ID2201 Distributed Systems, basic course

Loggy: A Logical Time Logger

Johan Montelius, Vladimir Vlassov and Klas Segeljakt
Email: {johanmon,vladv,klasseg}@kth.se
July 7, 2023

Introduction

This exercise will teach you 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 being 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 timestamps on the messages, but we will not do very much with them now.

```
-module(logger).

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

start(Nodes) ->
    spawn_link(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 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", [Time, From, Msg]).
```

Erlang will give us a FIFO order for message delivery but this only orders messages between two processes. If 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 (unfortunately 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, we will randomly pass messages 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 and access to the logger. We also provide the worker with 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, and the jitter value will introduce a random delay between the sending of a message and the sending of a log entry.

```
-module(worker).

-export([start/5, stop/1, peers/2]).

start(Name, Logger, Seed, Sleep, Jitter) ->
    spawn_link(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 had 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. For convenience, we provide a functional interface.

```
peers(Wrk, Peers) ->
   Wrk ! {peers, Peers}.
```

The worker process is quite simple. The process will wait for a message from one of its peers or, after a random sleep time, select a peer process that is sent a message. The worker does not know about time, so we create a dummy value, `na`, to have something to pass to the logger. The messages could contain anything, but here we include a hopefully unique random value to track the sending and receiving of a message.

```
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,
            Message = {hello, random:uniform(100)},
            Selected ! {msg, Time, Message},
            jitter(Jitter),
            Log ! {log, Name, Time, {sending, Message}},
            loop(Name, Log, Peers, Sleep, Jitter)
    end.
```

The selection of which peer to send a message to is random, and the jitter introduces a slight delay between sending the message to the peer and informing the logger. If we don't introduce a delay here, we would hardly ever have messages occur out of order when running in the same virtual machine.

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

jitter(0) -> ok;
jitter(Jitter) -> timer:sleep(random:uniform(Jitter)).
```

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([run/2]).

report on your initial observations
run(Sleep, Jitter) ->
    Log = logger:start([john, paul, ringo, george]),
    A = worker:start(john, Log, 13, Sleep, Jitter),
    B = worker:start(paul, Log, 23, Sleep, Jitter),
    C = worker:start(ringo, Log, 36, Sleep, Jitter),
    D = worker:start(george, Log, 49, Sleep, Jitter),
    worker:peers(A, [B, C, D]),
    worker:peers(B, [A, C, D]),
    worker:peers(C, [A, B, D]),
    worker:peers(D, [A, B, C]),
    timer:sleep(5000),
    logger:stop(Log),
```

```
worker:stop(A),
worker:stop(B),
worker:stop(C),
worker:stop(D).
```

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.

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 and see if you can increase or decrease (eliminate?) the number of wrong entries.

4 Lamport time

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

To compare our solutions and also change them in an interesting way, you should implement the handling of the Lamport clocks in a separate module `time`. This module should contain the following functions:

- zero() : return an initial Lamport value (could it be 0)
- inc(Name, T) : return the time T incremented by one (you will probably ignore the Name, but we will use it later)
- merge(Ti, Tj) : merge the two Lamport time stamps (i.e., take the maximum value)
- leq(Ti,Tj) : true if Ti is less than or equal to Tj

Ensure that the worker module only uses this API and does not contain any knowledge of how you represent Lamport time (which might be as simple as 0,1,2,..). We might want to change the representation later and then only have to work with the `time` module.

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

4.1 The tricky part

Now for the tricky part. If the logger would collect all log messages and save them for later, one could wait with the ordering until all messages had

been received. If we 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 keep a holdback queue of messages 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 you must remember to cover before you get it right. There are some ways to solve this, but we should follow these guidelines to make some changes to the implementation later. The logger should have a *clock* that keeps track of the timestamps of the last messages seen from each worker. It should also have a *holdback queue* where it keeps log messages that are still unsafe to print. When a new log message arrives, it should update the clock, add the message to the holdback queue and then go through the queue to find messages that are now safe to print.

Extend the `time` module so that it also implements the following functions:

- clock(Nodes) : return a *clock* that can keep track of the nodes;
- update(Node, Time, Clock) : return 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?

The logger should only use the API from the time module; there should be no knowledge of how the Lamport time nor the clock is represented. If you do it right, you should later be able to change the representation without changing the logger.

5 Requirements to pass the homework assignment

You must implement the worker and logger process to pass this homework assignment. The worker keeps track of the logical time and updates it as it sends and receives messages. The logger keeps a holdback queue with messages not printed yet. When messages are printed, they should be in the right order.

You are also to write a report that describes your `time` module (please don't give me a page of source code, describe in your own words). Did you detect entries out of order in the first implementation, 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 holdback queue be, make some tests and try to find the maximum number of entries.

You present and demonstrate your homework in person to a course teaching assistant at a reporting seminar.

6 Optinal task for extra bonus: Vector clocks

How would things be different with vector clocks? This is a very interesting issue. If you think the homework was easy, try implementing vector clocks as an optional task for an extra bonus. You will find that it's not that difficult and it does actually give you some benefits.