Report 3: Loggy - A logical time logger

Emil Ståhl

September 25, 2019

1 Introduction

This report covers the implementation of a logical time logger which is a mechanism for handling the order of events that happens in a distributed system. Such systems may have no physically synchronous global clock, so a logical clock allows global ordering on events from different processes. This is achieved by so called Lamport timestamps. The main topic covered in this work is to get a deeper understanding of the problems that message passing results in and how they can be solved.

2 Main problems and solutions

The main challenge to finish this work was to understand how Lamport timestamps work. This was mainly done ny studying lecture notes as well as *Distributed Systems: Concepts and Design* (5th Edition) by George Coulouris. Given below is a presentation of the initial modules and their functions.

2.1 Initial modules

Loggy

This module was firstly named ”Logger” but that resulted in the error `Can't load module 'logger' that resides in sticky dir`, probably because that is a built library or so in Erlang. It was therefore renamed to ”Loggy” which also is the name of the assignment. However, the loggy module mainly accepts events and prints them on the screen.

Worker

The purpose of the worker module is to send and receives events from the peers. It also adds a sleep and jitter value to better simulate a real network; the sleep value will determine how active the worker is sending messages, the jitter value will introduce a random delay between the sending of a message and the sending of a log entry.

Test

Sets up a simple test to benchmark the system. However, the timer value of 5000 was changed to be given as an argument to the method in order to better control the test from the terminal.

2.1.1 Test of initial modules

The initial code described above was tested in a separate branch called ”Loggy and Worker” and it could be seen that with the jitter value set to 500 messages got printed in the wrong order. However, with the jitter value set to zero the entries that went wrong was eliminated. This could be seen through that messages were given a random number for identification and received messages where wrongly printed before its sent counterpart.

```
3> test:run(2000, 500, 3000).
log: na ringo {received,{hello,57}}
log: na john {sending,{hello,57}}
log: na john {received,{hello,77}}
log: na paul {sending,{hello,68}}
log: na ringo {sending,{hello,77}}
log: na ringo {received,{hello,68}}
log: na john {received,{hello,20}}
log: na george {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}}
log: na john {sending,{hello,7}}
stop
```

Wrong order

```
2> test:run(2000, 0, 3000).
log: na john {sending,{hello,57}}
log: na ringo {received,{hello,57}}
log: na ringo {sending,{hello,77}}
log: na john {received,{hello,77}}
log: na paul {sending,{hello,68}}
log: na ringo {received,{hello,68}}
log: na george {sending,{hello,58}}
log: na ringo {received,{hello,58}}
log: na ringo {sending,{hello,20}}
log: na george {received,{hello,20}}
log: na paul {sending,{hello,28}}
log: na john {received,{hello,28}}
log: na paul {sending,{hello,43}}
log: na ringo {received,{hello,43}}
log: na george {sending,{hello,100}}
log: na ringo {received,{hello,100}}
log: na john {sending,{hello,90}}
log: na paul {received,{hello,90}}
stop
```

Correct order

2.2 Implementing Lamport Time

In order to solve these problems described above in section 2.1.1 Lamport timestamps was implemented in the initial modules. By Lamport's definition each process / worker is given its own logical clock which is represented as a simple counter. To be able to capture the so called happened-before the following steps are taken; All the process counters start with value 0. A process increments its counter for each event (internal event, message sending, message receiving) in that process. When a process sends a message, it includes its (incremented) counter value with the message. On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one. Now it's possible to check the order in which the events happened. An events timestamp shall be smaller if it happened before the other events timestamp it was compared to. The Lamport timestamp was implemented in a module called time, the API was given in the manual and has been implemented as follows:

```
-module(time).

-export([zero/0, inc/2, merge/2, leq/2, clock/1, update/3, safe/2]).

zero() -> 0.

inc(_Name, T) -> T + 1.

merge(Ti, Tj) -> max(Ti,Tj).

leq(Ti,Tj) ->

      case Ti =< Tj of

            true -> true;

            false -> false

      end.

clock(Nodes) ->

      lists:foldl(fun(Node, Acc) -> [{Node, zero()} | Acc] end, [], Nodes).

update(Node, Time, Clock) ->

      UpdatedClock = lists:keyreplace(Node, 1, Clock, {Node, Time}),

    lists:keysort(2, UpdatedClock).

safe(Time, [{_,X}|_]) ->

    leq(Time, X).
```

- zero() : return an initial Lamport value
- inc(Name, T) : return the time T incremented by one
- merge(Ti, Tj) : merge the two Lamport time stamps (i.e. take the maximum value)
- leq(Ti,Tj): true if Ti is less than or equal to Tj
- clock(Nodes): takes a node list and returns a list of tuples with node names and zero set counters
- update(Node, Time, Clock): updates a time stamp for a specified node in a clock initially returned from clock/1
- safe(Time, Clock): should return **true** if it is safe to log an event with given time and otherwise **false**

Moreover, the loggy module was also updated with a holdback queue where it keeps log messages that are still not safe to print. When a new log message arrives it should update the clock, add the message to the hold-back queue and then go through the queue to find messages that are now safe to print.

