HW3 Report, Loggy: A Logical Time Logger

David Fischer

September 24, 2025

1 Introduction

Lamport and vector clocks serve as fundamental tools that enable causal ordering in distributed systems. Though implementations might differ, their core principles are still represented in multiple areas such as distributed tracing, message queues, and distributed garbage collection. This assignment's goal was the implementation of *Loggy*, a logging procedure continually receiving random messages on a random delay from workers.

The implementation spans a central logging module with a holdback queue to correctly print messages, workers sending and receiving messages between each other on a random delay with random jitters before reporting to the central logger, two clock modules, Lamport and vector based, and multiple analytics modules to generate insights.

2 Implementation

2.1 Time

The Lamport *time* module contains functions to initialize, update, and review timestamps and clocks. The Lamport implementation handles timestamps as simple numbers and the clock as a map containing the worker names keys and individual timestamps as values. Updating the clock is handled by comparing the current value with the new value for a specified process and then changing the map if the new value is greater. A message is determined as safe to print when the given timestamp is less or equal to the minimum value from the clock map.

The same methodology was applied to the vector (*vect*) module, with the major difference being individual timestamps functioning like the clock from the Lamport implementation. This also makes the `safe/2` function equivalent to the `leq/2` function, as identifying whether a message with vector timestamp is safe to log is checking if the timestamp is causally before or concurrent with the current clock state.

To use both modules without changes to the workers or *Loggy* requiring recompilation, the `apply/3` function was utilized.

2.2 Holdback Queue

The holdback queue in the *Loggy* module was implemented as a simple list, starting out empty initially. Each time a message is received, it is appended to the holdback queue, which is then sorted using the `time:leq/2` function. The sorted list is then split with `lists:partition/2` calling `time:safe/2` using the clock held by the loop and timestamp from each message in the queue. The resulting entries in the `Safe` list are then logged and the `Unsafe` list is passed to the next recursive function call of `loggy:loop/3`.

3 Main problems and solutions

3.1 Validating Order

Confirming the order of messages printed by *Loggy* presented a challenge initially, as issues could be caused by the clock and holdback queue implementation. Additionally, insight into the clock state and actual message times of each individual process was limited. As the randomness of the workers is seeded, a jitter value set to `0` results in the holdback queue being skipped completely and messages being printed instantly when received. This output was then cross-checked with a regular test run. To further confirm total order, even with jitters greater than zero and especially for the vector clock, a *mermaid* state chart graphing module was created as an observability tool.

3.2 Log Parsing

Once the vector clock module was implemented mentally parsing the *Loggy* output posed a greater challenge. To ease the load, control sequences were utilized to format the logged messages in a more coherent and readable way.

```
io:format("log: s:~-2w ~-6s ~-8w (~3w) c:~w~n", [:]).
log: s:0  george sending ( 99) c:john => 5,paul => 8,ringo => 6
```

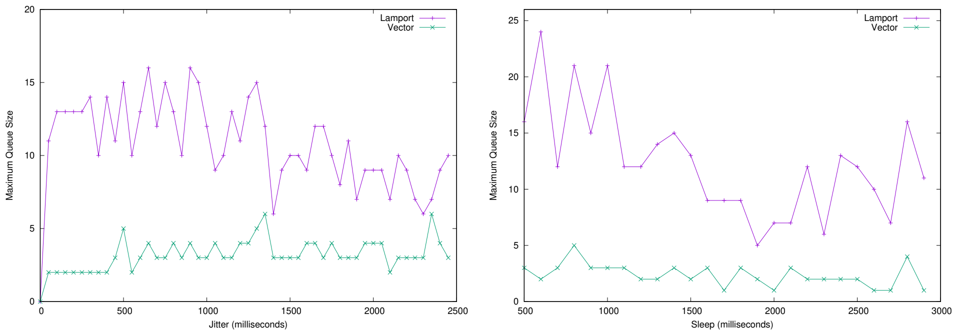
4 Evaluation

4.1 Truths

With this implementation, one can state that causal ordering is always preserved and therefore always true due to the individual timestamps shared by the workers and the implementation of the holdback queue. The `safe/2` function guarantees correctness, as it ensures a message is only printed when no earlier timestamp could arrive. To aid evaluating this, a module *mermaid* was implemented to create visual representations of the messages flowing between the workers including the data and clock states. Examples of these visualizations and corresponding *Loggy* outputs can be found in the Appendix.

It is sometimes true that the total ordering matches the real-time ordering, as due to the delays and artificial jitters in communication with *Loggy*, the order of logged events might not completely match the real order of events. This occurs in situations where events are concurrent, like two independent workers sending a message.

4.2 Queue Size Comparison



(a) Jitter with 1500ms `Sleep` (b) Sleep with 250ms `Jitter`

Figure 1: Queue size comparison on different delays and jitters

Due to the inherent randomness of the *Loggy* implementation these results can't be taken as absolutes for correlating sleep or jitter to the maximum queue size. Still, the difference between the maximum size of the Lamport queue when compared to the vector queue demonstrates the logging throughput being greater, at the cost of larger messages.

