Homework Report 3

Riccardo Fragale

September 24, 2025

1 Introduction

In this homework we were asked to implement a logging procedure that receives log events from a set of workers. The worker will be very simple; it will wait for a while and then send a message to one of its peers. While waiting, it is prepared to receive messages from peers. To keep track of what is happening and in what order things are done, we send a log entry to the logger every time a worker sends or receives a message. The events are tagged with the Lamport time stamp of the worker and the events must be ordered before being written to stdout. Each worker is given a unique name and access to the logger. We also provide a sleep and jitter value; the sleep value will determine how active the worker is sending messages, and the jitter value will introduce a random delay between the sending of a message to a worker and the sending of an entry to the logger.

2 My implementation

I implemented the *time* module. It contains 4 basic functions:

- `zero()`: return an initial Lamport value
- `inc(Name, T)`: return the time T incremented by one
- `merge`(T_i, T_j): merge the two Lamport time stamps
- `leq`(T_i, T_j): true if $T_i \leq T_j$.

The logger has a clock that keeps track of the timestamps of the last messages seen from each worker. It should also have a holdback queue (module **holdb_queue**), where it keeps log messages that are still unsafe to print. When a new log message arrives, the clock is updated, the message is added to the holdback queue and then we go through the queue to find messages that are now safe to print.

Inside the module time I implemented other three functions (called *clock*, *update* and *safe*. I show below the code of the function **safe**.

```
safe(Time,Clock)->
    lists:all(fun({_Node,T}) -> Time =< T end,Clock).
```

This implies that a message is considered safe if the Lamport Time of the message received is lower or equal with respect to the Lamport time of the clock. This function is used to partition the holdback queue between safe and unsafe messages every time the logger loops. In each iteration, the logger is printing only safe messages while it is maintaining in the queue the unsafe messages that will be printed in future loops when they will eventually become safe.

This, instead, is the function used to update the clock when a message is received by the logger.

```
    update(Node, Time, Clock)->
        OldTime = case lists:keyfind(Node,1,Clock) of
                    {Node,T} -> T;
                    false -> 0
                end,
        [{Node, merge(OldTime,Time)} | lists:keydelete(Node,1,Clock)].
```

I first check the presence of the node in the clock and then apply the function merge to identify the correct value for the Node inside the list where the clock is kept. Found the correct timestamp, the new tuple {Node, NewTime} is simply added to the clock list.

3 Tests

First of all I decided to run a script (*detect.py* to verify automatically whether there were errors in the ordering. The logic behind the detection is quite simple; if the logger sees a message of type *sending* with a lower timestamp with respect to the same message of type *received* this is for sure an ordering violation. I did some tests incrementing a lot the jitter and I was able to detect some errors somewhere when the jitter is very high (like 40 or 50). In this case the violation of the ordering is very subtle. In the final part of the logger life(just before receiving a stop) it receives a message of type *receiving.* Then, since it is stopped, the logger doesn't receive nor print a message of type *sending.*

Log Output

```
127,john,{sending,{hello,56}}
....
130,paul,{sending,{hello,100}}
130,george,{received,{hello,56}}
131,john,{received,{hello,100}}
132,john,{sending,{hello,59}}
133,paul,{received,{hello,59}}
135,george,{received,{hello,3}}
```

As it is easily predicted, simply reducing a lot the jitter brings down the probability of having such an error to almost 0.

I tried also to test the maximum dimension of the holdback queue. I did a test where the system was on for two minutes with Sleep=50 and Jitter=10. The graph below shows how the queue length vary in all the loop of the logger. Actually the maximum value measured is 39 and the average one is around 10.

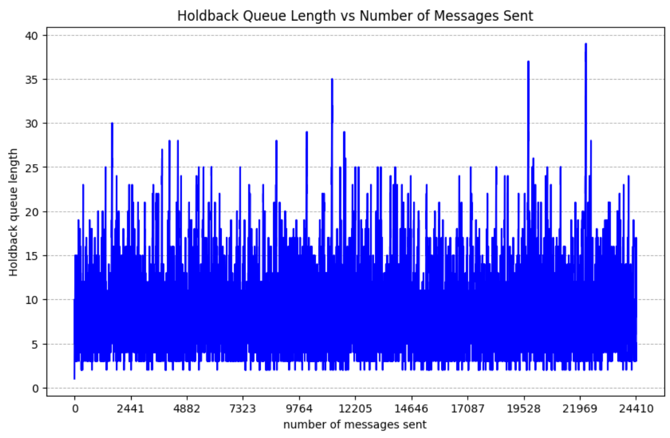

Figure 1: Queue length variation over logger loops.

I tried other tests increasing the time in which the system stays on, and also reducing the worker's sleeping time, but still the maximum value of the holdback queue length doesn't increase to more than 40/45.

4 Optional task: Vector clocks

I implemented a new module called **vect** where I included all the functions already implemented in the *time* module but refactored to deal with vector clocks. The vector clock is a map that keeps track of the logical time of each node in the system. The main difference is just that a single node needs to verify the clocks of every node when an incoming message is received and then update the clock of itself. I also wrote a script to detect possible errors, which initially happened because I was not sorting correctly the messages in the holdback queue of the logger. Then everything started to work properly. I observed that both the average and the maximum queue length is reduced a lot with the usage of the vector clock timing. This means that, in general, since the probability of wrong ordering is nearly the same, I would always go for a system with a vector clock timing since the memory resources required for the logger is heavily reduced.