Loggy – a logical time logger

Óttar Guðmundsson
21 September, 2017

1 Introduction

This week's assignment was to create a service for logging processes events. All of the events are supposed to have their own Lamport time stamp that resembles an agreed time between the processes and the service can then print the events in the same order as they were posted.

2 Main problems and solutions

2.1 The first try

The structured coded given was pretty straightforward. In short description, we ran a test that initiated a Logger function that had list of atoms that later were referred as processes. These processes (note that all of their names related to The Rolling Stones) all had a different type of Jitter and Sleep to simulate server and network related problems in the system. They could either send a message to a random process after a short interval or receive a message and the logger would then print that out. In the first try, we didn't use any special notation of the time which made things pretty easy. But we wanted to include the timestamp of the processes so they would arrive in the correct order, not a FIFO.

```
log: na ringo {received,{hello,57}}
log: na john {sending,{hello,57}}
log: na paul {sending,{hello,68}}
log: na john {received,{hello,20}}
log: na george {sending,{hello,91}}
log: na paul {sending,{hello,20}}
log: na george {received,{hello,16}}
log: na paul {sending,{hello,16}}
log: na paul {sending,{hello,54}}
log: na paul {sending,{hello,43}}
log: na paul {sending,{hello,55}}
log: na paul {sending,{hello,84}}
log: na paul {sending,{hello,50}}
log: na paul {sending,{hello,21}}
```

Image 1: No time – no problem!

Every time a process sent or received a message it increased its timestamp but as the logger printed the events out, one could obviously tell that something was wrong since the timestamps are not in a correct order. To print events in the correct order we needed to implement Lambort clock functionality.

```
28> test:runlamb(50,100).
log: 1 ringo {received,{hello,57}}
log: 1 ringo {received,{hello,68}}
log: 1 john {sending,{hello,57}}
log: 1 george {sending,{hello,91}}
log: 4 george {received,{hello,36}}
log: 4 ringo {sending,{hello,36}}
log: 1 paul {sending,{hello,68}}
log: 1 paul {received,{hello,91}}
log: 2 ringo {received,{hello,20}}
log: 3 paul {sending,{hello,40}}
log: 2 john {sending,{hello,20}}
log: 3 john {received,{hello,40}}
log: 6 john {received,{hello,39}}
log: 6 john {received,{hello,95}}
log: 6 george {sending,{hello,39}}
```

Image 2: Wrong order

2.2 Lampert clock

We knew that the only way to print safe messages was to keep track of time that all processes agreed on. It was done by keeping count of an "internal" clock which can basically be described as notation of time that each process thinks the time is. Each message was kept in a message que so we wouldn't print out before we were sure it was okay.

2.2.1 Lampert Time

To do this, we needed an extra module which we called *Time* that managed time variable each process held. The module used some basic functions used by the workers, like starting a new time, comparing times and returning the higher value of times. The tricky part was to implement the functions used by the logger itself, e.g. when it was safe to print by comparing a message timestamp to the internal clock or updating the timestamp. The clock used was a list of tuples represented as {*name, timestamp}*.

2.2.2 Lampert Worker

The worker itself was given in a structured skeleton code but needed some modifications to work with the Lampert clocks. First it needed to initiate with a new starting time provided by the time module. When sending a message to another worker, the timestamp needed to be increased and sent with the message. On receiving, the message included the timestamp from the sending worker which could be merged with the receiving worker. After doing so the agreed timestamp was then increased and sent to the log.

2.2.3 Lampert Logger

The Logger itself had to be changed drastically. Firstly the Logger needed to use an internal clock of its own as described above. For each message it received, it updated the workers time in the clock. The next part was to add the message to the message que and sort it.

```
addToQue(From, Time, Msg, []) ->
  [{From,Time,Msg}];
% addToQue if it exists
% sort recursively by total order
addToQue(From, Time, Msg, MessageQue) ->
  [LastAddedMessage | OtherMessages] = MessageQue,
  {_From, TimeOfLast, _Msg} = LastAddedMessage,
  case time:leq(TimeOfLast, Time) of
    true ->
      [LastAddedMessage | addToQue(From, Time, Msg, OtherMessages)];
    false ->
      [{From, Time, Msg} | MessageQue]
  end.
```

Image 3: Sorting algorithm

I tried out a couple of ways to sort the message que, like *list:keysort* but after reading a little bit about Erlang on the internet, I saw this fine method of sorting recursively which I used. By doing so I could use the time module I just wrote earlier. Last but not least, checking the message que was needed to see if there were any messages safe to print simply by finding the first message of the sorted que and comparing it to the lowest time of the clock. If it was lower or equal to the lowest number of the clock it could be printed and removed from the que.

This was all nice and well but could be improved. We were challenged to implement Vector clocks instead of Lampert clock for an extra bonus point which provided a fun exercise.

2.3 Vector clocks (Extra points)

The vector clocks needed to be adjusted a little bit. Each worker now had to include a vector, or a list of tuples represented just like the clock.

2.3.1 Vector Time

The old time file itself had to be replaced or highly modified since we weren't comparing simple integers anymore. The *increase* and *zero* functions were pretty much the same but both *merge* and *leq* had to be changed drastically. For example, the merging function needed to loop through all of the tuples recursively and merge all of them if the existed in the list. If they didn't, the tuple was simply added. Similar solution was used in the latter function since the only way to check if vector time stamp is less than or equal to another timestamp if each of its entries are less than or equal to the entries of the other timestamp. Thus it was done recursively as well. Finally, *update* updated the time stamp in the internal clock if it was found in the workers timestamp.

2.3.2 Vector Worker

The worker was pretty much the same, except the *increase* and *merge* functions had to deal with Vectors now. Not a problem if the former module was implemented as requested.

