HW Report 3: Loggy – A Logical Time Logger

Ayushman Khazanchi

September 29th, 2021

1 Introduction

In this report we learn how to use a logger based on logical time. We implement a logging procedure that takes random log events from a set of workers, processes them, and displays them in the correct order based on their Lamport time stamps, and later on using Vector Clocks.

2 Main Problems and Solutions

2.1 Random Reception

Initially when we setup the logger and workers with the test module we see that the events are received in a randomised order introduced by the Jitter function which provides a small delay to the worker process. The output from the logger is not ordered at all but we can at least see the events appearing in our output. Our first task here is to introduce Lamport time stamps to the workers so that we can bring some semblance of order to the event output. The output below shows our initial problem. The logger and workers are setup and the events are being printed but messages are at times "received" by the receiving worker even before they are "sent" by the mailing worker. This is obviously incorrect.

```
1> test:run(1000,100).
log: na ringo {received,{hello,31}}
log: na john {sending,{hello,50}}
log: na paul {received,{hello,50}}
log: na george {received,{hello,31}}
log: na john {sending,{hello,1}}
```

2.2 Ordered Disorder and Safety Checks

However, we soon realize that even with Lamport time stamps the order is only partial. This is, in a way, a sort of ordered disordered. We see from the output below that when we introduce Lamport time stamps in our workers we have the output appearing ordered by their Lamport time stamps but within the same time stamp for two workers we can still run into the wrong order for sending/receiving events.

```
3> test:run(1000,100).
log: 1 ringo {received,{hello,31}}
log: 2 john {received,{hello,50}}
log: 1 paul {sending,{hello,50}}
log: 1 george {sending,{hello,31}}
log: 3 john {sending,{hello,1}}
```

We realize that each worker above manages a time stamp now but that the logger can still print out events too early. This can sometimes result in messages being in the right order in the overall Lamport time stamp but in the wrong order in an individual timestamp. We were able to identify this by seeing that some "Sending" messages were appearing after the "Received" messages even though they were correctly ordered by their Lamport time stamps. This part can be fixed by implementing the safe function that ensures that the logger holds messages in a holdback queue and "checks" whether they're safe to print or not.

2.3 Vector Clocks

We realize that each worker above only knows about its own Lamport time stamp. This problem can be solved by using Vector Clocks where Vector Clocks allow us to pass the entire array of timestamps of all workers presently running in the system. This way each worker has the timestamp of all other workers and in the time module we can sift through these timestamps to figure out which event came first. With the implementation of Vector Clocks, the output now looks much cleaner, consistent, and truly ordered.

3 Time Module Implementation

The first three functions of the time module are pretty standard. We initialize a new worker with Lamport time set to zero (zero function). The inc function increments a value of one with each new event that is sent or received by the worker. For receiving a message we also conduct a merge that ensures the Lamport time stamp of the worker is updated to use the "max" of the two values before conducting an increment on this value.

The bulk of the ordering happens in the rest of the functions shown below.

```
%clock(Nodes) : return a clock that can keep track of the nodes
clock(Nodes) ->
    lists:map(fun(Node) -> {Node, 0} end, Nodes). % Initialize each Node to zero
for start
```

The *clock* function takes a list of Nodes and initializes each Node to zero. This is our starting point for the ordering.

```
%• 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
update(Node, Time, Clock) ->
    lists:keyreplace(Node, 1, Clock, {Node, Time}).
```

The *update* function then takes a new Node, a Time, and the list of nodes maintained in Clock. It searches for the Node by using the keyreplace function and if the Node is found it updates it with its new Lamport time stamp.

```
%• safe(Time, Clock) : is it safe to log an event that happened at a given time,
true or false
 safe(Time, Clock) ->
    % sort UpdatedClock first so least value is at the start
    UpdatedClock = lists:keysort(2, Clock), % sort Clock (list of nodes) on Time
(position 2)
    [{_, Counter} | _] = UpdatedClock, % pull out Time/Counter from UpdatedClock
    leq(Time, Counter). % and check against Time coming from loggy using leq
```

