Seminar Report
Loggy - A logical time logger

Pierre FLEITZ

September 29, 2015

1 Introduction

The main goal of this exercice was to learn how to use logical time in a practical example : a message logger. A logging procedure receives messages from workers processes. All the messages are tagged with the Lamport timestamp of the message and the tricky part is to manage to have all the events ordered before written to stdout.

2 Main problems and solutions

As it is said in the subject, it is ”slightly more tricky than one might first think” because we have to create the architecture which is able to order the received messages.
Despite a lot of time I spent on the problem I haven’t been able to find a way to code in Erlang what I had in mind, but I will discuss this later in the last part of the report.

3 Evaluation

3.1 Chapter 3: Test

If we run the method test:run(4000,500) given in the third part, messages displayed are not ordered. Indeed, some ”received” messages are logged before ”sending” messages.
This is an example of a result we could have :

```
Eshell V7.0  (abort with ^G)
1> test:run(4000,500).
log: na ringo {received,{hello,57}} // received before the
    sending.
log: na john {sending,{hello,57}} // here is the sending
log: na john {received,{hello,77}}
log: na ringo {sending,{hello,77}}
```

```
log: na ringo {received,{hello,68}}
log: na paul {sending,{hello,68}}
log: na george {received,{hello,20}}
log: na john {received,{hello,20}}
log: na paul {sending,{hello,20}}
log: na ringo {sending,{hello,20}}
log: na george {received,{hello,84}}
log: na john {sending,{hello,84}}
log: na george {received,{hello,16}}
log: na paul {sending,{hello,16}}
log: na paul {received,{hello,7}}
log: na george {received,{hello,97}}
log: na ringo {sending,{hello,97}}
stop
```

We can explain this result with two reasons :

1. The logger process doesn't have a procedure to order messages yet. When the logger receives a message, it logs it immediately.
2. Because of the jitter, the "sending" message arrived after the "receiving" message.

I ran some tests to see if when you decrease the jitter you can eliminate the problem and I haven't be able to do so expect obviously if you put the jitter to 0 but then you don't have a concrete example of what can happen in a real situation.

3.2 Chapter 4: Lamport Time

The point of this chapter was to add a logical timestamp into the messages sent by the worker process.
In order to do this and to be able to compare our solutions, we implemented the handling of the Lamport clocks in a separate module time.

Nothing in particular to say about module time, expect that we used erlang:max(A,B) for the merge(Ti,Tj) function.

After we implemented this module we had to change the worker process to add logical timestamp using this API. To sum up we changed the init by adding an initial Lamport value in the argument (time:zero()) and we improved the loop function to add the timestamp.

The timestamp is calculated according to Lamport theory by getting the max between the local time and the time of the received message, and by increasing by 1.

```
{msg,Time,Msg} ->
      NewTime = time:merge(Counter,Time)+1,
      Log ! {log, Name, NewTime, {received, Msg}},
      loop(Name, Log, Peers, Sleep, Jitter, NewTime);
```

The console result is shown below:

```
19> test:run(10000,500).
log: 2 ringo {received,{hello,57}} <- Wrong order
log: 1 john {sending,{hello,57}} <- Wrong order
log: 4 john {received,{hello,77}}
log: 3 ringo {sending,{hello,77}}
log: 4 ringo {received,{hello,68}}
log: 1 paul {sending,{hello,68}}
stop
```

We can see that the messages are still in a wrong order. We know it because the first log to be displayed shouldn't have a timestamp higher than the second one, here we can see that it happens several times. Also, like in the chapter 3 we can see that we received things before sending them, that is because we didn't improve the logger yet. However, by having logical time into the workers process, we ensure that the intra-process messages are ordered.

3.3 Chapter 4.1: The tricky part

This is the tricky part of the seminar, and I have to admit that I haven't been able to implement all the rest of the time module and that I stayed quite stuck here. It is really frustrating since I know what I am supposed to do, I understand where we want to go but I don't know how to code it in Erlang.

My plan was to have a list of workers, and each worker in that list would have a list of messages that contains the logical timestamp and the contents of the message. When a message is received we save the message into the list associated to the worker that sent the message.
For the safe function : for me it is safe to print a message if when we look inside all the lists of workers and they contain at least one message, then we can display the first messages : we look into the lists and we display the message that have the smaller timestamp. And we do this until one of the worker has no messages waiting in the queue.
The problem is that I really had some trouble to find a way to do this in Erlang...

4 Conclusions

That exercice is a good way to understand that even if concurrency is nice, it eventually brings some problems compared to a single process system.
In a personal point of view it is quiet frustrating that I haven't been able to end this seminar by myself knowing deeply that I do understand what I am supposed to do, I feel like my lack of experience in Erlang is responsible for this, but it is also a question of time because I know that if I was able

to spend as many time on this seminar that on the last one I would have managed to do it. I do hope that I will be able to understand during the seminar and that I will have access to a correction so I can work on this again.
Despite all of this, since I tried hard, I learnt a lot of things about the Erlang programmation again but also I have been able to put so pratical tests on the tricky logical time things we talked during the lecture.