Homework 3: Loggy

Patrik Zhong

September 28, 2021

1 Introduction

In this homework assignment we examined the issues with sending and receiving messages in order and how we could deal with that with Lamport timestamp theory. This lab also shows how keeping track of time in a distributed setting is deviously hard.

2 firsts tests

When using a high sleep number in our test and a low jitter number we get things in order. However, when lowering the sleep, things might still be in order, but it wont be as pretty. This is because the functions dont sleep long enough to be able to "catch up".

```
log: na john{sending,{hello,57}}
log: na ringo{received,{hello,57}}
log: na ringo{sending,{hello,77}}
log: na john{received,{hello,77}}
log: na paul{sending,{hello,68}}
```

But as we start to increase the jitter value, here with the value of 1, things start going out of order due to variability of the time it takes to print and process.

```
log: na john{sending,{hello,57}}
log: na john{received,{hello,13}}
log: na paul{sending,{hello,68}}
log: na ringo{sending,{hello,13}}
log: na ringo{received,{hello,57}}
log: na ringo{received,{hello,68}}
log: na ringo{received,{hello,58}}
```

Here we see how John receives message 13, but Ringo sends the message at a print further down below! This "lag" is why we implement the lamport time. If we increase the jitter value, things spiral downwards even faster.

3 LamportTime

To solve this, we want each node to have their "own" logical time they can compare with the time everyone else has, so that the node can wait a bit before they send their message. This is why we implement time.erl. The time.erl API we implements adds a few functions that increment each node's time, and also compare them.

By using these, we're able to implement a very rudimentary "internal" clock. This clock simply sends a counter value to where it says "na" on the previous logs.

```
log: 1 john{sending,{hello,57}}
log: 2 john{received,{hello,13}}
log: 2 paul{sending,{hello,68}}
log: 4 ringo{sending,{hello,13}}
log: 5 ringo{received,{hello,57}}
log: 3 ringo{received,{hello,68}}
log: 1 ringo{received,{hello,58}}
```

We have however once again only implemented a counter on each node. The messages are not in order, nor are they neatly after each other in terms of process order.

4 Implementing the HoldBackQueue

To implement the holdbackqueue, we need to add to our time.erl api. We add clock() that appends an initalised time counter to each tuple, an update() function that updates/replaces tuples and a safe function that simply checks the size of time between nodes.

```
clock(Nodes) ->
        lists:map(fun (Node) -> {Node, zero()} end, Nodes).
```

```
update(Node, Time, Clock) ->
        UpdatedClock = lists:keyreplace(Node, 1, Clock, {Node, Time}),
    lists:keysort(2, UpdatedClock).

safe(Time, [{Name, OldTime}|List]) ->
    leq(Time, OldTime).
```

Finally we implement our holdbackqueue in our main loop in our loggy.erl module. This relatively easy implementation utilizes our new API functions to first update our clock in each message and then browse through each message to see their timestamp and if it's safe to send through. If not, we simply put it into the holdbackqueue.

5 Bonus Task

Implementing the vector clock was relatively straightforward, or so I thought. I realized that I didn't really understand the output I was supposed to get after having implemented my own take on the skeleton code. I thought I understood the representation, but when printing the result:

```
log: [{john,1}] john {sending,{hello,22}}
log: [{john,2}] john {sending,{hello,73}}
log: [{ringo,1}] ringo {sending,{hello,20}}
log: [{paul,1}] paul {sending,{hello,46}}
log: [{ringo,2},{paul,1}] ringo {received,{hello,46}}
log: [{paul,2},{john,1}] paul {received,{hello,22}}
log: [{ringo,3},{paul,1}] ringo {sending,{hello,94}}
log: [{george,1}] george {sending,{hello,51}}
```

The answers aren't directly after the receives, even after they're in order :(.

6 Conclusion

This assignment was very much easier than the last one. Most of the hard work in the assignment came down to understanding the theory behind lamport time. Once that was understood, much of the code really helped, especially with the API skeletons being given. Debugging the code was rather forgiving due to how you could very easily see if things were out of order. The hardest part was probably the implementation of the holdbackqueue. It took me awhile until I REALLY thought exactly about how I wanted to build up my queue.