HW3 : Loggy - a logical time logger

[redacted]

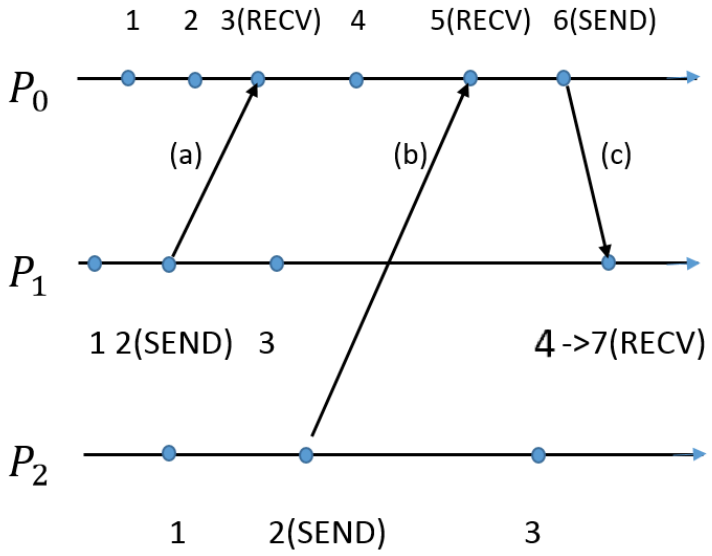
September 24, 2024

1 Introduction

We implement a time logger that tracks in which time and order different happenings occur in a distributed system. The point is to understand if event A happens before event B, which can explain the relationship between the events. For instance - if event A happens before event B, event A could have influenced event B.

How the logger functions in short:

1. **Initialization**: Each process starts with a clock initialized to 0.
2. **Local Event**: When a process performs an internal event , it increments its own clock.
3. **Message Sending**: Before sending a message, the process increments its clock and sends it along with the message.
4. **Message Receiving**: When a process receives a message, it updates its clock to be the maximum of its current clock and the clock attached to the message, ensuring that it is at least as large as the sender's clock. It then processes the message.



2 Main problems and solutions

Handling Lamport Time

The main problem we are solving is to keep track of the time, like the model above. In the time module, we have functions to increment the time that the worker has when it calls the time function. Clock and time management in time module:

```
%take the time and increment it by 1.
inc(_Name, Time) ->
   Time+1.

% merge the two Lamport time stamps (i.e., take the maximum value)
merge(Timei, Timej) ->
   lists:max([Timei, Timej]).
```

These functions are called whenever a worker does anything basically.
When the worker sends a message to another worker, it looks like this. The clock is incremented before a message is sent:

```
      % Increment the clock before sending the message
      NewClock = time:inc(Name, Clock),
      Message = {hello, rand:uniform(100)},
      % Send the message with the updated clock
      Selected ! {msg, NewClock, Message},
```

And when a message is received, the time is incremented with also considering the time of the sending worker.

```
         % Message received from another worker
          % Merge the current clock with the received message's clock
          NewClock = time:inc(Name, time:merge(Clock, ReceivedTime)),
          % Log the message with the updated clock
          Log ! {log, Name, NewClock, {received, Msg}},
```

Handling safety (the holdback queue)

Another challenge is ensuring that messages are printed or delivered in the correct order. Since messages can arrive out of order due to network delays, a process may receive a message with a timestamp that indicates it should be processed after another event that hasn't arrived yet.

A holdback queue is implemented in the logger process to hold messages that are not yet

safe to print. Each process maintains a clock for each peer (using the `time` module) and only delivers messages when it is safe to do so. A message is considered "safe" if all other events that should come before it in order have been processed.

The logger checks each message's timestamp and compares it with the latest timestamps from other workers. If a message's timestamp is too large (there might still be earlier messages to process), it is held back in the queue until it is safe to print.

```
deliver_safe(_Clock, [], Acc) ->
   {lists:reverse(Acc), []}; % Return accumulated safe messages

deliver_safe(Clock, [{Time, From, Msg} | Rest], Acc) ->
   case time:safe(Time, Clock) of
       true ->
           % If the message is safe, add it to the accumulated list
           deliver_safe(Clock, Rest, [{Time, From, Msg} | Acc]);
       false ->
           % If not safe, hold it back
           {lists:reverse(Acc), [{Time, From, Msg} | Rest]}
   end.
```

When running the test module, it seems to be working as expected, i.e. the time is increasing as messages are being passed between the workers. The holdback queue isn't shown in the logs, but the fact that messages are being logged in the correct order (based on timestamps) suggests that the queue is doing its job of holding back messages until it is safe to print them.

```
3> test2:run(20,2).
log: 2 paul {received,{hello,88}}
log: 3 ringo {received,{hello,64}}
log: 4 john {received,{hello,24}}
log: 6 george {received,{hello,16}}
log: 8 ringo {received,{hello,83}}
log: 8 george {received,{hello,57}}
log: 10 john {received,{hello,90}}
log: 10 george {received,{hello,16}}
log: 11 george {received,{hello,60}}
log: 12 john {received,{hello,16}}
log: 12 george {received,{hello,15}}
log: 14 ringo {received,{hello,73}}
log: 15 ringo {received,{hello,89}}
log: 16 ringo {received,{hello,100}}
log: 18 john {received,{hello,97}}
```