Report 3: Loggy

Corentin LEROY

September 26, 2018

1 Introduction

The goal of this assignment was to implement logical time in a logging procedure. We have different worker processes that send messages to each other randomly. Each time a message is sent or received by a worker, this worker sends a timestamped log entry to the logger. To simulate real-life networks latencies and generate unordered logs, we introduced some random delays in worker processes before sending/receiving a message and before sending a log entry to the logger. That way, we can assure that the logger will not receive all the log entries in the right order. To overcome this problem, we used Lamport clocks.

2 Main problems and solutions

I will here describe the `time` module that contains all the functions used to create and update a logical time representation. The first 3 functions are only called in the `worker` module. Their purpose is to treat time values, incremented each time a worker sees an event (sends or receives a message). Each worker has a time value that represents its vision of the global state of the system.

- `zero/0`

  This function initiates a time counter (with a value of 0).

- `inc/2`

  This function increments the time value of the worker. It is called each time a message is sent or received by the worker.

- `merge/2`

  This function keeps the most up-to-date time value between the one owned by the worker and the one that it receives from another worker, that is, the highest value of both.

- `leq/2`

  This function is just a comparison test to determine if a time value is higher than another one.

The 3 other functions are used in the `loggy` module to manage a clock, which gives a global time representation of the system.

- `clock/1`

  This function creates a clock, that is a list containing a tuple for each worker. Each tuple contains the name of the worker, and the time value that it owns and that it has sent to the logger in a log entry.

- `update/3`

  This function updates the time value of the corresponding node in the clock. It is used by the logger when it receives a log entry from a worker.

- `safe/2`

  This function is to know if a log entry can be printed by the logger or not, given its timestamp. As explained in the introduction, the logger may receive log entries in a wrong order. To print them out in the right order, it evaluates with this function if the log entry is safe to print at a specific moment of time. If yes, the entry is printed, if not, it is put in a sorted queue where it will wait until it is safe to be printed. The biggest difficulty of this seminar work was to find and express the right condition for a log entry to be safe. At first, the condition I wrote was that its time was inferior or equal to the minimum time of the whole clock. But I realized the logger sometimes waited to print a log, even if it was actually safe to print it. That was due to the fact that at the beginning, some times in the clock are still at 0 because the workers have not sent or received messages yet. So the logger had to wait until all workers had been active to start printing logs. With the initial sleep value of 5000, and for example 2000 for the sleep and the jitter values, only 2 or 3 logs had time to be printed. Therefore I thought something was wrong, as a lot of time was wasted by the logger waiting. To avoid that I tried to take the strictly positive minimum if there was one. But then I had a few logs that were not in order. I finally realized that there was no better way to safely check that a log could be printed than to compare its timestamp with the minimum time value in the clock, be it 0. So I just increased the sleep value before stopping the workers, and I could see that indeed it was working well, and all the logs were in order.

3 Evaluation

The workers update their time value only when sending or receiving messages, so the logs' times may not be accurate to indicate the global time. But we can know if logs are not in order thanks to the number sent in the messages' body. For example if the logger prints `john {received,{hello,68}}` before `ringo {sending,{hello,68}}`, we know for sure that these two logs are not in the right order. Before implementing the Lamport time, of course, many logs were printed in a wrong order. But once logical time was added, everything was printed in chronological order. Or at least, in an order that is coherent with the real chronological order. In fact, we are sure that all the events described by printed log entries have happened in that exact same order, except for the logs with the same timestamp. Let us take these logs for example:

```
log: 1 john {sending,{hello,57}}
log: 1 paul {sending,{hello,68}}
log: 1 george {sending,{hello,58}}
log: 2 ringo {received,{hello,57}}
log: 2 george {sending,{hello,100}}
log: 3 ringo {sending,{hello,77}}
log: 4 john {received,{hello,77}}
```

Here we cannot be absolutely sure that `john` sent a message before `paul` or that `ringo` received a message before `george` sent one. However, we are sure that no log for a message reception will be printed before a log for the sending of the same message.

Furthermore, I tried to see the impact of the sleep and jitter values on the length of the hold-back message queue. I printed the maximum length obtained each time with a simulation time (sleep value in the `test` module) of 10000 ms.

sleep	jitter	Max hold-back queue length
1	1	41
10	1	43
100	1	30
1000	1	25
5000	1	10
10000	1	10
1	10	41
1	100	30
1	1000	22
1	5000	15
1	10000	10
10	10	41
100	100	30
1000	1000	21
5000	5000	12
10000	10000	5

Table 1: Maximum hold-back queue length evolution with sleep and jitter values

We can see that the biggest queues are obtained for small sleep and jitter values.

4 Conclusions

This seminar was a good way to experiment with logical time and understand its usefulness. It allows to order events without having to synchronize clocks, and Lamport time is one solution. This logger procedure was a simple example and yet I could see it was not trivial to order everything the right way.