HW3: Loggy - A Logical Time Logger

Jian Wen

September 26, 2023

1 Introduction

1.1 Update Clock

For this assignment, I have known about and dealt with the time synchronization in a distributed system. In such a distributed system, it's common to encounter a problem that the clock and progress is vary between different nodes. Such problem may cause parameters misconfigure and lead to bigger disaster. To tackle this problem, Lamport clock is created to order events from different nodes based on logical time.

2 Main problems and solutions

2.1 Update clock on each node

One of the objective in this assignment is to use time to "mark" messages. This parameter is append to the message based on the clock on each node. Then, when should we update the clock?

As a solution, I have configured the workers' nodes to update the clock when receiving or sending messages. At the logger node, we will have a set of clocks to track each worker. The corresponding clock will be updated when receiving a new log request from that worker. In a word, on each node in our logger, the clock will be updated when a new interaction is happening.

What is different between the worker node and the logger node is that, the clock update at the worker self-motivated. The clock at the logger will be passively updated as respond to requests from workers.

```
loop(LogClock, HoldbackQueue, MaxSize) ->
  receive
    {log, From, Time, Msg} ->
      NewLogClock = time:update(From, Time, LogClock),
      % Update the LogClock for that node.
      ...
```

```
loop(Name, Log, Peers, Sleep, Jitter, LocalTime)->
  ...
  receive
    {msg, Time, Msg} ->
      NewTime = time:inc(Name, time:merge(LocalTime, Time)),
      % Once receive, update local timer.
      ...
  after Wait ->
    ...
    NewTime = time:inc(Name, LocalTime),
    % When sending, update local timer as well.
    ...
```

2.2 Set up holdback queue for logs

Now we have all the workers tag messages with a timer from their local clock, and logger is updating the clocks for each workers to track their timestamps. Then it's time to decide whether to print a log information based on what we have. To be specific, to construct and make use of the holdback queue.

For every newly arrived message, it will be first put into our holdback queue. Then we sort this queue by the timestamp of those messages. However, the queue is still not completely safe to print. We need to compare the timestamp of messages in the queue with the corresponding tracking clock. Here is how we check: if the timestamp is prior than the tracking clock + 1, then we allow it to be printed. Once we have a message coming from beyond, we keep the rest messages in the holdback queue. The reason we compare in this way is to make sure those messages arriving in advance can wait for the messages that should be printed before it.

```
loop(LogClock, HoldbackQueue, MaxSize) ->
  receive
    {log, From, Time, Msg} ->
      NewLogClock = time:update(From, Time, LogClock),
      % Update the LogClock for that node.
      NewQueue = lists:append(HoldbackQueue, [{From, Time, Msg}]),
      % Add this log into queue.
      UnsafeQueue = queue_guard(NewQueue, NewLogClock),
      % For each entry, if safe: log; if unsafe: keep in queue.
      ...

queue_guard([], _) ->
  [];
queue_guard(Queue, LogClock) ->
  SortedQueue = lists:keysort(2, Queue),
```

```
[{From, Time, Msg} | T] = SortedQueue,
case time:safe(Time, LogClock) of
  true ->  % For safe msg, just log it.
    log(From, Time, Msg),
    queue_guard(T, LogClock);
  false ->
    SortedQueue  % The log left are unsafe (beyond LogClock).
end.
```

3 Evaluation

The evaluation is performed with the test.erl file. To test with multiple initial states, it has been modified a bit to allow manually configured rand seeds. The figure 1 shows the result of my implementation.

The maximun length of holdback queue is 16.

```
12> test:run(1400, 300).
log: 1 paul {sending,{hello,70}}
log: 1 george {sending,{hello,44}}
log: 2 john {received,{hello,70}}
log: 2 ringo {received,{hello,44}}
log: 2 paul {sending,{hello,70}}
log: 2 george {sending,{hello,44}}
log: 3 ringo {sending,{hello,53}}
log: 3 george {sending,{hello,44}}
log: 3 paul {sending,{hello,70}}
log: 4 john {received,{hello,53}}
log: 4 ringo {received,{hello,44}}
log: 4 george {sending,{hello,44}}
log: 4 paul {sending,{hello,70}}
log: 5 john {received,{hello,70}}
log: 5 ringo {sending,{hello,53}}
log: 6 john {received,{hello,53}}
log: 6 ringo {received,{hello,44}}
log: 7 ringo {sending,{hello,53}}
log: 8 john {received,{hello,53}}
log: 8 ringo {received,{hello,44}}
log: 9 john {received,{hello,70}}
log: 9 ringo {sending,{hello,93}}
log: 10 george {received,{hello,93}}
log: 10 ringo {sending,{hello,93}}
log: 10 john {sending,{hello,48}}
log: 11 george {received,{hello,93}}
log: 11 ringo {sending,{hello,93}}
log: 11 paul {received,{hello,48}}
log: 11 john {received,{hello,70}}
log: 12 george {received,{hello,93}}
log: 12 ringo {sending,{hello,93}}
Max holdback queue length: 16
stop
```

Figure 1: Test result using (1400, 300, default seeds)

4 Conclusion

According to the result in figure 1, the logs are printed correctly, with all messages sorted by their timestamps. In one of my previous course called interactive system, I have tried to implement a game played on several computers. One of the important tasks to do is to make sure each player get same updates across different devices. Actually, synchronization of time is extremely important in a distributed time. Time is an important measurement when we want to confirm the status of different nodes.