Distributed Systems

P4: Synchronization

by
Asger Song Høøck Poulsen
Firas Harbo Saleh

A Distributed Systems
Project

AARHUS
UNIVERSITY

December 6, 2023

Contents

1 Introduction

In the expansive field of distributed systems, the current project embarks on an insightful journey to explore and elucidate the fundamental concepts of logical clock algorithms, specifically focusing on Lamport Timestamps and Vector Clocks. This project aims to design, implement, test, and compare these algorithms, emphasizing their ability to order events in a distributed system with accuracy and efficiency.

2 Methods and Materials

This section discusses the core concepts of Lamport Timestamps and Vector Clocks, their theoretical underpinnings, and the practical considerations in their implementation.

2.1 Lamport Timestamp

The concept of Lamport Timestamps, introduced by Leslie Lamport[1], serves as a cornerstone in the realm of distributed systems for establishing a partial ordering of events. At the heart of this algorithm lies a simple yet powerful idea: using logical clocks — counters that are not tied to physical time — to sequence events across different processes in a distributed environment.

Lamport Timestamps operate on the principle that each process in a distributed system maintains its own logical clock. When an event occurs, be it a message send or receive, or an internal event, the clock is incremented. The elegance of this system is its relative simplicity and the minimal overhead it incurs, making it a foundational approach in the study of distributed systems.

2.1.1 Happens-before

To establish synchronization among logical clocks in distributed systems, Leslie Lamport introduced the fundamental concept of happens-before, a crucial relation in Lamport Timestamps. This relation, denoted by $a \to b$, defines a chronological order between two events, stating that event a happens before event b. This relation is transitive, meaning that $\forall a, b, c$ if $a \to b$ and $b \to c$, then $a \to c$. The happens-before relation is also irreflexive, meaning that $\forall a$: $a \nrightarrow a$, and antisymmetric, meaning that $\forall a, b$: $a \neq b$, if $a \to b$ then $b \nrightarrow a$.

The happens-before relation is used to establish a partial ordering of events in a distributed system. The ordering is established by comparing the timestamps of two events. If $a \to b$, then $C(a) < C(b)$, where $C(a)$ denotes the timestamp of event a. However, it is important to note that the converse, if $C(a) < C(b)$, then $a \nrightarrow b$, is not necessarily true. This is because the happens-before relation is a partial ordering, which means it does not determine the ordering of all pairs of events. When it is the case that $a \nrightarrow b$ and $b \nrightarrow a$, the two events are said to be concurrent.

2.1.2 Lamport Timestamp Algorithm

Considering the happens-before relation, Lamport Timestamps can be defined as follows: *The timestamp of an event is the maximum of its own timestamp and the timestamps of all events that happen-before it, plus one.*

Consider the following example, where three processes, P_1, P_2 and P_3, communicate with each other depcited in figure 1. The processes run on different machines, and each process has its own logical clock. The clocks run at different rates. P_1 is incremented by 2 units, 5 units in process P_2, and 10 units in process P_3, respectively.

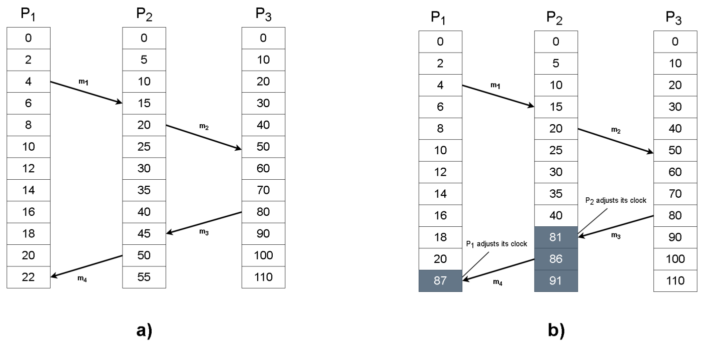


Figure 1: (a) Three processes, each with its own (logical) clock. The clocks run at different rates. (b) Lamport's algorithm synchronizes the clocks.

