HW3 Report, Loggy: A Logical Time Logger

David Fischer

September 23, 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 assignments 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 Main problems and solutions

2.1 Validating Order

Confirming the order of messages printed by *Loggy* presented a challenge initially, as issues could be caused 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.

2.2 Holdback Queue Implementation

2.3 Log Parsing

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

```
io:format("log: s:~-3w ~-6s ~-8w (~3w) c:~w~n",
          [Size, From, Act, Msg, Time]).
% log: s:0   john   sending  (  6) c:john => 1
% log: s:0   paul   received ( 26) c:john => 1,paul => 1
```

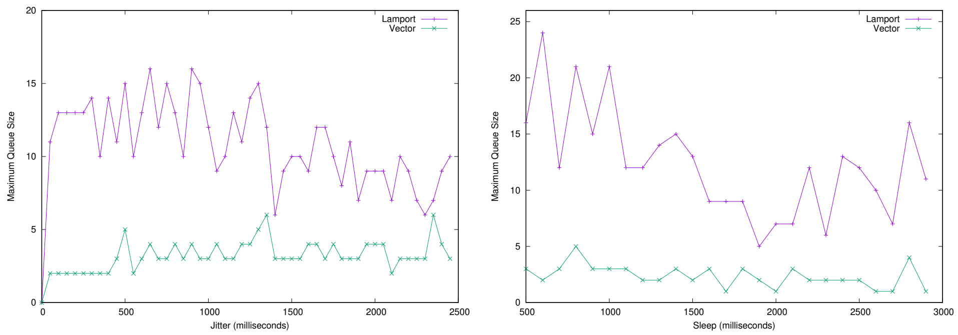
3 Evaluation

3.1 Visualization

To aid evaluation, a module *mermaid* was implemented to create visual representations of the messages flowing between the vector clocks.

See appendix. (TODO, proper sendoff to the appendix, google how)

3.2 Queue Size Comparison



(a) Comparison of Jitter using a 1500ms Sleep

(b) Comparison of Sleep using a 250ms Jitter

Figure 1: Queue size comparison between the clock implementations on different delays and jitters

Due to the inherit randomness of the *Loggy* implementation these results can't be taken as absolutes for corelating 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.

4 Conclusions

An interesting part of this *Loggy* implementation was the similarity between the full Lamport clock and the individual times held by the vector clock workers. Concluding, the vector implementation is similar to the Lamport implementation at its core due to the shared methodology of incrementing a simple value to represent [the state in time]