2.3.3 Vector Logger

Changes to the vector logger were quite big. First of all, we didn't need to sort the message que (we got two different answers from the TA's) so I did it anyways because I had implemented it with my *leq* function. After doing so the internal clock had to be updated and finally print out the messages that were safe to print. I had problems printing out those who were safe to print in the beginning, but finally did a simple function that looped through the list and printed out those who were safe using the improved *safe* function, which literally was the *leq* functionality. The function would then return those that were not printed.

3 Evaluation

As the assignment started out, the final result were very interesting. Printing out every step of the code was needed to figure out if this was working as intended. Tweaking the code in a few places was needed after the first implementations but finally I could be sure that it was working as intended.

```
log: john 1 {sending,{hello,57}}
log: paul 1 {sending,{hello,68}}
log: george 1 {sending,{hello,91}}
log: ringo 2 {received,{hello,57}}
log: paul 2 {sending,{hello,20}}
log: ringo 3 {received,{hello,68}}
log: john 3 {received,{hello,20}}
log: paul 3 {received,{hello,91}}
log: ringo 4 {sending,{hello,36}}
log: john 4 {sending,{hello,90}}
log: paul 5 {received,{hello,90}}
log: george 5 {received,{hello,36}}
log: ringo 5 {sending,{hello,2}}
log: john 5 {sending,{hello,25}}
log: paul 6 {sending,{hello,55}}
log: ringo 6 {received,{hello,25}}
log: george 7 {received,{hello,55}}
log: ringo 7 {sending,{hello,59}}
END WITH QUE LENGTH : 6 stop
```

Image 4: Lampert clock ran with test:runlamb(2000,500)

We could know that the messages were all safely printed since the *sending* log came before the *receiving* log and the timestamps were in correct order. As longer time duration was tested out, far more messages got stuck in the message que and ended out never being printed. This was due to

starvation, since some workers where faster than others and all of the fast workers messages had to wait for the slower ones. The algorithm only acknowledged partial ordering of messages. This is a classic case of why players from DOOM were kicked out of online games because they were making other people "lag" since the service was not ready to render the game because it was waiting for slower users.

```
log: [{john,1},{paul,0},{ringo,0},{george,0}] john {sending,{hello,57}}
log: [{john,1},{paul,0},{ringo,1},{george,0}] ringo {received,{hello,57}}
log: [{john,0},{paul,1},{ringo,0},{george,0}] paul {sending,{hello,68}}
log: [{john,1},{paul,1},{ringo,2},{george,0}] ringo {received,{hello,68}}
log: [{john,0},{paul,2},{ringo,0},{george,0}] paul {sending,{hello,20}}
log: [{john,2},{paul,2},{ringo,0},{george,0}] john {received,{hello,20}}
log: [{john,0},{paul,0},{ringo,0},{george,1}] george {sending,{hello,91}}
log: [{john,0},{paul,3},{ringo,0},{george,1}] paul {received,{hello,91}}
log: [{john,1},{paul,1},{ringo,3},{george,0}] ringo {sending,{hello,36}}
log: [{john,1},{paul,1},{ringo,3},{george,2}] george {received,{hello,36}}
log: [{john,3},{paul,2},{ringo,0},{george,0}] john {sending,{hello,90}}
log: [{john,3},{paul,4},{ringo,0},{george,1}] paul {received,{hello,90}}
log: [{john,3},{paul,5},{ringo,0},{george,1}] paul {sending,{hello,55}}
log: [{john,3},{paul,5},{ringo,3},{george,3}] george {received,{hello,55}}
log: [{john,1},{paul,1},{ringo,4},{george,0}] ringo {sending,{hello,2}}
log: [{john,3},{paul,5},{ringo,4},{george,4}] george {received,{hello,2}}
log: [{john,4},{paul,2},{ringo,0},{george,0}] john {sending,{hello,25}}
log: [{john,4},{paul,2},{ringo,5},{george,0}] ringo {received,{hello,25}}
log: [{john,3},{paul,5},{ringo,4},{george,5}] george {sending,{hello,17}}
log: [{john,3},{paul,6},{ringo,4},{george,5}] paul {received,{hello,17}}
log: [{john,3},{paul,5},{ringo,4},{george,6}] george {sending,{hello,47}}
log: [{john,3},{paul,7},{ringo,4},{george,6}] paul {received,{hello,47}}
END WITH QUE LENGTH : 2 stop
```

Image 5: Vector clock ran with test:runvect(2000,500)

The vector clocks surely did not suffer from starvation as seen as the image above. We could not detect when two timestamps were not ordered with respect of each other, called Causal ordering. So the vector clocks were indeed faster processing messages sent in the que at the cost of more memory. Another nice thing about vector clocks was that we could simply add more processes to the vector clocks and the algorithm would take care of rest. Each time a worker sent to a message to a new undefined process it would simply add it to its list of vectors.

I had some fun changing the Jitter and Sleep parameters as well as adding the effects in other places in the code. Fortunately, the algorithm held and messages still came in the correct order.

4 Conclusions

The Lamport clocks demonstrated and excellent usage and demo of the distributed systems. Most of my colleagues agreed that this project was both fun and rewarding experience. I think that doing the extra bonus point, Vector clock, really displayed the difference between the algorithms and in which case you should use either one. The Lamport clock is simpler and takes less memory than vector clocks but the message que might grow larger with more workers. Lamport clocks are thus vulnerable to starvation but might be improved with Round-Robin processing or something similar. With the vector clocks we are introduced to a more sophisticated algorithm that can figure out if events are related to each other or not. The message que will not get as big as in the Lamport at the cost of bigger vectors.