Report 3: Loggy, a logical time logger

Simon Cathébras

September 20, 2012

1 Introduction

1.1 Context

During this assignment, we have manipulated logical time, and more specifically Lamport's clock.This assignment has highlighted that, more than the time itself, the ordering of events is a critical issue in distributed systems. Moreover, measuring precisely the time, is something very difficult to do in computer science. And thus, trying to synchronize several component of a distributed system with physical time, is impossible to do in real life. That's why Lamport's logical time only relies on event's ordering. But, even if this concept seems easy to understand, implementing it can be a little bit tricky.

1.2 What have been realised

We implemented the Lamport's clock mechanism for the logger. The worker are keeping trace of their own state, and this state is updated each time a message is received. The Logger keeps a queue of message "not safe to print yet", and a list containing each workers and their respective states. A message is printed out of the queue when his `TimeStamp` is less than the minimum of worker's state. The output seems correct. Unfortunately, we could only verify this correction ourselves, instead of writing a program to do so.

2 Main problems and solutions

Spot wrong ordering Our first main problem was to spot wrong ordering in the first implementation. We solved this when we realised that when a message was displayed as `received` before be displayed as `sent`, then, there is a problem somewhere...

Lamport Clock Implementing the change of states inside each processes was not really a problem. As a matter of fact, the problem was, as the logger: "When can I write a message safely ?". We first solved this in realising that a message can

be displayed when we are sure to not receive a message with an older `TimeStamp`. Then, we needed a way to be sure that for a given `TimeStamp`, we would not receive a message with an inferior `TimeStamp`. We solved this with the following:

- The logger keeps tracks of the states of each process with a list of t-uples: `{ProcessName, Time}`
- The logger has a `CurrentStamp`.
- Whenever the logger receives a message, the state list of all processes is updated. Then it checks if the message's `TimeStamp` is bellow or equal to `CurrentStamp`. If so, the message is displayed right now. If not, the message is stored in the `Queue` and we are looking if the minimum Time of the state list of processes is greater than `CurrentStamp`. If so, we update the stamp and we flush the `Queue` of all messages with a `Stamp` less than the new `CurrentStamp`. Note, that before we drain the Queue, it MUST be ordered.

Efficiency and reliability After we found out how to implement the Lamport's ordering, we wondered if our solution was correct concerning efficiency and resources consumption. That's why we implemented a function to check the size of the list during the execution of the scenario. Here are the results we observed.

Number of Processes	5	10	15	20
Maximum size of the queue	20	56	129	193

It appears that this implementation is very greedy in resources when there is an increase of the number of clients (See the graphic in Annexe). I look forward to discuss it during the seminar.

3 Conclusion

This assignment raised several issues about time management inside of distributed systems. The thing is, that implementing a Lamport's ordering is slightly more tricky than it appears at the first look.

4 Annexe

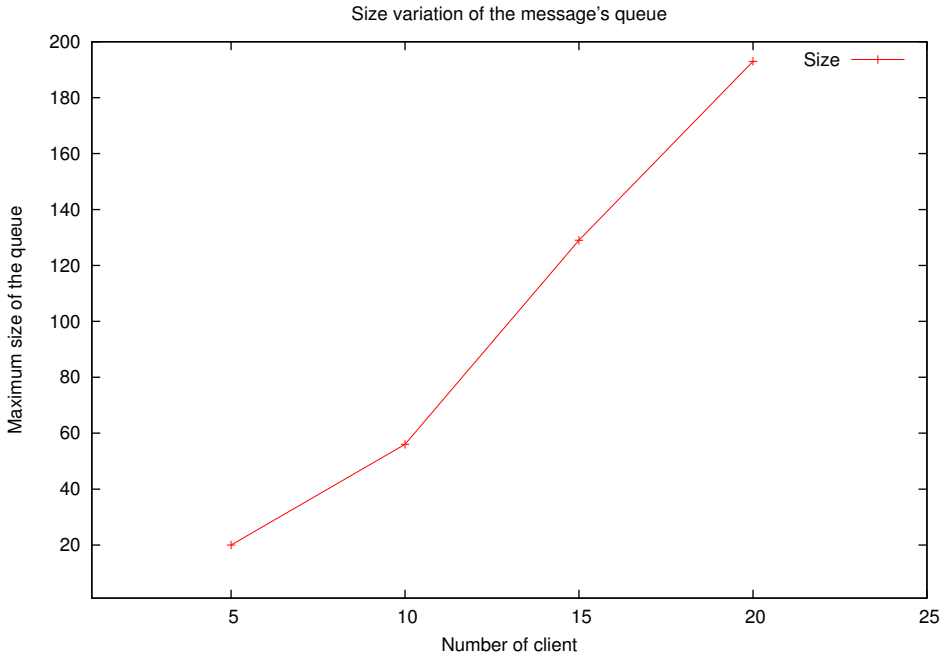

Figure 1: Length variation depending on number of workers