Report 3: Loggy - a logical time logger

Emile Le Gallic

September 26, 2024

1 Introduction

In this seminar, we focused on implementing a logging procedure that deals with log events from a set of distributed workers. The main challenge was to ensure the correct ordering of these events based on logical time, specifically using Lamport timestamps. This exercise taught us how to handle and order events in a distributed system where global time coordination is difficult or impossible due to system latency and other issues.

We also touched on how the use of vector clocks could improve the system's handling of concurrency and causality. While Lamport timestamps provide a simple way to order events, they do not capture the full causality of the events. In contrast, vector clocks allow each worker to maintain a more comprehensive view of the state of the system by tracking the knowledge of the entire system, not just event ordering.

2 Main problems and solutions

During the implementation of the distributed logging system, the primary challenge was ensuring the correct ordering of log messages based on logical time. This exercise introduced two major implementations: one using **Lamport timestamps** and another using **vector clocks**.

2.1 Problems Encountered and Solutions

2.1.1 Ordering of Log Messages

The first challenge was correctly ordering log messages received from different workers in a distributed environment. The initial implementation used **Lamport timestamps**, but since Lamport timestamps only provide a partial ordering, the system couldn't fully capture the causality between events.

Solution: We implemented the `safe/2` function in `time.erl` to ensure that messages are only logged when it is "safe," meaning the message's timestamp is consistent with the clock's current state. This solution worked well but did not provide detailed information on the causality between events from different nodes.

2.1.2 Causality Tracking with Vector Clocks

When we switched to **vector clocks** (in `vect.erl`), we faced the issue of maintaining the causality between events across multiple nodes. The vector clocks required more sophisticated merging and comparison operations between timestamps.

Solution: We implemented the `merge/2` and `leq/2` functions in `vect.erl` to manage the partial ordering and merge vector timestamps from different nodes. The `update/3` function was also crucial to update the local clock whenever a new log message was received from another node. This solution gave us better causality tracking.

```
merge([{Name, Ti} | Rest], Time) ->
   case lists:keyfind(Name, 1, Time) of
      {Name, Tj} ->
         [{Name, max(Ti, Tj)} | merge(Rest, lists:keydelete(Name, 1, Time))
             ];
      false ->
         [{Name, Ti} | merge(Rest, Time)]
   end.
```

This merging process ensures that the vector clock is updated correctly when log messages from different nodes are received.

2.1.3 Queue Management for Log Messages

The logs had to be queued until they could be safely logged. With Lamport timestamps, this was easier as we only needed to check whether the current timestamp was safe to log. However, with vector clocks, checking for safety (`safe/2`) became more complex, as we needed to compare entire vectors rather than just scalar timestamps.

Solution: In both implementations (`time.erl` for Lamport and `vect.erl` for vector clocks), we implemented a `log_queue/3` function to manage the logs in a queue and only log them when safe. The function folds over the list of log messages, checking the safety condition before logging:

```
lists:foldl(fun({T, F, M}, Acc) ->
   case vect:safe(T, Clock) of
      true ->
         io:format("log: ~w ~w ~p~n", [T, F, M]),
         Acc;
      false ->
         [{T, F, M} | Acc]
   end
end, [], LogsQueue).
```

This ensured that logs were processed only when the causality and ordering were correct.

2.2 Conclusion

The major challenge was maintaining the correct order of log messages across distributed nodes. Initially, **Lamport timestamps** were sufficient for partial ordering, but they did not capture full causality. Switching to **vector clocks** provided more accurate tracking of the relationships between events, allowing us to properly manage concurrency and causality in the logging system.

In summary:

- **Lamport timestamps** were easy to implement but limited in tracking causality.
- **Vector clocks** provided better causality tracking but required more complex operations for merging and safety checks.

3 Evaluation

Part 1: *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.*

3.1 Naive implementation (no clock)

In this part, we examine the log messages produced by running the system without any clock synchronization mechanism (neither Lamport clocks nor vector clocks). The log is expected to sometimes display messages in the wrong order due to the absence of time ordering guarantees in the distributed system.

3.1.1 Identifying Logs in the Wrong Order

To identify if the log messages are printed in the wrong order, we focus on the sequence of events received and sent by each worker process. Since the system does not enforce any global or local ordering of events, there can be inconsistencies where a node might print a message indicating it received a message before it sent one.