The first example in figure 1 shows the clocks of the three processes. Consider message m_3 which leaves P_2 at 80 and arrives at P_2 at 40. Similarly, m_4 from P_2 to P_1 leaves at 50 and arrives at 22. These values are clearly impossible in a real system, as the clocks are not synchronized. This is where Lamport Timestamps come into play. The second example shows the clocks after the

Lamport Timestamp algorithm has been applied. The algorithm follows the happens-before relation to synchronize the clocks. Since m_3 left at 80, it must arrive at 81 or later. To ensure this, each message carries the sending time according to its sender's clock.

To implement the Lamport Timestamp algorithm, each process maintains a local counter C_i which are updated according to the following steps[2]:

1. When a process P_i sends a message, it increments its counter C_i by one, and attaches the new value to the message.
2. When a process P_i receives a message, it sets its counter C_i to the maximum of its current value and the value in the received message, and increments its counter by one.
3. When a process P_i experiences an internal event, it increments its counter C_i by one.

2.1.3 Overhead

The Lamport Timestamp algorithm incurs a minimal overhead in terms of time, space, and message complexities. The time complexity is $O(1)$, as the timestamp of an event is updated in constant time. The space complexity is also $O(1)$, as each process maintains a single counter, independent of the number of processes in the system. The message complexity is $O(1)$ as well, because only a single timestamp value is attached to each message, regardless of the total number of messages or processes involved.

2.2 Vector Clocks

Vector Clocks extend the concept of Lamport Timestamps, providing a refined approach to event ordering in distributed systems. Unlike Lamport Timestamps, which offer a partial ordering, Vector Clocks allow for a comprehensive understanding of total ordering between events.

2.2.1 Principles and Operation

In Vector Clocks, each process maintains an array of counters, one for each process, to track events across the distributed system. These arrays, known as Vector Clocks, are updated based on specific rules for different event types:

1. **Internal Events**: Increment the process's own counter.
2. **Message Sending**: Increment the process's counter and attach the updated Vector Clock to the message.
3. **Message Reception**: Update each counter to the maximum of the current value and the received value, then increment the process's own counter.

2.2.2 Overhead

Vector Clocks incur a higher overhead compared to Lamport Timestamps, primarily due to the requirement for each process to maintain an array (vector) of counters—one for each process in the distributed system. This leads to a space complexity of $O(n)$ per process, where n is the number of processes in the system. The message complexity is also higher, as each message must carry the entire Vector Clock of the sending process, amounting to $O(n)$ complexity per message. However, the time complexity for updating the Vector Clocks upon events is similar to that of Lamport Timestamps, as updates are performed in a consistent and efficient manner.

2.3 Development and Environment Tools

The implementation of the Lamport Timestamp algorithm and vector clocks was done using the following tools and technologies:

- **Git**: Git is a distributed version control system for tracking changes in source code during software development. It was used to manage the source code of the Lamport Timestamp and Vector Clock implementations

- **GitHub**: GitHub is a web-based hosting service for version control using Git. It was used to host the source code of the project.

- **Visual Studio Code**: Visual Studio Code is a source-code editor. It was used to write the source code of Lamport Timestamps and Vector Clocks.

- **Pytest**: Pytest is a testing framework for Python. It was used to write and run unit tests for both the Lamport Timestamp and Vector Clock implementations.

3 Experiments, Results and Discussion

This section presents the experimental setup, results, and a detailed discussion of the findings. The experiments were designed to test the accuracy, efficiency, and scalability of Lamport Timestamps and Vector Clocks in various distributed system scenarios.

3.1 Implementations

This subsection details the implementation of Lamport Timestamps and Vector Clocks, including the design and architecture of the class implementations.

