Seminar Report : Loggy

Vincent Delaunay

October 2, 2014

1 Introduction

Loggy is a logger for a simple distributed system.

This seminar aims to offer a grasp on logical time. We have a small distributed system, where nodes witness events such as "sent message X", or "received message Y". And we want loggy to be able to log theses events in a way such as what is logged always describes a consistent state. We will try and do it using Lamport's logical time.

We were given a "worker" implementation already, so we didn't have to think about the way we would do the distributed system.

First, I will present the functions I wrote to evaluate whether the logger acts correctly or not. Then we will see my Lamport's time implementation, see how well it does, and in which cases it fails. Next we will solve theses issues by staging he messages before logging them. Finally, I will go further and set up a more advanced model using a "network" layer, see how it makes the previous implementations fail, and how to solve it.

2 Evaluations

We need automated methods which can check for us whether the logger behaves correctly or not.

The first function I wrote is used to check that no message is actually "received" before having been "sent".

```
received_before_sent(Msgs) ->
    Received = lists:filter(fun({_,_,_,{Type,_}})-> Type==received end,Msgs),
    Was_sent_before = lists:map(
        fun(X)->
            {_,_,_,{_,{Message,Token}}} = X,
            case lists:keysearch({sending,{Message,Token}},4, sublist_until(X, Msg
                false -> {Token};
                _ -> true
            end
        end,
        Received),
    lists:filter(fun(X)-> not (X==true) end, Was_sent_before).

sublist_until(Elem, List) ->
    Index = string:str(List, [Elem]),
    {Previous_elems,_} = lists:split(Index-1,List),
    Previous_elems.
```

And I wrote some unit tests to test this function.

With the system implementation given in the subject, using this evaluation function, we can see that whenever we have a "Jitter" value which isn't zero, we see messages being "received" before having been "sent". The bigger the Jitter, the more errors we have.

The second evaluation I wrote just checked whether, when using Lamport's time, the history list of message was actually well sorted according to it's logical time :

```
    logical_order(Msgs) ->
        Msgs == lists:keysort(2,Msgs).
```

Ps : using theses methods allowed me to run the simulation with more messages being sent (usualy 1000+), so I had to modify the "id" of each message, to :

```
    Message = {hello, random:uniform(100000000000)}
```

3 Lamport's time

We use Lamport's time to assign a logical timestamp to every message.

Adding the timestamps to the messages just required minor modifications to the "worker" :

```
    loop(Name, Log, Peers, Sleep, Jitter, Local_time)->
        % ...
        receive
            {msg, Remote_time, Msg} ->
                Time = erlang:max(Remote_time,Local_time)+1,
                Log ! {log, Name, Time, {received, Msg}},
                loop(Name, Log, Peers, Sleep, Jitter, Time);
            % ...
            after Wait ->
                %...
                Time = Local_time+1,
                %...
                Selected ! {msg, Time, Message},
        %...
```

Before implementing the stages, I found it interesting to just build a list of the messages received by the logger, and sort them according to their timestamp. Using my first evaluation function, I noticed that there were very few errors in this case, and it was always the same pattern : one of the last messages was a "received", for wich the "sent" had not yet been seen because the system stopped to early.

It put me on the track for the next step :

4 Staging area

When using Lamport, we are able to order our messages, but how can we be sure we have all the messages we need before actually logging ?

The trick is to build a Stage. New messages go to the stage, and when we're sure that for a given time T, we received all the messages timestamped T2 =¡ T, we can log all thoses messages. Our stage will actually consist of a Substage for each node that we know of. Incoming messages go to the Substage of their node (wich is sorted since the messages from one node are not scrambled). When we have a Substage empty, we can't tell where in the logical timeline it's node actually is. But when we have at least one message in each substage, we know that we can safely unstage all the message until

```
Tmin = Min(Time for the eldest message of each node)
```

When unstaged, messages can be safely logged.

The main parts of the implementation :

```
init(Nodes) ->
    % Initialisation of the stage ith dummy entries.
    Stage = lists:map( fun(X)-> {X, []} end, Nodes ),
    loop(Stage,[]).

loop(Stage,Msgs) ->
    receive
        {log, From, Time, Msg} ->
            % Stage the new message.
            Updated_stage = stage_message({From, Time, Msg},Stage),
            % Process the stage to get the new stage and the unstaged messages
            {Unstaged_messages, New_stage} = process_stage(Updated_stage, []),
            % For each unstaged message, log it and prepend it to the list of alre
            New_Msgs = lists:foldl(
                fun(X,M)-> log(X), [X|M] end,
                Msgs, lists:reverse(Unstaged_messages)
            ),
            loop(New_stage, New_Msgs);
    % ...

process_stage(Stage, Messages) ->
    None_empty = no_substage_empty(Stage),
    % If we have at least one message staged for every node.
    if None_empty ->
        % Get the time of the oldest message we have staged.
        Min_time = min_time_from_stage(Stage),
        % Unstage the messages from this time.
        {New_messages, New_stage} = unstage_messages(Min_time, Stage, Messages),
        % Repeat until one of the stages is empty.
        process_stage(New_stage, New_messages);
    % Else, no unstaged messages, and the stage isn't updated.
    true ->
        {Messages, Stage}
    end.
```

At this point, we can verify that everything works and ours logs are well ordered (but not all the messages received are logged, some are still in the stage, obviously) :

```
% ...
log: ringo 72 {received,{hello,28675091633}}
log: john 73 {received,{hello,44944231916}}
log: ringo 73 {sending,{hello,61415827722}}
log: john 74 {sending,{hello,80333723677}}
log: ringo 74 {sending,{hello,30451414687}}
log: george 74 {received,{hello,61415827722}}

*********************************************
Still staged :
[{john,[{john,75,{received,{hello,30451414687}}},
        {john,76,{sending,{hello,42600842534}}},
        {john,79,{received,{hello,9376204718}}}]},
 {paul,[{paul,77,{received,{hello,85407721504}}}]},
 {ringo,[]},
 {george,[{george,75,{received,{hello,80333723677}}},
          {george,76,{sending,{hello,85407721504}}},
          {george,77,{received,{hello,42600842534}}}]}]

  *********************************************
  Messages received before sent (if any) :[]
  Are messages logically sorted ? : true
```

So.. let's go further !