For example, in one of the runs, the following logs can be observed:

```
log: 16:37:3 ringo {sending,{hello,53}}
log: 16:37:3 john {received,{hello,53}}
log: 16:37:4 ringo {sending,{hello,53}}
log: 16:37:4 john {received,{hello,53}}
```

In this case, it appears that John received the message with a timestamp of `16:37:3` before Ringo sent it, also at `16:37:3`. This is a case of a log being printed in the wrong order. The mismatch between the send and receive times demonstrates that without a clock to order events, the system cannot maintain causality.

3.1.2 Effect of Jitter on Log Order

Jitter simulates network latency by introducing random delays into message delivery and execution. We experimented with various jitter values and observed its effect on the frequency of wrong log entries.

- **High jitter:** As expected, increasing jitter causes more messages to be delayed unpredictably, which makes it more likely that messages are printed out of order. With a jitter of 1000 milliseconds, we saw many instances of incorrect log ordering, similar to the one mentioned earlier.

- **Low jitter:** Reducing jitter decreases the likelihood of out-of-order logs, but it does not eliminate them entirely. Even with a jitter of 500 milliseconds, some messages were printed in the wrong order, although the frequency was lower compared to high jitter.

- **No jitter:** In the absence of jitter (i.e., jitter set to 0), fewer log entries appear out of order because the system processes events more deterministically. However, due to the asynchronous nature of the worker processes and their randomized sleep periods, wrong orderings can still occur.

Part 2: *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.*

3.2 Lamport clock implementation

3.2.1 Out of Order Detection in the First Implementation

Yes, entries were detected out of order in the first implementation. This was observed by noting that events which should have happened sequentially were logged in reverse order. Specifically, messages received by a node before another node had sent them were a clear indicator of incorrect ordering. For example, when a node logs a message as ”received” before the ”sending” log from the originating node, we can be certain that the logs are printed out of order.

3.2.2 Lamport Clock Implementation

The final log, after implementing the Lamport clock and message holdback queue, shows that events are now ordered correctly according to Lamport's logical clock. This means that messages are not processed by a node until the clock value of the sending event is correctly synchronized, ensuring that no message is received before it has been sent. The log reflects a consistent sequence of events, where each node's local time is respected and there are no premature ”received” messages.

3.2.3 Order of Events in the Log

In the final implementation, the log is reflective of the actual event sequence, as the use of Lamport timestamps ensures that events are processed in logical order. However, while the log presents the events in the order they happened in the system, it does not necessarily represent the real-time chronological order due to the message delay and system jitter. Instead, it shows the causal relationships between events.

3.2.4 Maximum Holdback Queue Size

The maximum holdback queue size varied during testing, depending on the jitter and message delay. For example, in tests with a jitter of 1000ms and message delays ranging from 100ms to 2000ms, the queue size grew to a maximum of 18 entries. The holdback queue size is directly related to the amount of jitter in the system, as larger jitter values result in more messages arriving out of order, which then need to be held back until they can be processed in the correct order.

Tests with different parameters showed that the holdback queue size could fluctuate depending on the message frequency and delay parameters. Here are some results:

- Test 1 (jitter = 1000ms, delay = 2000ms): Maximum queue size = 6
- Test 2 (jitter = 1000ms, delay = 1000ms): Maximum queue size = 14
- Test 3 (jitter = 1000ms, delay = 500ms): Maximum queue size = 18
- Test 4 (jitter = 1000ms, delay = 100ms): Maximum queue size = 11

This indicates that the larger the delay or jitter, the more entries accumulate in the holdback queue, as more messages arrive out of order.

Part 3: *If you run tests using the Lamport clock and the vector clock you might see some differences. You could augment your code of the logger and do a print out every time it queues a message in the hold back queue. Is there a difference?Another thing to note is that the logger does not know a forehand how many nodes we have in the system so we can start new nodes as we go. You can start a set of workers and then start another set that partly overlap the first set. The logger will still work and log events in an happen-before order.*

3.3 Vector clock implementation

When comparing the logs generated using Lamport Clocks and Vector Clocks, we observe several key differences.

3.3.1 Hold-back Queue Behavior

In the case of the Lamport clock, messages are queued in the hold-back queue when their timestamps indicate they arrived before an earlier message is processed. For instance, if node *ringo* sends a message with a lower Lamport timestamp than one already processed, it is held back until the system catches up.

With the vector clock, a message is held back if its vector clock is not causally consistent with the vector clock of the receiving process. This means that if one process has a newer event for another node than the one reflected in the vector timestamp of the incoming message, the message is held back.