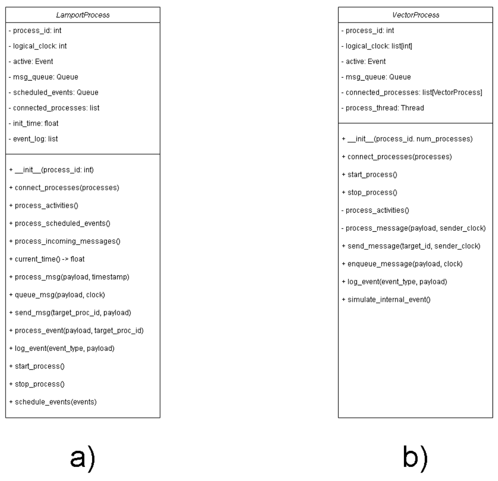

LamportProcess
- process_id: int
- logical_clock: int
- active: Event
- msg_queue: Queue
- scheduled_events: Queue
- connected_processes: list
- init_time: float
- event_log: list
+ __init__(process_id: int)
+ connect_processes(processes)
+ process_activities()
+ process_scheduled_events()
+ process_incoming_messages()
+ current_time() -> float
+ process_msg(payload, timestamp)
+ queue_msg(payload, clock)
+ send_msg(target_proc_id, payload)
+ process_event(payload, target_proc_id)
+ log_event(event_type, payload)
+ start_process()
+ stop_process()
+ schedule_events(events)
a)
VectorProcess
- process_id: int
- logical_clock: list[int]
- active: Event
- msg_queue: Queue
- connected_processes: list[VectorProcess]
- process_thread: Thread
+ __init__(process_id. num_processes)
+ connect_processes(processes)
+ start_process()
+ stop_process()
- process_activities()
- process_message(payload, sender_clock)
+ send_message(target_id, sender_clock)
+ enqueue_message(payload, clock)
+ log_event(event_type, payload)
+ simulate_internal_event()
b)

Figure 2: (a) UML class diagram of the Lamport Timestamp implementation. (b) UML class diagram of the Vector Clock implementation

3.1.1 Lamport Timestamp

The Lamport Timestamp algorithm's implementation is encapsulated within the `LamportProcess` class, representing individual processes in a distributed environment. Figure 2a depicts the class diagram of the implementation.

This class is equipped with a `logical_clock` attribute, maintaining the current state of each process's logical clock. Core functionalities of the Lamport Timestamp algorithm are methodically distributed across three key methods: `send_message` for managing outgoing messages, `process_event` for internal event handling, and `process_msg` for processing incoming messages. These methods collectively ensure the proper incrementation of the logical clock and adherence to the principles of the Lamport Timestamp algorithm. By following the steps outlined in 2.1.2, the following implementations of the main methods were developed:

```python
def process_msg(self, payload, timestamp):
    """
    Process a received message, updating the logical clock.
    Args:
        payload: The message payload.
        timestamp: The timestamp of the received message.
    """
    self.logical_clock = max(timestamp, self.logical_clock) + 1
    self.log_event("Received", payload)
```

(a) process_msg method

```python
def send_msg(self, target_proc_id, payload):
    """Send a message to a target process."""
    if 0 <= target_proc_id < len(self.connected_processes):
        self.logical_clock += 1
        self.connected_processes[target_proc_id].queue_msg(payload, self.logical_clock)
        self.log_event("Sent", payload)
    else:
        self.log_event("Error", f"Invalid target process ID: {target_proc_id}")
```

(b) send_msg method

```python
def process_event(self, payload, target_proc_id):
    """Process an event, either local or sending a message."""
    if payload == "STOP":
        self.active.set()
        return
    if target_proc_id == self.process_id:
        self.logical_clock += 1
        self.log_event("Local", payload)
    else:
        self.send_msg(target_proc_id, payload)
```

(c) process_event method

Figure 3: Key methods of the LamportProcess class

By meticulously handling the synchronization and ordering of events, the `LamportProcess` class serves as a testament to the algorithm's utility in distributed system design. Its implementation not only aligns with the theoretical underpinnings of Lamport Timestamps but also demonstrates their practical adaptability and scalability in complex distributed architectures.

3.1.2 Vector Clock

The implementation of Vector Clocks is effectively realized through the `VectorProcess` class, embodying each node in a distributed system. A visual representation of the implementation, including the class and sequence diagram, is illustrated in Figure 2b.

This class maintains a `logical_clock`, a list, that holds the logical time for each process in the system. Key operations of this implementation are encapsulated in methods such as `send_message`, `process_message`, and `simulate_internal_event`. These methods are designed to manage the synchronization and update of vector clocks during various types of events, including message sending, receiving, and internal events. By following the principles described in 2.2.1, the following implementations of the main methods were developed:

```python
def process_message(self, payload, sender_clock):
    """Update vector clock by comparing with the sender's clock"""
    for i in range(len(self.vector_clock)):
        self.vector_clock[i] = max(self.vector_clock[i], sender_clock[i])
    self.vector_clock[self.process_id] += 1
    self.log_event("Received", payload)
```