2.2.1 Secondary test

Now a secondary test was performed with the newly implemented time module. The results is given below:

```
4> test:run(2000, 500, 7000).
log: 1 ringo {sending,{hello,1}}
log: 1 george {sending,{hello,69}}
log: 2 george {sending,{hello,64}}
log: 2 paul {received,{hello,1}}
log: 2 ringo {received,{hello,69}}
log: 3 george {sending,{hello,88}}
log: 3 ringo {sending,{hello,52}}
log: 3 john {received,{hello,64}}
log: 4 john {sending,{hello,75}}
log: 5 paul {received,{hello,75}}
log: 5 john {sending,{hello,56}}
log: 6 paul {received,{hello,52}}
log: 7 paul {received,{hello,56}}
log: 8 paul {sending,{hello,4}}
log: 9 paul {sending,{hello,39}}
log: 9 ringo {received,{hello,4}}
log: 10 paul {received,{hello,88}}
log: 10 john {received,{hello,39}}
log: 10 ringo {sending,{hello,98}}
log: 11 paul {sending,{hello,76}}
log: 11 john {received,{hello,98}}
log: 12 paul {sending,{hello,86}}
log: 12 john {sending,{hello,39}}
log: 13 george {received,{hello,39}}
log: 13 ringo {received,{hello,86}}
log: 14 ringo {sending,{hello,6}}
log: 14 george {received,{hello,76}}
```

Correct order achieved with Logical timestamps

This time all the messages are printed as sent before they are received, thanks to the timestamps and holdback queue.

2.3 Bonus task - Vector clock implementation

One drawback of using Lamport time is that if a process has a higher timestamp than another one we still cannot be sure if that event occurred before or after the other event it's compared to. However, it works in our case since our program only does send and receive operations between the workers.

The vector clock solves this problem due to that is has all the clock values from other processes and not just the sending process. This way we now know that if one vectors values are smaller than all values of another vector the event will have to have happened before the other.

This was implemented in a separated module called *vect*. Its API is exactly the same as the time module, just differently represented inside it.

2.3.1 Test with vector clock implementation

A third test was performed, this time with the vector module. The results are given below:

```
2> test:run_vect(2000, 500, 5000).
log: [{john,1}] john {sending,{hello,89}}
log: [{paul,1},{john,1}] paul {received,{hello,89}}
log: [{george,1},{ringo,1}] george {received,{hello,66}}
log: [{ringo,1}] ringo {sending,{hello,66}}
log: [{john,2}] john {sending,{hello,66}}
log: [{ringo,2},{john,2}] ringo {received,{hello,66}}
log: [{ringo,3},{john,2}] ringo {sending,{hello,66}}
log: [{john,3},{ringo,3}] john {received,{hello,66}}
log: [{george,2},{ringo,1},{paul,2},{john,1}] george {received,{hello,24}}
log: [{paul,2},{john,1}] paul {sending,{hello,24}}
log: [{george,3},{ringo,4},{paul,2},{john,2}] george {received,{hello,26}}
log: [{ringo,4},{john,2}] ringo {sending,{hello,26}}
log: [{john,4},{ringo,3},{paul,3}] john {received,{hello,84}}
log: [{paul,3},{john,1}] paul {sending,{hello,84}}
log: [{john,5},{ringo,3},{paul,3}] john {sending,{hello,34}}
log: [{george,4},{ringo,4},{paul,2},{john,2}] george {sending,{hello,89}}
log: [{ringo,5},{john,2},{george,4},{paul,2}] ringo {received,{hello,89}}
log: [{ringo,6},{john,5},{george,4},{paul,3}] ringo {received,{hello,34}}
log: [{paul,4},{john,1}] paul {sending,{hello,61}}
log: [{ringo,7},{john,5},{george,4},{paul,4}] ringo {received,{hello,61}}
log: [{george,5},{ringo,4},{paul,3},{john,6}] george {received,{hello,66}}
log: [{john,6},{ringo,3},{paul,3}] john {sending,{hello,66}}
log: [{paul,5},{john,5},{ringo,8},{george,4}] paul {received,{hello,6}}
log: [{ringo,8},{john,5},{george,4},{paul,4}] ringo {sending,{hello,6}}
log: [{paul,6},{john,5},{ringo,9},{george,4}] paul {received,{hello,67}}
log: [{ringo,9},{john,5},{george,4},{paul,4}] ringo {sending,{hello,67}}
log: [{john,7},{ringo,3},{paul,3}] john {sending,{hello,63}}
log: [{paul,7},{john,7},{ringo,9},{george,4}] paul {received,{hello,63}}
log: [{george,6},{ringo,4},{paul,3},{john,6}] george {sending,{hello,43}}
log: [{john,8},{ringo,4},{paul,3},{george,6}] john {received,{hello,43}}

Size of Holdback Queue: 0
stop
```

Correct order achieved with vector clock implementation

This time the order is correct as well as that the message is received directly after it was sent. With the previous implementation, other message was sent out before the sent message was received. The holdback queue is also empty due to that it always knows the order of all messages, so they are safe to be printed directly.

3 Evaluation

The evaluation of this work is that with only the loggy and worker module, messages got printed in the wrong order when introducing a jitter value as low as 10. However, when testing with Lamport timestamps the sending and receiving order was preserved but with the minor drawback that messages is being printed as received after other messages have been sent. The vector clock implementation solves both this issues and prints message in the correct order as well as directly after sending.

It was observed that if one process/worker is idle and not involved with the other ones; no message will ever be ”safe”.

4 Conclusions

This assignment was easier than the last one but definitely not easy. It required a deep understanding of the Lamport timestamp theory. However, I appreciated that the whole API was described in the manual, so the flow of the program was pretty much clear from the beginning. The hardest part was to understand what was safe to print and what happened when log events with different timestamps got ”safe” at the same time. It was also a challenge to implement the vector cock even if it pretty much worked exactly the same way as the time module. But it was hard to get the different methods to do the correct thing.

The main topics learned was understanding the problems that occurs in a distributed systems when sending messages. As well as get a practical understanding of how Lamport timestamps and vector clocks work.