Report 3: Homework Report loggy

Marcus Samuelsson

September 27, 2023

1 Introduction

In this report the challenge of handling order when message passing in a distributed system is shown and the solution is implemented. This issue is tackled with the use of Lamport timestamps, which works as a logical clock to keep track of the order between events.

2 Main problems and solutions

The problems with this task was a lot fewer then previous tasks, mainly because *Erlang* has become more familiar and the amount of code is less then the last task. The biggest issue was to grasp how the *Lamport clock* works and should be implemented. This issue was overcome through interpreting the instructions given in the lecture and the accompanying notes. Otherwise the only issue was that the updated time was not being passed to the next loop, causing the numbers to not increase properly, which was fixed through testing until the issue was found.

The most confusing part was how to implement the **safe()** function in the **time** module. This was because the instructions for it was somewhat vague and figuring it out required looking deeper into the Lamport clock.

3 Evaluation

3.1 First implementation

In the first implementation that was given through the instruction, it was possible to see how the order of entries where in the wrong order when the *jitter* was increased. This could be seen as the messages where assigned a random number and the **received** message was sometimes logged before the **send** message of the same number. This is also possible to see after implementing the time counters(see Figure 1).

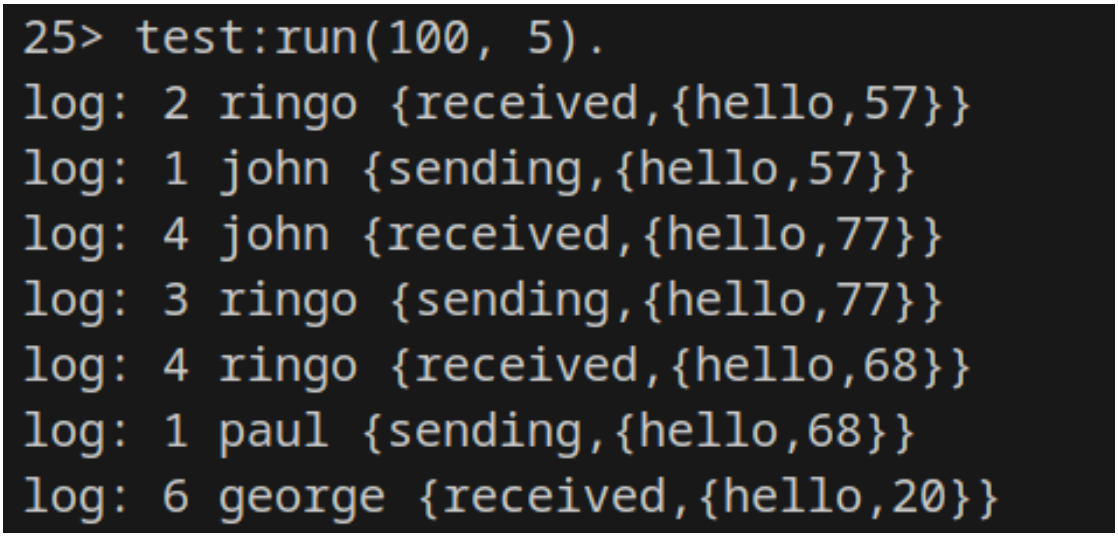

Figure 1: Run of the first implementation with time counter of loggy

3.2 Implementation with Lamport clock

This project is built on three different modules **loggy**, **worker** and **time**. The **Loggy** module was origianally supposed to be called "logger", however this resulted in the error "sticky dir" because it is already a module in *Erlang* so it was changed. This handles the updates of the clock and the order of the logs through a queue. This makes sure that every message is printed in the correct order. **Worker** keeps track of the logical time and updates it as it sends and receives messages. Time is the module will be more thoroughly explained in Section 3.2.1.

3.2.1 Implementation of time.erl

The **time** module consists of the following functions:

- **zero()** returns the initial *Lamport value*(In this case zero).
- **inc(_Name, T)** return the time **T** incremented by one.
- **merge(Ti, Tj)** returns the largest of **Ti** and **Tj**.
- **leq(Ti, Tj)** true if **Ti** is less than or equal to **Tj**.
- **clock(Nodes)** return a clock that can keep track of the nodes with the initial time value from **zero()**.
- **update(Node, Time, Clock)** return a clock that has been updated given that we have received a log message from a node at a given time. Through deleting the current entry and adding a new entry with the new **Time**.

- **safe(Time, Clock)** returns true if the current node has the smallest time compared to all the other clocks.

3.2.2 Result of new implementation

When the **time** module is implemented the results show that the output is in the correct order with the time of each call. In this implementation it can be seen that the jitter does not effect the output as nothing receives a message before it is sent and the numbers just increase for each node(see Figure 2).

```
27> test:run(100, 5).
log: 1 john {sending,{hello,57}}
log: 1 paul {sending,{hello,68}}
log: 2 paul {sending,{hello,20}}
log: 2 ringo {received,{hello,57}}
log: 3 paul {sending,{hello,16}}
log: 3 ringo {sending,{hello,77}}
log: 4 ringo {received,{hello,68}}
log: 4 john {received,{hello,77}}
log: 5 john {received,{hello,20}}
log: 5 ringo {sending,{hello,20}}
log: 6 ringo {sending,{hello,97}}
log: 6 john {sending,{hello,84}}
log: 6 george {received,{hello,20}}
log: 7 george {received,{hello,84}}
log: 7 john {sending,{hello,7}}
log: 8 paul {received,{hello,7}}
log: 8 george {received,{hello,16}}
log: 8 john {sending,{hello,23}}
log: 9 john {sending,{hello,36}}
log: 9 paul {sending,{hello,60}}
log: 9 george {received,{hello,97}}
log: 10 john {sending,{hello,81}}
log: 10 paul {sending,{hello,79}}
```

Figure 2: Run of the final implementation

4 Conclusions

This task was much easier then the previous when it came to using *Erlang* and the amount of code that had to be implemented. The most difficult parts where understanding *Lamport clocks*, but this was solved through the material in the lecture around this subject.

Finally, this gave a better understanding of what issues come with message parsing and how it could be solved for distributed systems.