Loggy: a logical time logger

Johan Montelius

September 24, 2009

Introduction

In this exercise you will learn how to use logical time in a practical example. The task is to implement a logging procedure that receives log events from a set of workers. The events are tagged with the Lamport time stamp of the worker and the events must be ordered before written to stdout. It's slightly more tricky than one might first think.

1 A first try

To have something to start from we will first build a system that at least does something.

1.1 the logger

The logger simply accepts events and prints them on the screen. It will be prepared to receive time-stamps on the messages but we will not do very much with them now.

```
-module(logger).

-export([start/0, stop/1]).

start(Nodes) ->
    spawn(fun() ->init(Nodes) end).

stop(Logger) ->
   Logger ! stop.

init(_) ->
    loop().
```

The logger is given a list of nodes that will send it messages but for now we simply ignore this. We might use it later when we extend the logger.

```
loop() ->
    receive
        {log, From, Time, Msg} ->
```

```
            log(From, Time, Msg),
            loop();
        stop ->
            ok
    end.

log(From, Time, Msg) ->
    io:format("log: ~w ~w ~p~n", [From, Time, Msg]).
```

Erlang will give us a FIFO order for message delivery but this only orders messages in between two processes. If a process A sends a message to the logger and then sends a message to process B, process B can act on the message from A and then send a message to the logger. We would of course like to have the log message from A to be printed before the message from B but there is nothing that guarantees that this will happen (unfortunate for this tutorial you will have to run extensive tests before you detect this in real life, we could however introduce some delay in the system to increase the probability).

2 the worker

The worker will be very simple, it will wait for a while and then send a message to one of its peers. While waiting, it is prepared to receive messages from peers so if we run several workers and connect them with each other we will have messages randomly being passed between the workers.

To keep track of what is happening and in what order things are done we send a log entry to the logger every time we send or receive a message. Note that the original version of the worker will not keep track of logical time, it will simply send events to the logger.

The worker is given a unique name, access to the logger and some values to make things more random.

```
-module(worker).

-export([start/5, stop/1]).

start(Name, Logger, Seed, Sleep, Jitter) ->
    spawn(fun() -> init(Name, Logger, Seed, Sleep, Jitter) end).

stop(Worker) ->
    Worker ! stop.

init(Name, Log, Seed, Sleep, Jitter) ->
    random:seed(Seed, Seed, Seed),
```

```
    receive
        {peers, Peers} ->
            loop(Name, Log, Peers, Sleep, Jitter);
        stop ->
            ok
    end.
```

The upstart phase in `init/2` looks odd but it is there so that we can start all workers and then inform them who their peers are. If we would have given the list of peers when the worker was started we could have a race condition where a worker is sending messages to workers that have not yet been created.

We will give the worker a unique value to seed its random generator. If all workers started with the same random generator they would be more in sync, more predictable and less fun. We also provide a sleep and jitter value; the sleep value will determine how active the worker is sending messages, the jitter value will introduce a random delay between the sending of a message and the sending of a log entry.

```
loop(Name, Log, Peers, Sleep, Jitter)->
    Wait = random:uniform(Sleep),
    receive
        {msg, Time, Msg} ->
            Log ! {log, Name, Time, {received, Msg}},
            loop(Name, Log, Peers, Sleep, Jitter);
        stop ->
            ok;
        Error ->
            Log ! {log, Name, time, {error, Error}}
    after Wait ->
            Selected = select(Peers),
            Time = na,
            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)
    end.

select(Peers) ->
    lists:nth(random:uniform(length(Peers)), Peers).
```

The worker does not know about time so we simply create a dummy value `na` in order to have something to pass to the logger. The messages

could of course contain anything but here we include the hopefully unique delay so that we can track the sending and receiving of a message.

3 the test

If we have the worker and the logger we can set up a test to see that things work.

```
-module(test).

-export([start/0, stop/1]).

start() ->
    Log = logger:start([john, paul, ringo, george]),
    A = worker:start(john, Log, 13, 2000, 100),
    B = worker:start(paul, Log,  34,2000, 100),
    C = worker:start(ringo, Log, 45, 2000, 100),
    D = worker:start(george, Log, 19, 2000, 100),
    A ! {peers, [B, C, D]},
    B ! {peers, [A, C, D]},
    C ! {peers, [A, B, D]},
    D ! {peers, [A, B, C]},
    [Log, A, B, C, D].

stop([Log, A, B, C, D]) ->
    Log ! stop,
    A ! stop,
    B ! stop,
    C ! stop,
    D ! stop.
```

This is only one way of setting up a test case. As you see we start by creating the logging process and four workers. When the workers have been created we send them a message with their peers.

In order to stop this test we have to keep track of the process identifiers returned from the start procedure and use it as an argument in the stop procedure. There are other ways of solving this, such as registering processes with names but this will work for now.

In an Erlang shell try the following calls:

```
>All = test:start().
 :
 :
```

```
>test:stop(All).
 :
 :
>f().
 :
```

The call to `f/0` is a shell command that makes the Erlang shell forget about the binding of All. This means that you can try the same calls again without having a binding conflict.

Run some tests and try to find log messages that are printed in the wrong order. How do you know that they are printed in the wrong order? Experiment with the jitter ans see if you can increase or decrease (eliminate?) the number of entries that are wrong.

4 Lamport Time

Your task now is to introduce logical time to the worker process. It should keep track of its own counter and pass this along with any message that it sends to other workers. When receiving a message the worker must update its timer to the greater of its internal clock and the time-stamp of the message before incrementing its clock.

Do some tests and identify situations where log entries are printed in the wrong order. How do you identify messages in the wrong order. What is always true and what is sometimes true?

5 The tricky part

Now for the tricky part. If the logger would just collect all log messages and save them for later, one could wait with the ordering until all messages had been received. If we do want to print messages to a file or stdout in the correct order but during the execution we must take care not to print anything too early.

We must somehow keep a hold-back queue of messages that we can not yet deliver because we do not know if we will receive a message with an earlier time stamp. How do we know if messages are safe to print?

This sounds easy, and it is of course once you get it right, but there will be things that you forget to cover before you get it right.

At the seminar you should first have the worker and logger process implemented. The workers should keep track of the logical time and update it as it sends and receives messages. The logger should keep a hold-back queue with messages that it has not printed yet. When messages are printed they should be in the right order.

You should also write a two page report on your initial observations (please don't give me two pages of screen dumps). Did you detected entries out of order and if so, how did you detect them. What is it that the final log tells us? Did events happen in the order presented by the log? How large will the hold back queue be, make some tests and try to find the maximum number of entries.

6 Discussion

How would things be different with vector clocks? This is a very interesting issue. If you think the first part of this seminar was easy, try to implement vector clocks. You will find that it's not that difficult and it does actually give you some benefits.