of these functions are very basic, `clock/1` and `safe/2` are however worth explaining.

- `clock(Nodes)` returns a clock structure, the structure is on the form: `[{Node, Timestamp} | Clock]`. `clock/1` simply returns a tuplelist where each node in `Nodes` has its own entry where the timestamp is set to 0.
- `safe(Time, Clock)` returns true or false, depending on if it is safe to print a message with time `Time`. It checks this by checking that there is no timestamp in `Clock` that is lower than `Time`.

The `log` module had to be slightly changed in order to use these timestamps. In the initialization function `time:clock/1` was used to create a clock with each workers name as a node, this is used to keep track of each workers last timestamp. A hold-back queue is also introduced to keep a record of all the messages that have not yet been printed. When the logger received a message, both the clock and hold-back queue are updated. Each entry in the hold-back queue is then visited to see if the message is now safe to print. Each message printed this way is removed from the queue.

Modifying the `worker` module was considerably easier. Each worker keeps track of its own timestamp, when sending a message it also includes this timestamp. When receiving a message the worker should update the timestamp by taking the max of its own time and the time received in the message, and then increaseing this value by one. Each worker should also increment its timestamp before sending a message.

With this implementation the messages were no longer printed in the wrong order. That is to say, the messages were printed in casual order. A problem I encountered with my first attempt at logical time was that some events weren't printed in casual order. I didn't order the hold-back queue in regards to each message's timestamp. This resulted in messages printed in the wrong order if the hold-back queue grew too large. After sorting the

hold-back queue this no longer became a problem. There was another problem I noticed when checking how big the hold-back queue could grow, if one worker happen to never send a message the logger won't be able to print anything. The largest I could get the hold-back queue to while testing was 42 entries.

3 Conclusions

This assignment did a very good job of teaching me how distributed systems handle the ordering of messages. I also understood the benefits and disadvantages of Lamport time, that it is easy to implement but vulnerable to starvation.