By augmenting the logger to print every time a message is queued in the hold-back queue, you can observe that the vector clock results in more complex hold-back queueing behavior. Messages can be delayed if any part of the vector clock is not causally consistent with the local clock, whereas in the Lamport clock, it's based on a single integer timestamp. This leads to potentially fewer messages being queued under the Lamport clock than the vector clock.

3.3.2 Dynamic Node Addition

The logger does not assume knowledge of how many nodes are present in the system from the start, which is an important feature. You can dynamically start new nodes, and the logger will continue to work without any issues.

For example, you could initially start a set of workers, say *john*, *paul*, *ringo*, and *george*, and later start new nodes that overlap with these existing ones. The logger will adjust and ensure that events are logged in the correct happen-before order, regardless of whether you use the Lamport clock or vector clock.

This dynamic adjustment is more straightforward with the Lamport clock, as it relies on a single global counter that can be adjusted as new events are added. With the vector clock, the addition of new nodes requires extending the vector for each node, adding a level of complexity. However, once the vectors are updated, the system continues to respect the causal relationships as indicated by the vector timestamps.

3.3.3 Conclusion

In conclusion, while both clocks achieve the same goal of preserving causal ordering, the vector clock provides more granular control over the causal relationship between events, at the cost of greater complexity in the hold-back queueing process. The Lamport clock is simpler and works with a global scalar timestamp, making it easier to integrate new nodes dynamically. However, it lacks the fine-grained causality tracking of vector clocks.

4 Conclusions

In exploring Lamport and vector clocks, I learned how these mechanisms manage event ordering in distributed systems. Lamport clocks provide a simpler way to establish global order, while vector clocks offer richer causal tracking but add complexity. This understanding is crucial for designing consistent and reliable distributed systems, as it highlights the trade-offs between simplicity and the need for nuanced event handling.

5 Fixing the Lamport Clock

Following first report, I fixed the Lamport Clocks and here are the things I changed :

5.1 `update` function

I changed the `update` function from :

```
update(Node, Time, Clock) ->
   dict:store(Node, merge(Time, dict:fetch(Node, Clock)), Clock).
```

to :

```
update(Node, Time, Clock) ->
   dict:store(Node, Time, Clock).
```

The change removed the `merge/2` step, so now the function simply stores the incoming Time directly in the Clock, without comparing it with the existing value

5.2 `safe` function

I changed the `safe` function from :

```
safe(Time, Clock) ->
   dict:fold(fun(_, T, Acc) ->
      leq(Time, T) and Acc
   end, true, Clock).
```

to :

```
safe(Time, Clock) ->
   dict:fold(fun(_, T, Acc) ->
      leq(T, Time) and Acc
   end, true, Clock).
```

I inverted the two times leading to a wrong ordering of the messages.

5.3 Updating time when receiving a message from the worker

I changed this part of the `loop` function from :

```
   receive
      {msg, TimeRcv, Msg} ->

         case Config of
            % Part 1: Just the time the message was received
            1 ->
               NewTime = TimeRcv;
            % Part 2: Lamport time
            2 ->
               NewTime = time:merge(TimeRcv, Time);
            % Part 3: Vector clock
            3 ->
               NewTime = vect:merge(TimeRcv, Time)
```

```
        end,

        Log ! {log, Name, NewTime, {received, Msg}},
        loop(Name, Log, Peers, Sleep, Jitter, NewTime, Config);
```

to :

```
  receive
     {msg, TimeRcv, Msg} ->

        case Config of
           % Part 1: Just the time the message was received
           1 ->
              NewTime = TimeRcv;
           % Part 2: Lamport time
           2 ->
              NewTimeTemp = time:merge(TimeRcv, Time),
              NewTime = time:inc(Name, NewTimeTemp);
           % Part 3: Vector clock
           3 ->
              NewTime = vect:merge(TimeRcv, Time)
        end,

        Log ! {log, Name, NewTime, {received, Msg}},
        loop(Name, Log, Peers, Sleep, Jitter, NewTime, Config);
```

In order to first merge the time and then increment the time when receiving a message.

5.4 Testing this new implementation

You can test it yourself using :

```
c(logs).
c(worker).
c(time).
c(test).
c(vect).

test:run({SLEEP_TIME}, {JITTER_TIME}, 2).
```

Where SLEEP_TIME is your custom sleep time and JITTER_TIME your custom jitter time. The number 2 is for the config ID (2 = Lamport Clock).