Loggy: extended with a vector clock

Johan Montelius

October 2, 2015

Introduction

So you have solved the initial assignment and constructed a logger that works with Lamport clocks. Now why not extend it so it can work with vector clocks - tricky, no not really.

The time API

If you followed the instructions in the assignment you should have a module called `time` that exports the following functions:

- zero() : returns an initial timestamp
- inc(Name, T) : returns a timestamp incremented by a named process
- merge(Ti, Tj) : merges the two timestamps
- leq(Ti,Tj) : true if the timestamp Ti is less than or equal to Tj
- clock(Nodes) : returns a *clock* that can keep track of the nodes
- update(Node, Time, Clock) : returns a clock that has been updated given that we have received a log message from a node at a given time
- safe(Time, Clock) : is it safe to log an event that happened at a given time, true or false

I would guess that you implemented your module by representing Lamport timestamps simply as integers: 0,1,2.. etc. The four first functions then became more or less trivial while the last three required some thought.

We should now implement a module called `vect` that has the same API but uses vector clocks instead.

Representing a vector clock

There are mainly two ways to represent a vector clock: as a vector (or tuple in Erlang speak) or as a list of process names and values. Assume we have four processes *john*, *paul* etc, we could then represent a vector time by a tuple `{3,4,2,1}` but we would have to remember in which order they stand (*john* is the first etc). We would also have to know how many processes we have to start with in order to know how big the tuple should be.

A more flexible way of implementing the vector is by a list of process names and values. The vector given above could the be represented by the list:

```
[{john, 3}, {ringo, 2}, {paul, 4}, {george, 1}]
```

Note that the order does not matter. Also note that the list `[{ringo, 2}, {george, 1}]`, is a perfectly valid vector time stamp. It simply states that the event is independent from any event executed by *john* or *paul* (or any of the members of the Rolling Stones or other groups). This means that `[]` also is a perfect valid vector time stamp, it is the *zero* time stamp.

How would you implement the first two functions? You could use the following skeleton code:

```
zero() -> ....

inc(Name, Time) ->
    case lists:keyfind(Name, 1, Time) of
        ... ->
            lists:keyreplace(Name, 1, Time, ...);
        false ->
            [...|Time]
    end.
```

The next functions is equally simple (sort of), the only tricky thing is that the timestamps are not ordered by name nor do they have to be complete; just because we have `{john, 3}` in one timestamp does not mean that we have an entry for `john` in the other timestamp.

```
merge([], Time) ->
    ...;
merge([{Name, Ti}|Rest], Time) ->
    case lists:keyfind(Name, 1, Time) of
        {Name, Tj} ->
            [... |merge(Rest, lists:keydelete(Name, 1, Time))];
        false ->
            [... |merge(Rest, Time)]
    end.
```

If you have completed `merge/2`, then `leq/2` is a walk in the park. We only have to remember that a 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. If the other time stamp does not have an entry for a given process that means that it implicitly has a zero entry.

```
leq([], _) ->
    ...;
leq([{Name, Ti}|Rest],Time) ->
    case lists:keyfind(Name, 1, Time) of
        {Name, Tj} ->
            if
                Ti =< Tj ->
                   ...:
                true ->
                   ...
            end;
        false ->
            ...

    end.
```

Hmm, now we only have the *clock* and its operations left to implement. We have some alternatives here but let's ponder what we want to achieve. When the logger is being sent a message with a vector timestamp, it wants to determine if it is safe to log the message. The *clock* should reflect what messages we have seen from each of the nodes. Is this not exactly what we have done in the Lamport clock solution?

The only difference is that we implement the vector clock so that it is independent from how many nodes we have in the system. The initial clock will thus reflect that we have seen no messages at all.

```
clock(_) ->
    ....
```

Updating a clock is only slightly more complicated from before. If receive a message from *john* at time `[{john, 2}, {paul, 3}]`, how should we update our clock? We have to take care of the case where this is the first message from *john*, so we might lack an entry.

```
update(From, Time, Clock) ->
    ...  = lists:keyfind(From, 1, Time),
    case lists:keyfind(From, 1, Clock) of
        {From, _} ->
            lists:keyreplace(From, 1, Clock, ...);
        false ->
             [...| Clock]
    end.
```

Ok, you're almost there - now for the `safe/2` function. Given a clock we want to determine if it is safe to log a message. When is it safe to log `[{john, 2}, {paul, 3}]` - when you have seen two messages from *john* and three from *paul*. If your clock contains entries for both *john* and *paul*, and the values 2 and 3 are less than or equal to the corresponding values in the clock - then it is safe to log the message.

So, if for each entry in `Time`, you have an entry in `Clock` and the entry in `Time` is less than or equal to the entry in the `Clock` - then it is safe to log the message; have we seen this before?

```
safe(Time, Clock) ->
    ....
```

I think you're done, you have (fingers crossed) implemented a module that will replace the Lamport clock that you have used. Let's see how it performs.

Take it for a spin

If you run tests using the Lamport clock and the vector clock you might see some differences. You could augment your code of the logger and do a print out every time it queues a message in the hold back queue. Is there a difference?

Another thing to note is that the logger does not know aforehand how many nodes we have in the system so we can start new nodes as we go. You can start a set of workers and then start another set that partly overlap the first set. The logger will still work and log events in an happen-before order.