The safe function is used by the loggy holdback queue to check if a message is safe to print. We do this in an easy manner by first sorting our list of Nodes (Clock) and putting it in UpdatedClock variable. Then we get the "Time" of the first entry of the UpdatedClock and compare it with the Time that we're passing into the function. This function is easier to understand by looking at it in conjunction with the checkSafe function in loggy.erl.

```erlang
% recursively check
checkSafe(Clock, Queue) ->
    % Remove first tuple from Queue and check against it
    [{From, Time, Msg} | Rest] = Queue,
    % base condition of when queue is empty
    if Queue == [] ->
        [];
    true ->
        case time:safe(Time, Clock) of
            % if Time in removed tuple is less than Clock, log it
            true ->
                log(From, Time, Msg),
                checkSafe(Clock, Rest);
            false ->
                % return Queue eventually to remove all the successful entries
                Queue
        end
    end.
```

With the above in place, we're able to see an ordered output appearing from the logger as it has more logic in place now to order events and ensure they're safe to print. The output below shows the implementation being a success.

```
16> test:run(1000,100).
log: 1 ringo {sending,{hello,31}}
log: 1 john {sending,{hello,50}}
log: 1 paul {received,{hello,50}}
log: 1 george {received,{hello,31}}
log: 2 john {sending,{hello,1}}
log: 2 ringo {received,{hello,1}}
log: 3 paul {sending,{hello,61}}
log: 3 george {received,{hello,61}}
log: 4 paul {sending,{hello,68}}
log: 4 ringo {received,{hello,68}}
log: 5 paul {sending,{hello,80}}
log: 5 george {received,{hello,80}}
```

We can also see the output from the implementation of the Vector Clocks is also ordered with the correct time stamps.

```
24> vtest:run(1000,100).
log: [{john,0},{paul,0},{ringo,1},{george,0}] ringo {sending,{hello,31}}
log: [{john,1},{paul,0},{ringo,0},{george,0}] john {sending,{hello,50}}
log: [{john,1},{paul,1},{ringo,0},{george,0}] paul {received,{hello,50}}
log: [{john,1},{paul,2},{ringo,0},{george,0}] paul {sending,{hello,61}}
log: [{john,1},{paul,2},{ringo,0},{george,1}] george {received,{hello,61}}
log: [{john,1},{paul,2},{ringo,0},{george,2}] george {sending,{hello,40}}
log: [{john,1},{paul,2},{ringo,2},{george,2}] ringo {received,{hello,40}}
log: [{john,1},{paul,2},{ringo,1},{george,3}] george {received,{hello,31}}
log: [{john,2},{paul,0},{ringo,0},{george,0}] john {sending,{hello,1}}
log: [{john,2},{paul,2},{ringo,3},{george,2}] ringo {received,{hello,1}}
log: [{john,1},{paul,3},{ringo,0},{george,0}] paul {sending,{hello,68}}
log: [{john,2},{paul,3},{ringo,4},{george,2}] ringo {received,{hello,68}}
log: [{john,1},{paul,2},{ringo,1},{george,4}] george {sending,{hello,76}}
log: [{john,3},{paul,2},{ringo,1},{george,4}] john {received,{hello,76}}
log: [{john,1},{paul,4},{ringo,0},{george,0}] paul {sending,{hello,80}}
log: [{john,1},{paul,4},{ringo,1},{george,5}] george {received,{hello,80}}
log: [{john,1},{paul,5},{ringo,0},{george,0}] paul {sending,{hello,7}}
log: [{john,4},{paul,5},{ringo,1},{george,4}] john {received,{hello,7}}
```

4 Conclusions

This was an interesting report to understand logical time and Vector Clocks and events can be logged in real-time without the need for physical time. It also helps to understand why logical time is a better implementation than using physical time because physical time can be different on different workers and thus unreliable as a logical logging tool. The report also helps to understand how to use a module purely through its API implementation.