Report 3: Loggy - a logical time logger

Md Sakibul Islam

September 24, 2025

1 Introduction

This report covers the implementation of Loggy, a distributed logging system that uses logical time to ensure causal ordering of events across multiple worker nodes. The system addresses the fundamental challenge of maintaining correct event ordering in distributed environments where message delays and processing times can cause inconsistencies. This work explores both Lamport clocks and Vector clocks, demonstrating their effectiveness in preserving causality in distributed systems.

2 Main problems and solutions

2.1 Initial Implementation Without Logical Time

The initial implementation without logical time suffered from causal ordering violations. Workers sent messages with dummy timestamps (na) and logged events immediately, leading to scenarios where:

- Messages from different workers arrived at the logger out of sequence due to network delays
- Receive events were logged before their corresponding send events when high jitter was introduced

Detection Method: Out-of-order entries were identified by running tests with varying Sleep and Jitter parameters and tracking message correlations through unique random numbers in message content.

2.2 Lamport Clock Solution

The solution involved implementing logical time using Lamport timestamps and a holdback queue in the logger:

Time Module API:

The `time` module abstracted Lamport clock operations:

- `zero/0`, `inc/2`, `merge/2`, `leq/2` for basic timestamp operations.
- `clock/1`,`update/3`, `safe/2` for logger clock management.

Worker Modifications: Workers maintained logical clocks, incrementing on sends and merging on receives, ensuring proper causality tracking.

Logger Implementation: Used a sorted holdback queue that only delivered messages when they became "safe" (all nodes had advanced beyond the message's timestamp).

3 Results:

The final implementation ensured that all log entries were printed in increasing timestamp order, with send events always having lower timestamps than their corresponding receive events. For example:

```
log: 1 george sending,hello,85
log: 4 ringo received,hello,85
```

Here, the receive timestamp (4) is greater than the send timestamp (1), preserving causality.

Holdback Queue Size: Tests with `Sleep=50` and `Jitter=20` showed that the holdback queue typically contained between 5 and 15 messages at its maximum. The queue size remained manageable because messages were processed promptly once they became safe. The size depended on factors like network latency and message rate but did not grow excessively.

4 Bonus: Vector Clocks Implementation

Sample output:

```
log: [{george,2}] from george: {sending,{hello,85}}
log: [{ringo,2},{george,2}] from ringo: {received,{hello,85}}
log: [{paul,2}] from paul: {sending,{hello,76}}
```

Differences from Lamport Clocks

- Per-process counters in each timestamp
- Concurrency detection
- Partial node sets and dynamic topologies

4.1 Implementation Details

Vector timestamps use `[Node, Count]` pairs:

```
[{george, 2}, {ringo, 3}]
```

Safety condition:

```
safe(Time, Clock) ->
    lists:all(fun({Node, T}) ->
        case lists:keyfind(Node, 1, Clock) of
            {Node, C} -> T =< C;
            false -> T =< 0
        end
    end, Time).
```

4.2 Performance Comparison

Aspect	**Lamport Clocks**	**Vector Clocks**
Timestamp Size	$O(1)$	$O(n)$
Causality Detection	Total ordering	Partial ordering + concurrency
Memory Usage	Low	Higher
Debugging Info	Basic	Detailed

Table 1: Lamport vs. Vector Clock Characteristics

5 Conclusions

This implementation successfully demonstrated the importance of logical time in distributed systems. Both Lamport and vector clocks effectively preserved causal ordering, with each approach offering distinct advantages:

- **Lamport Clocks:** Simple, efficient, and sufficient for applications requiring only total ordering. The implementation showed reliable performance with manageable holdback queue sizes.
- **Vector Clocks:** Provide superior causality tracking and debugging capabilities at the cost of increased complexity and storage overhead. Particularly valuable for systems requiring precise causal analysis.

The holdback queue mechanism proved effective in both implementations, maintaining correct delivery order while handling network asynchrony. This work highlights the practical trade-offs between simplicity and precision in distributed system timekeeping, providing valuable insights for building reliable distributed applications where event ordering matters.