Report 3: Loggy Seminar Report

Alberto Lorente Leal

21 September 2011

1 Introduction

In this seminar, we cover in this case how the notion of time is handled in distributed systemst. In this scenario, process from different systems send and receives messages from one place to the other but in the end it is not so simple. It is possible that during the transmissions of messages, the time we consider might not be the same for systems located in a distributed manner and so we could have issues in the handling of information. In this seminar we learn how this problem is solved by using concept of logical time that lets us to address this issue by different implementations like lamport time or vector clocks.

2 Task:

The main task we had to do with loggy was to take the basic loggy implementation which spawned some processes that started to send messages between them. We extended this implementation and we added the concept of logical time by introducing the concept of Lamport Time which addresses the problem of time management in distributed systems. Also we implemented a hold-queue in the logger in order to hold back the messages in order to print them in the correct time order.

2.1 Implementing Lamport Time:

As the Algorithm states, in this case the processes communicating should keep a notion of their time; in this case by a personal counter. This counter should be updated in the following cases:

- Receive a message from another process. We get their time counter from the message and compare that time with our personal time counter in the recepting process. If this incoming time is greater from our time, we update our timer with this new value. Then we increment its clock.
- Sending a message to another process. We increment our clock.

The modified code will correspond to the worker module in this case the loop function in which we added a new argument in order to increase it counter. It should be something like this:

```
loop(Name, Log, Peers, Sleep, Jitter, Counter)->
        Wait = random:uniform(Sleep),
        receive
                {msg, Time, Msg} ->
        if Time > Counter ->
                Log ! {log, Name, Time, {received, Msg}},
        loop(Name, Log, Peers, Sleep, Jitter,Time+1);
        true->
                Log ! {log, Name, Time, {received, Msg}},
                loop(Name, Log, Peers, Sleep, Jitter,Counter+1)
        end;

        stop ->
                ok;

        Error ->
                Log ! {log, Name, time, {error, Error}}

        after Wait ->
                Selected = select(Peers),
                Time = Counter,
                Delay = random:uniform(Jitter),
                Message = {hello, Delay},
                Selected ! {msg, Time, Message},
                timer:sleep(Delay),
                Log ! {log, Name, Time, {sending, Message}},
                loop(Name, Log, Peers, Sleep, Jitter, Time)
        end.
```

2.2 Implementing the hold queue:

The next step was to implement a hold back queue in order to track the messages receive and print them in the correct time order. Now that we had the processes working with lamport time, this will help us to order the messages but we still need to think when to print the messages. If we have in mind how the messages are treated in this scenario, their will be no harm if we find that in our queue we find two messages from the same process in different times. We should have in mind that it is possible that we have messages from the other processes that may have arrived in earlier in between that range. Lets remark that this approach works if we have an ordered list

so we need to keep it updated. We did the following modifications in the logger module:

```
init(Nodes) ->
        loop([],Nodes).

loop(Queue,Nodes) ->
        receive
                {log, From, Time, Msg} ->
                Unordered = [{From,Time,Msg}|Queue],
                Sorted = lists:keysort(2, Unordered),
                NewQueue = print(Sorted,Nodes),
                loop(NewQueue,Nodes);

        stop ->
                ok
        end.
```

To address the problem we first collected the messages in a tuple list in which we ordered using lists:keysort/2 which lets us order a tuple list having the nth element of a tuple as an index. We implemented the print function to show the messages in order and the existcolleagues function to see if the condition to print is present.

```
print([{From,Time,Msg}|Rest],Nodes)->
        RestNodes=Nodes--[From],
        case existcolleagues(Rest,RestNodes) of
        true->
                log(From,Time,Msg),
                print(Rest,Nodes);
        false ->
                [{From,Time,Msg}|Rest]
        end.

existcolleagues(Rest,[])->
        true;
existcolleagues(Rest,[H|T])->
        case lists:keymember(H, 1, Rest) of
        true->
                existcolleagues(Rest,T);
        false->
                false
        end.
```

3 Results

3.1 Basic Logger:

In one of the runs I managed to identify an instance in which the the messages where unordered. It was possible to detect this because we could see that the message order was not correct as before receiving several messages got in between.

```
log: george na {sending,{hello,1}}
log: george na {received,{hello,29}}
log: ringo na {sending,{hello,29}}
log: ringo na {received,{hello,44}}
log: paul na {sending,{hello,44}}
log: paul na {received,{hello,64}}
log: john na {sending,{hello,64}}
log: john na {received,{hello,1}}
log: george na {received,{hello,32}}
log: john na {sending,{hello,32}}
```

3.2 Lamport Time without Queue:

The following run was in order to see that the lamport time was implemented correctly and we could see the messages where printed in the incorrect order. In this case by checking the logical time we can identify the messages that are in the wrong order.

```
(ed48c_alberto_c74d77_erlide@alberto-M11X)6>
log: george 0 {sending,{hello,1}}
log: george 0 {received,{hello,29}}
:
:
log: george 4 {received,{hello,99}}
log: ringo 4 {sending,{hello,99}}
log: george 5 {received,{hello,46}}
log: john 5 {sending,{hello,46}}
log: ringo 5 {received,{hello,35}}
log: john 5 {sending,{hello,35}}
log: john 4 {received,{hello,95}}
log: paul 4 {sending,{hello,95}}
log: john 6 {received,{hello,96}}
log: ringo 6 {sending,{hello,96}}
:
:
```

3.3 Lamport Time with Queue:

The last run made correctly printed the messages in the correct order but we need to keep in mind that still this is some sort of fix. In real life, the messages are still received out of orderdue to the different race conditions that may arise (propagation delay, skew in clocks, etc..) and which are impossible to predict and correct. Although this still happens, we manage to address this problem thanks to the concept of logical time which gives a simple way to know the order of the messages received. Thanks to Lamport Clocks, we have a simple time stamp that in lets us identify the order in which messages are thrown in between the different processes without having to keep a general time reference that is susciptable to possible modifications from the real world.

4 Conclusion

Thanks to this seminar, we learn how to address the concept of time in distributed systems. This a quite important concept as in some cases depending on the application which might depend in the correct flow of events. Also we need to realise that although time inside a system might be better controlled, in a distributed system we see that new race conditions appear which makes this concept of time to be taken in consideration.