Also notable is the bypass of the holdback queue when the `Jitter` is set to zero as can be seen in Figure 1a.

5 Conclusions

In conclusion, the holdback queue is crucial to maintain causal order to guarantee sends being logged before receives, although it cannot guarantee correctness for independent concurrent events. Improvements to the current implementation include replacing `apply/3` with a shared time module to improve performance, as it will always be slower than direct function calls, and implementation of more sophisticated observability as the current *mermaid* module can't correctly represent concurrent send messages.

A Lamport Clock

These tests were completed using `test:run(<module>, 1500, 500)`.

```
119> test:run(time, 1500, 500).
loggy: starting with module time
log: s:5   ringo  sending  ( 24) c:1
log: s:5   john   sending  (  6) c:1
log: s:5   george sending  ( 26) c:1
log: s:2   paul   received ( 24) c:2
log: s:2   john   received ( 26) c:2
log: s:2   paul   received (  6) c:3
log: s:2   john   sending  ( 50) c:3
log: s:2   ringo  received ( 50) c:4
log: s:2   john   sending  ( 73) c:4
log: s:2   paul   sending  ( 28) c:4
log: s:8   george received ( 28) c:5
log: s:8   ringo  sending  (  2) c:5
log: s:8   john   sending  ( 37) c:5
log: s:13  george received ( 73) c:6
log: s:13  paul   received (  2) c:6
log: s:13  john   sending  (  1) c:6
log: s:3   george sending  ( 48) c:7
log: s:3   paul   sending  ( 30) c:7
log: s:3   ringo  received ( 48) c:8
log: s:3   paul   received ( 37) c:8
log: s:3   ringo  sending  ( 86) c:9
log: s:3   george received ( 86) c:10
log: s:3   george received ( 30) c:11
log: s:3   george sending  ( 85) c:12
log: s:3   ringo  received ( 85) c:13
log: s:3   ringo  sending  ( 83) c:14
log: s:3   george received ( 83) c:15
log: s:3   ringo  received (  1) c:15
```

