Loggy: A logical time logger

Tobias Johansson

September 29, 2015

1 Introduction

A primitive Erlang logger is given that simply accepts events and prints them on the screen. Also a worker module and a test module that can setup a set of workers that send messages to each other there their send and received activities is sent to the logger. This reports goes through the work of implementing Lamport's logical time on the workers and the logger. So that the events will be ordered before printed on the screen.

2 Main problems and solutions

To solve the problem, that is the to implement Lamport time on on a set of given modules, the given given modules had to understood and therefore studied. Likewise so had Lamport's logical time needed to be studied. Material for Lamport time was found in *Distributed Systems: Concepts and Design (5th Edition)* by George Couloursi and studied. This section first presents the work with the initially given modules and after that the Lamport time implementation.

2.1 Initial modules

Three Erlang modules were given initially and analyzed; `logger` that accepts events and prints them on the screen, `worker` that sends and receives events to and from its peers and `test` sets up a test to see that things work with four workers.

After receive and send in the `worker` modules a log message is sent to a `logger` instance for printing. The given `test` module and its only function `run/2` had input for; jitter that introduces a random delay between the sending of a message and the sending of a log entry, and a sleep value that determines how active the worker is sending messages.

The initial code was tested and it could be seen that the jitter value induced messages to be printed in the wrong order. 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 presented before its sent counterpart.

2.2 Lamport time

To improve the logger; logical time was about to be implemented using Lamport's [1978] invented mechanism by which the happened-before ordering can be captured numerically. Following this mechanism each process ("worker" in this case) keeps its own logical clock, an increasing software counter, its *Lamport timestamp*. To capture the happened-before the counters should be incremented before each event, when a process sends a message it send with it its counter value and on receiving a message the receiver should set its counter to the max of the incoming and counter value and its own then incremented by one before time stamping the event. Now an event's Lamport time tamp is less than another if it happened-before the other event.

To implement the functionality in the described in the previous paragraph a `time` module was told to be used. For this module its API was given, described in words, and implemented as following:

```
-module(time).

zero() -> 0.

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

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

leq(Ti, Tj) -> Ti =< Tj.

clock(Nodes) -> [{N, zero()} || N <- Nodes].

update(Node, Time, Clock) ->
    lists:keyreplace(Node, 1, Clock, {Node, Time}).

safe(Time, Clock)  ->
    lists:all(fun({_N, T}) -> leq(Time, T) end, Clock).
```

The `zero/0` gives the initial counter value, `inc/2` increments by one, `merge/2` takes the max from two time stamps, `clock/1` takes a node list and returns a list of tuples with node names and zero set counters, `update/3` is for update a time stamp for a specified node in a clock initially returned from `clock/1`. The only function that was not straightforward to implement was `safe/2`, it should return `true` if it is safe to log an event with given time and otherwise `false`.

To know if a event is safe to log/print the introductory paragraph in this subsection is taken into account. The event is assumed to be safe to log if its timestamp is less than or equal to all other processes time stamps. Because then it would be impossible that an event with a lower time stamp could come after.

The `time` module was used in `worker` and `logger` to implement Lamport time. A new variable was introduced in `worker:loop/5` to keep track of it counter, its initial value was set to `time:zero/0`. To increment the time stamp before an event and to choose the max of incoming and current timestamp `time:inc/2` and `time:merge/2` was used. Initially in the code the timestamp sent to the logger was set to `na` this was substituted with the new variable that keeps track of the Lamport time.

More effort was needed to implement handle the incoming log events with timestamps and present them in correct order according to their Lamport time in the `logger` module. Two variables `Queue` and `Clock` was introduced in the initial `logger:loop/0` to be able to hold-back and save "unsafe" log events and to keep track of the highest seen timestamp for all processes (workers). So all incoming messages get append to `Queue` and and `Clock` gets updated with `time:update/3`, initially it is set to `time:clock(Nodes)` there `Nodes` contains a list of the workers. To clarifying; "safe" log events means those that returns `true` for `time:safe/2` for the input of its timestamp and `Clock`.

To separate "safe" and "unsafe" log events, `time:safe/2` is used to split the log event queue to a tuple as following:

```
{Safe, Unsafe} = safe_split(UpdatedQueue, UpdatedClock),
```

The own written `logger:safe_split/2` returns a two-tuple there the first element is a list of "safe" log events and the second element "unsafe" log events, it also *preserves the order* of `UpdatedQueue`. The events in `Safe` is printed and those in `Unsafe` recursively given to now `logger:loop/2`.

With the until now mentioned solution the messages is still not always in order of the Lamport timestamp. To better visualise the execution `UpdatedClock` was printed in every loop. It was now seen more clearly that multiple log events, with possibly different timestamps, that became "safe" in the same loop and presented in an incorrect order. An output from a test is given below:

```
clk: [9,11,5,1]
clk: [9,11,5,13]
log: 5 john {sending,{hello,498}}
log: 5 ringo {received,{hello,319}}
log: 4 ringo {received,{hello,376}}
log: 3 ringo {sending,{hello,424}}
```

```
log: 4 john {received,{hello,424}}
log: 2 ringo {received,{hello,312}}
```

If the hold-back queue also was sorted on timestamp after each update the output above transformed to:

```
clk: [9,11,5,1]
clk: [9,11,5,13]
log: 2 ringo {received,{hello,312}}
log: 3 ringo {sending,{hello,424}}
log: 4 ringo {received,{hello,376}}
log: 4 john {received,{hello,424}}
log: 5 john {sending,{hello,498}}
log: 5 ringo {received,{hello,319}}
```

3 Evaluation

The given modules and Lamport time was understood after they was studied and this process explained in the report. The implemented code explained in section 2.2 was tested multiple times. With the queue sorted after append on incoming log event, on the counter, "Lamport order" was always retained and therefore also sent messages always printed before their corresponding received messages.

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

A conclusion to the mentioned observation in section 3 is that if any worker or group of worker is isolated from the others, logging prints can be starved out or very much delayed (big hold-back queue). But it guessed that this would not matter in a more advanced logger. The logger can not always tell what order events happened, but it can tell much about it when processes interact and their Lamport clocks gets updated. With the data from the final log much causal order can be seen.

The hardest part of the problem solving was for the author to understand what was safe to print and wrap his head around what happened when log events with different timestamps got "safe" at the same time. After printing the updated clock in all loops of the logger this was became much less tricky and easy to handle. The observation there one worker is idle also made the logic more clear.