Loggy: a logical time logger

Joakim Öberg

September 24, 2020

1 Introduction

In this homework I implemented a logging procedure that receives log events from a set of workers. The files of this assignment are **logger1.erl**, **loggerVec.erl**, **test.erl**, **testVec.erl**, **time.erl**, **vect.erl**, **worker.erl** and **workerVec.erl**. In the first tests the messages are not tagged with any time stamp, we will both see examples when messages receives in and out of order. In the later tests the events are first tagged with Lamport time stamp of the worker and we will see how events should be printed and what we see means. Then we will also see a test where vector clock is used and what a vector clock let us know that the Lamport does not.

2 Main problems and solutions

This lab was easier to implement code wise than the other that we have had. After almost no time the test was able to run, but to understand what one saw was the bigger problem. To fully understand the printouts from the tests I used the lecture videos, the course book and youtube. The hardest part to grasp was what the benefits of a vector clock is over a logical clock.

2.1 The logger

The module logger is already taken from Erlang and gave an error when trying to use the same name for a module. I choose logger1 for name for the logger to make it work. The first implementation of the logger simply printed the message it received.

2.1.1 Lamport time, the tricky part

With out the Lamport time stamps the messages receved in logger from a worker was simply printed. To introduce time stamps the **worker** module was updated with a fifth argument *time* in the function loop. When initializing a worker we use the module **time** function zero to set a workers

initial time. The logger had also to be changed to handle time a queue with messages not yet safe to print was added and a clock that had all of the workers time. A method **safeToPrint/3** was also added that if a message was safe to print, ie the incoming time stamp was greater or equal to the lowest one in hold back queue printed it.

2.2 Bonus task: Vector time

Most of the code was given for the vector time implementation as a skeleton for us to add code in. I thought leq() was somewhat hard to do because first I did not get that we wanted all of the numbers in the list to be less to return true. For the worker, logger and the test I also needed to add code. But because we use the same API for time and vect, only some lines had to be changed from time: to vect: to make them work.

2.3 The test

To be able to run several tests with different output, test/2 was rewritten to test/3 to take a third argument used for time the workers was allowed to work before the test braked.

3 Evaluation

3.1 The worker

The loop sat a timer for some time, if no messages was received during this delay the worker randomly picked another worker from its peers and sent a message to the peer, then waited for some random time in jitter/1 function and then logged that it sent a message. The jitter function takes the jitter value and waits a random value between 0 and the jitter value. A high jitter value increases the chance for a pause that long so that another worker can receive and print its message before the pause is over.

If the worker got a message during the wait, the worker logged that it had received a message before recursively called itself.

3.2 The test.erl given

The test is designed in such a way that each worker sleep for a random time. If it receives a message during this sleep it logs it, if no message was received and the random sleep is over the worker pick one random other worker from its list with peers. Because no limit of messages is set the workers keeps on doing this. After the test has started all workers is set to send or recieve messages as described above, then the test waits in 5000 milliseconds before

stopping the test, this means that the messages that was not safe to print is stuck in the hold back queue.

3.3 A first test, without Lamport time stamps

In the first test we have no Lamport time. We have three figures, figure 3 where we tested with jitter=100, figure 1 where we tested jitter=7 and figure 2 where we sat jitter=0. We know from the code that a worker select one of its peers and sends a message to the peer, after some random delay between 0 and jitter the worker log that the message was sent. Both in figure 3 and figure 1 we see that the delay the worker got was longer than the time it took for the peer to receive the message and print it. In figure 2 when the jitter was set to zero, there is no delay between a peer is selected, a message sent to the peer and that it was sent is logged. Therefor all the events in figure 2 is orderd so that a sending message is logged before a received message.

```
log: na paul {received,{hello,56}}
log: na ringo {sending,{hello,56}}
log: na ringo {received,{hello,7}}
log: na ringo {received,{hello,50}}
log: na paul {sending,{hello,50}}
log: na john {sending,{hello,7}}
log: na george {received,{hello,52}}
log: na ringo {sending,{hello,52}}
log: na george {received,{hello,21}}
log: na paul {sending,{hello,21}}
log: na ringo {received,{hello,65}}
log: na paul {received,{hello,37}}
log: na john {sending,{hello,65}}
log: na paul {received,{hello,11}}
log: na ringo {sending,{hello,37}}
log: na george {sending,{hello,11}}
```