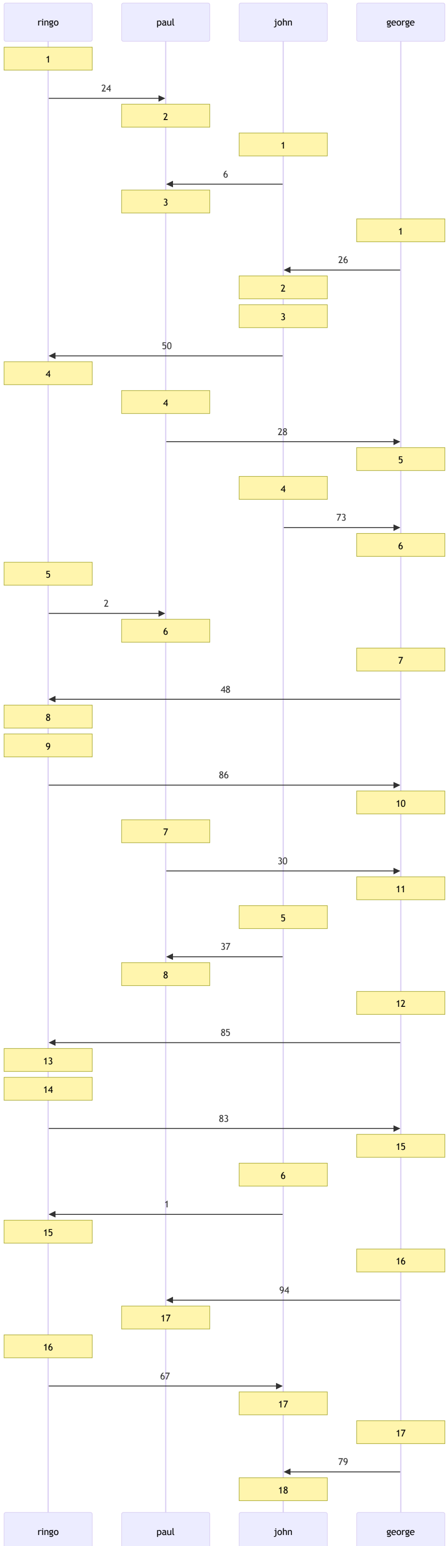


Figure 2: Sequence visualization of the Lamport timestamp algorithm

B Vector Clock

```
120> test:run(vect, 1500, 500).
loggy: starting with module vect
log: s:1   ringo  sending  ( 24) c:ringo => 1
log: s:1   paul   received ( 24) c:paul => 1,ringo => 1
log: s:0   john   sending  (  6) c:john => 1
log: s:0   paul   received (  6) c:john => 1,paul => 2,ringo => 1
log: s:0   george sending  ( 26) c:george => 1
log: s:0   john   received ( 26) c:john => 2,george => 1
log: s:0   john   sending  ( 50) c:john => 3,george => 1
log: s:0   ringo  received ( 50) c:john => 3,ringo => 2,george => 1
log: s:2   john   sending  ( 73) c:john => 4,george => 1
log: s:0   paul   sending  ( 28) c:john => 1,paul => 3,ringo => 1
log: s:0   george received ( 28) c:john => 1,paul => 3,ringo => 1,george => 2
log: s:0   george received ( 73) c:john => 4,paul => 3,ringo => 1,george => 3
log: s:0   ringo  sending  (  2) c:john => 3,ringo => 3,george => 1
log: s:0   paul   received (  2) c:john => 3,paul => 4,ringo => 3,george => 1
log: s:0   george sending  ( 48) c:john => 4,paul => 3,ringo => 1,george => 4
log: s:0   ringo  received ( 48) c:john => 4,paul => 3,ringo => 4,george => 4
log: s:1   john   sending  ( 37) c:john => 5,george => 1
log: s:1   paul   received ( 37) c:john => 5,paul => 5,ringo => 3,george => 1
log: s:0   ringo  sending  ( 86) c:john => 4,paul => 3,ringo => 5,george => 4
log: s:0   george received ( 86) c:john => 4,paul => 3,ringo => 5,george => 5
log: s:2   george sending  (  8) c:john => 4,paul => 3,ringo => 5,george => 6
log: s:1   ringo  sending  ( 84) c:john => 4,paul => 3,ringo => 6,george => 4
log: s:1   paul   received ( 84) c:john => 5,paul => 6,ringo => 6,george => 4
log: s:1   paul   received (  8) c:john => 5,paul => 7,ringo => 6,george => 6
log: s:1   paul   sending  ( 46) c:john => 5,paul => 8,ringo => 6,george => 6
log: s:1   george received ( 46) c:john => 5,paul => 8,ringo => 6,george => 7
log: s:1   john   sending  (  1) c:john => 6,george => 1
log: s:1   ringo  received (  1) c:john => 6,paul => 3,ringo => 7,george => 4
log: s:0   george sending  ( 99) c:john => 5,paul => 8,ringo => 6,george => 8
log: s:0   paul   received ( 99) c:john => 5,paul => 9,ringo => 6,george => 8
```

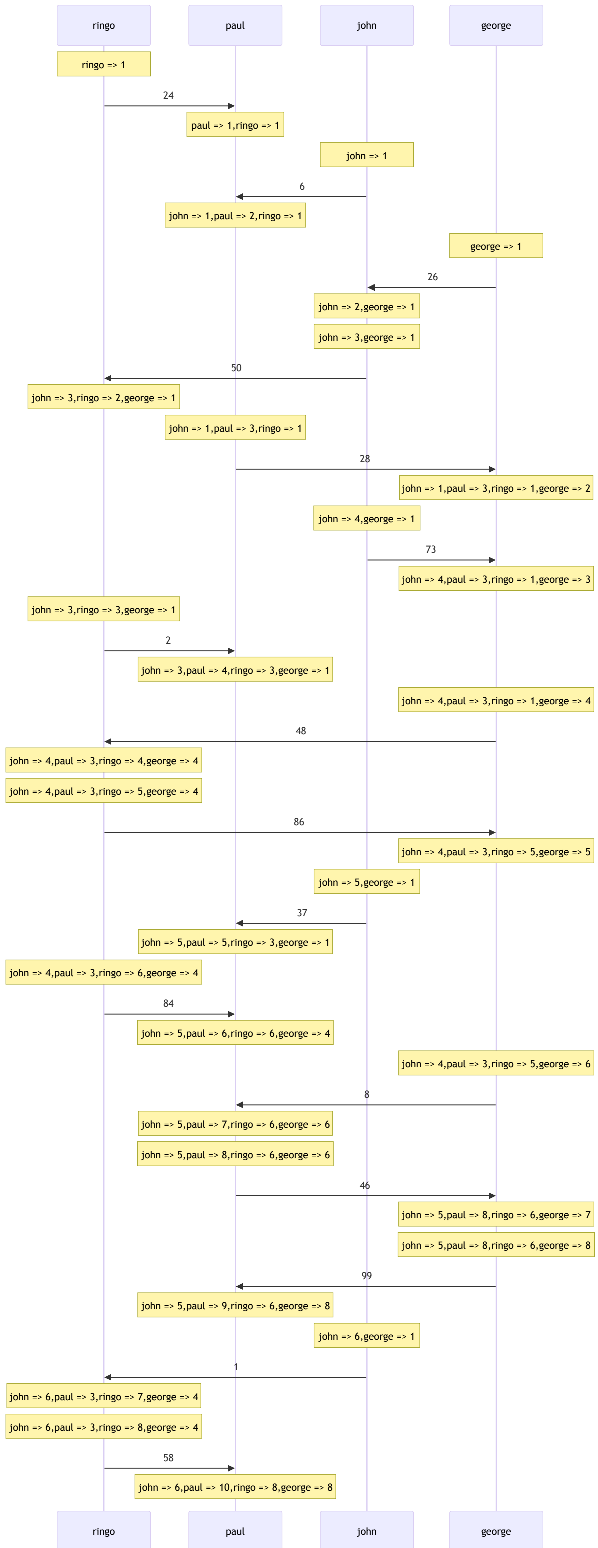


Figure 3: Sequence visualization of the vector clock implementation