5 Tweaking the model

The current modelization makes two hypothesis. How can we confront them ? The first hypothesis of our current Nodes/Logger model is that we have a given number of Nodes, and they won't change. Solving this is pretty straight-forward, and not really interesting to implement (just extend the protocol with a "join" and a "leave" message, and update the stage accordingly.)

The second hypothesis looks more interesting to me. Currently, the messages from a given node always reach the logger in the right order : message from Time4 won't arrive before message from Time3. It may be so with the right underlying protocol (TCP), but it might also not be the case (UDP anyone?).

Let's implement a new model. I decided to add an extra layer between the "Nodes" and the "Logger" : the "Network". My network just receives the messages from the nodes, stores them in a big list, and randomly outputs any of them at some time. It is a very bad network, as the first received message has the same odds to be outputed as the last received one. It could be changed using somthing else than a uniform distribution for the output, but having a bad network is good to test our system anyway.

We implement a function in the "test" module to run the system with the network :

```
run_network(Sleep, Jitter, Latency) ->
    %...
    Log = logger:start([john, paul, ringo, george]),
    Network = network:start(Latency,Log),
    A = worker:start(john, Network, 13, Sleep, Jitter),
    %...
    worker:peers(D, [A, B, C]),
    %...
    timer:sleep(5000),
    logger:stop(Log),
    network:stop(Network),
    %...
```

We can run it, and see our logger fail miserably :

```
8> test:run_network(200,300,10).
  %...
  log: ringo 72 {received,{hello,28675091633}}
  log: john 73 {received,{hello,44944231916}}
  log: ringo 73 {sending,{hello,61415827722}}
```

```
log: ringo 74 {sending,{hello,30451414687}}

*****************************************
Packets still in the network : []

**********************************************
Still staged :
[{john,[{john,75,{received,{hello,30451414687}}},
        % some more...
 {paul,[{paul,77,{received,{hello,85407721504}}}]},
 {ringo,[]},
 {george,[{george,75,{received,{hello,80333723677}}},
          % some more...

**********************************************
Messages received before sent (if any) :[{56087429704},
                                        % long list here...
                                         {28675091633}]
Are messages logically sorted ? : false
```

6 Dealing with the network

Here, the messages from each node won't arrive in order anymore.

My solution was to extend the "log" message so that every node would tell not only his current Time, but also his Previous_time :

```
    Log ! {log, Name, Time, Previous_time, {received, Msg}},
```

Then, in addition to the Substage I already have for each node, I will also have a Pre-Substage, in wich I put every message from this node until I have the one with the matching Previous_time. The Stage now has such a structure :

```
  [{Node1, Last_time, Pre_substage, Substage}, {Node2, Last_time, Pre_substage, Subs
```

The new stage_message method now deals with this Pre-Substage :

```
stage_message({From, Time, Previous_time, Msg}, Stage) ->
    lists:map(
        % We loop through the different Nodes to pass the message to stage in the ri
        fun(X)-> case X of
                    % If it is the right node and the previous time matches..
                    {From, Previous_time, Pre_substage, Substage} ->
                        % We look in the pre-substage if any message can now go to t
                        {Un_prestaged_messages, New_time, New_pre_substage} = un_pre
```

```
                                % We pass all of this to the substage.
                                {From, New_time, New_pre_substage, Substage++[{From,Time,Pre
                            % If it is the right node but the previous time doesn't matche,
                            {From, Other_time, Pre_substage, Substage} ->
                                % We want to keep the pre_substage sorted.
                                {From, Other_time, lists:keymerge(2,[{From,Time,Previous_ti
                            % if it doesn't concern this node, pass.
                            Any -> Any
                        end end,
        Stage
    ).
```

And as we can see, it will run perfectly :

```
8> test:run_network(200,300,10).
  %...
  log: paul 77 76 {sending,{hello,38495093866}}
  log: john 78 77 {sending,{hello,80333723677}}
  log: ringo 78 75 {received,{hello,38495093866}}

  *****************************************
  Packets still in the network : []

  **********************************************
  Still staged :
  [{john,78,[],[]},
   {paul,82,[],[{paul,82,77,{received,{hello,80717294763}}}]},
   {ringo,78,[],[]},
   {george,80,[],
           [{george,79,76,{received,{hello,80333723677}}},
            {george,80,79,{received,{hello,57764043984}}}]}]

  **********************************************
  Messages received before sent (if any) :[]
  Are messages logically sorted ? : true
```

7 Conclusions

The main conclusion I draw out of this is that dealing with time may cost a lot. It costs a lot of computational power to sort and order all theses lists whenever we receive a message, and for such a message received we're not even sure we will really output it. In fact, we always end up with some messages that are "stuck", in the logger. There is a trade-off between consistency and efficiency, but as we saw with the "network" model, we have some options in which hypothesis we consider, and they come at different costs.

On a personnal perspective, I had a lot of fun doing this assignment, and I really liked it being such an open subject !