Figure 1: No time stamps, out of order. sleep=1000, jitter=7

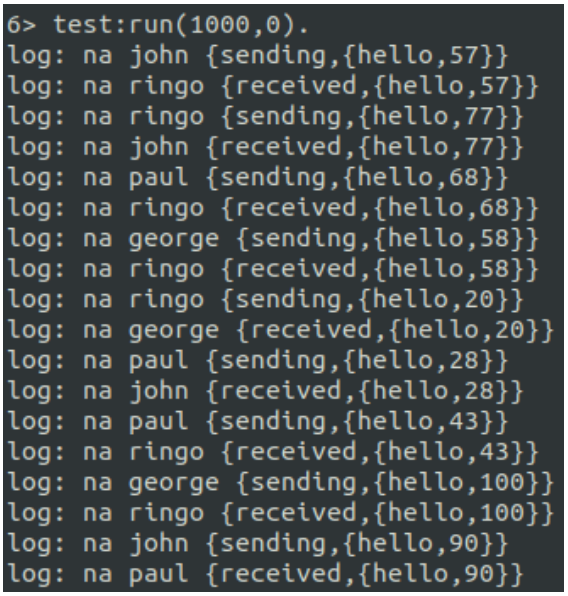

Figure 2: No time [width=8cm]stamps, in order, sleep=1000, jitter=0

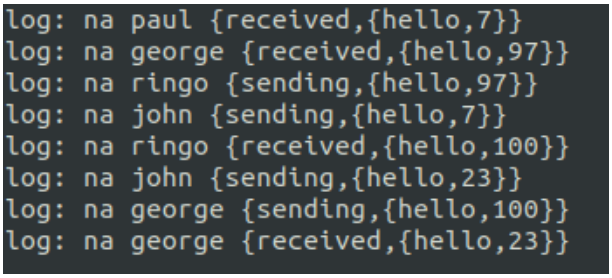

Figure 3: No time stamps, out of order. sleep=1000, jitter=100

3.4 Lamport time

With Lamport time we can place events into an accurate timeline. To do that we implement a protocol on all machines so that ordering of the events are consistent. With Lamport time stamps the following statement will be true: *If we have two events a and b, where b is caused by a. Then the time stamp of a is less than the time stamp of b* In the module **tiem** the functions zero(), inc(Name,T), merge(Ti,Tj) and leq(Ti,Tj) had good instructions and was easy to implement. From clock(Nodes), update(Node, Time, Clock) and safe(Time,Clock) was safe/2 and clock/1 the hardest to implement. The clock mostly because I did not understand if we have one

clock or one clock per node, I implemented one clock per node and kept them all in one list.

Why we implement a logical time stamp is because to get several machines to work at the same task in the correct order we need to know what each process has done or not. For example we can think of the logger as an important task where the task is divided up on several machines(workers). Even if one worker can finish all its work directly it is not sure that we can use it right away, we need to be sure that the other is in sync so that the right worker works at the right time.

3.4.1 Messages printed in actual order

Figure 4: With time stamps, out of order, sleep=2000, jitter=200

In figure 4 we see in what order the messages actually being logged. No event logs is hold back, they just work and messages are printed.

3.4.2 Messages printed in logical order

In figure 5 we see the logical time, the time where the messages are synced. Ringo and John are still the fastest worker but no messages are printed be-

Figure 5: With time stamps, printed correctly, sleep=2000, jitter=200

fore we are sure they are safe to print, we have the sorted clock (sorted on clock times) and check the lowest time in clock against the lowest time in the queue. If the time from the queue is lower than the lowest clock time, than we know that it can be printed.

Because of the tests implementation and that every message send was in sync with the logical clock some messages (around 8) was always stuck in the holdback queue when the test ended.

3.5 Bonus task: Vector time

With Lamport time stamps one event a:s time stamp is less than another event b:s if b was caused by a. The problem however is that with several machines everyone of them will have their view of the correct time and not dependent, concurrent events will sometimes have to wait in the hold back queue for the time stamp to increase before they could be printed. To solve this we implement vector clocks.

A vector clock has a copy for each of the processes times, stored as {ProcessName,Messages}

in a list. When a process is sending a message its own clock is updated, and when receiving a message the clock corresponding to the sender is updated(if not present a new clock is added with the counter set to 1). This means that by just looking at a clock we can see if an event occurred before, after or was concurrent with other events.

In figure 6 can we now see difference between messages occurring dependent on each other but also the one that happened concurrently. For example the first two logs from figure shows that in the firs line we only have [john,1] and in the second [{ringo,1},{john,1}], because ringo is not present in the first one we can see the first line as [{ringo,0},{john,1}] and now we can directly see that all counters are less in the first line and therefor we know it happened before the second.

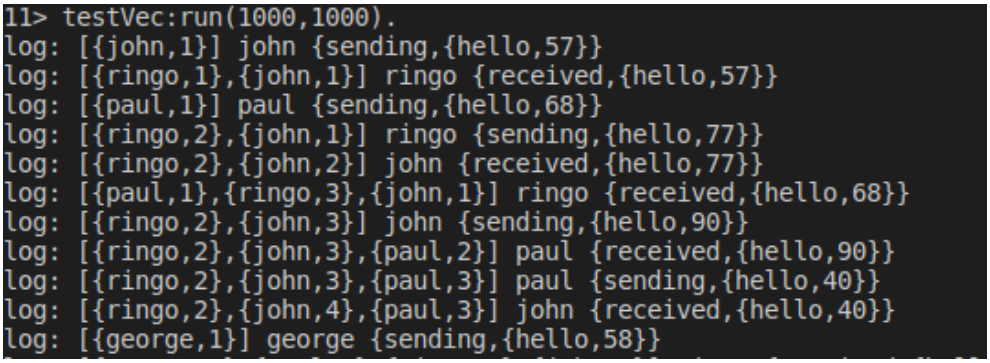

Figure 6: With vector time, printed correctly, sleep=1000, jitter=1000

Another example is line 3 where we have [{paul,1}], compared to the second line we now have entries sometimes bigger and sometimes smaller than the other one, line 3: [{paul,1},{john,0},{ringo,0}], line 2:[{ringo,1},{john,1},{paul,0}] this means that we can say that they happened concurrently independent on each other.

Because concurrent, non dependent massages could be sent/ received without waiting for all clocks the hold back queue did not grow as deep as for the logical clock. When the test stopped there was most of the time around 3 messages still in the queue and sometimes even zero.

4 Conclusions

The implementation of Lamport time make it possible to use several workers and be sure they do what they should in the right order. So that we can be sure that what is supposed to happen first actually happens before some-

thing that should happen later. The vector time implementation also made it easier to spot which events that where independent from other events and that could have happened concurrently.