5 Appendix

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

5.1 Lamport Clock

```
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
```

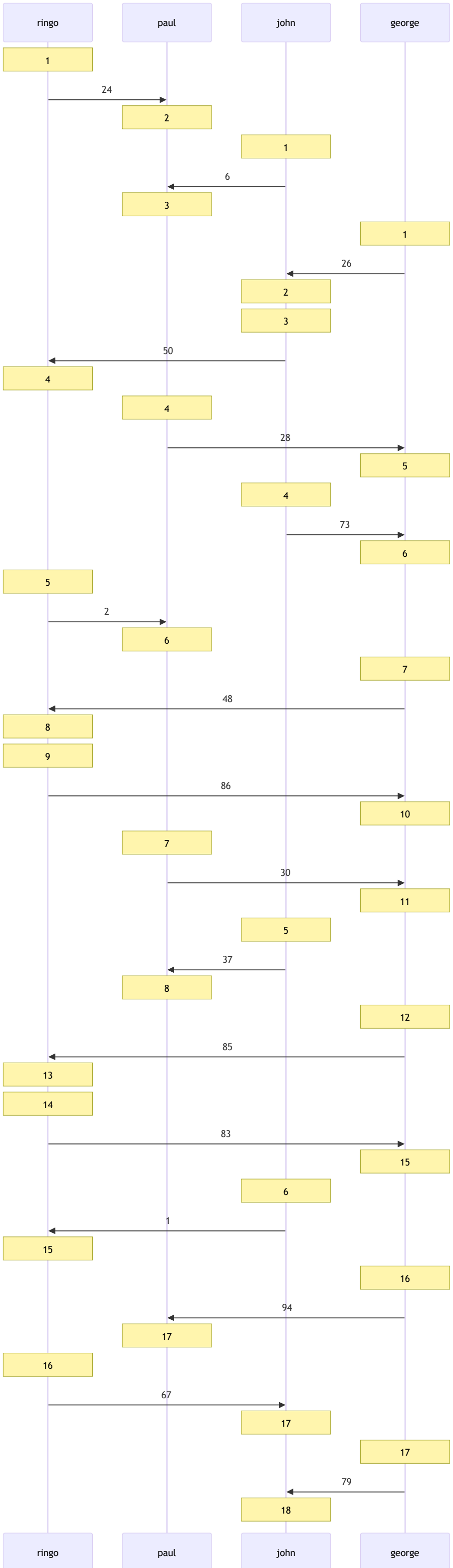


Figure 2: Sequence visualization of the Lamport timestamp algorithm

5.2 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
```

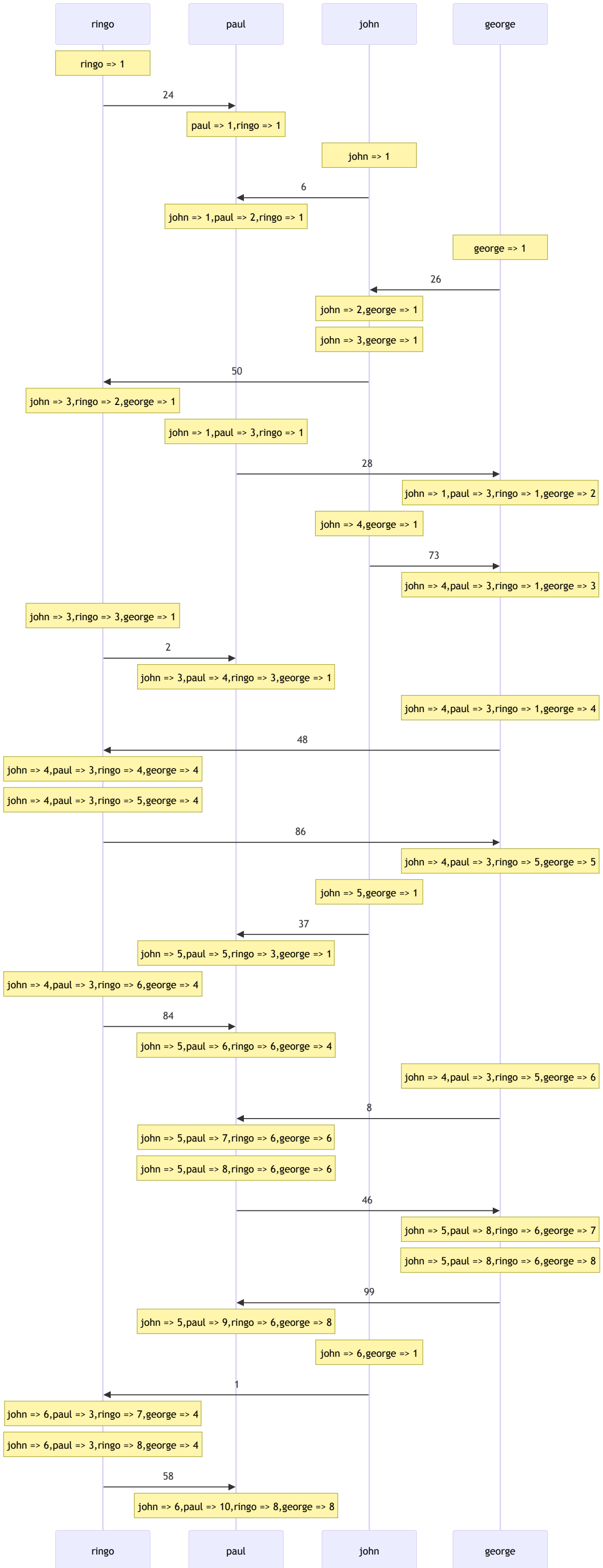


Figure 3: Sequence visualization of the vector clock implementation