(a) process_message method

```python
def send_message(self, target_id, payload):
    if target_id < len(self.connected_processes):
        self.vector_clock[self.process_id] += 1
        self.connected_processes[target_id].enqueue_message(payload, self.vector_clock.copy())
        self.log_event("Sent", payload)
```

(b) send_message method

```python
def simulate_internal_event(self):
    """Simulate an internal event for the current process."""
    self.vector_clock[self.process_id] += 1
    self.log_event("Internal Event", "Internal event occurred")
```

(c) simulate_internal_event method

Figure 4: Key methods of the VectorProcess class

The `VectorProcess` class demonstrates a simple approach of Vector Clocks in maintaining a causal relationship among distributed processes. It highlights the ability of Vector Clocks to not only track event chronology like Lamport Timestamps but also to provide a deeper understanding of the causal dependencies among events. This enhanced capability makes Vector Clocks a more robust solution for certain complex scenarios in distributed systems.

3.2 Test Scenarios and Methodology

This subsection outlines the specific scenarios and methodologies applied to test the Lamport Timestamp and Vector Clock implementations. The aim was to rigorously evaluate their correctness with respect to the theoretical underpinnings of the algorithms.

3.2.1 Testing Framework

The testing was conducted using the Python-based Pytest framework, which provided a robust and flexible environment for creating and executing a comprehensive set of test cases. Each logical clock algorithm was subjected to a series of tests designed to prove its correctness. All test cases are included in the `test_logical_clocks.py`-file. Figure 5 illustrates some of the test cases implemented.

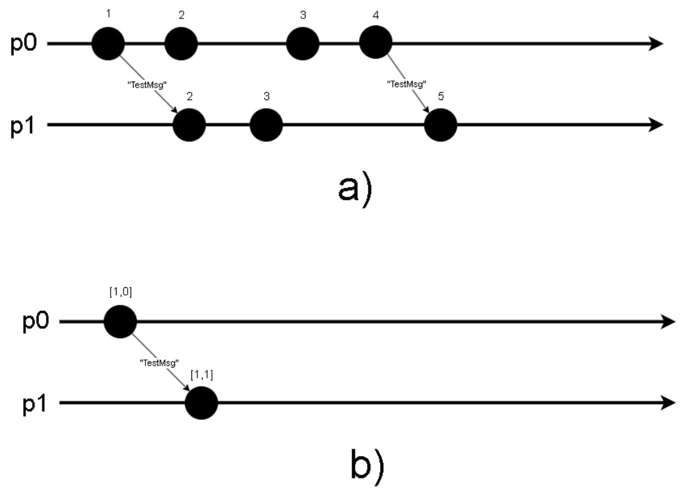


Figure 5: (a) Simple test case for the Lamport Timestamp algorithm. By the end of the test, the logical clock of `p0` and `p0` should be 4 and 5, respectively. (b) Simple test case for the Vector Clock algorithm. By the end of the test, the logical clock of `p0` and `p1` should be $[1, 0]$ and $[1, 1]$, respectively.

For both implementations, tests were designed to validate the correctness of logical clock increments during internal events, message sending, and message receiving. Key scenarios included:

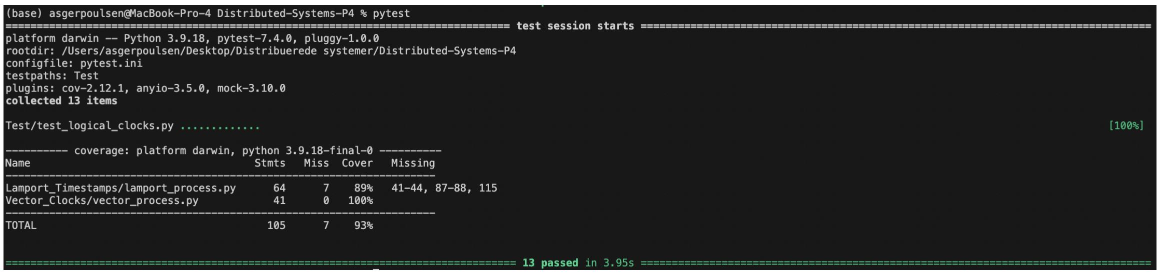
- **Local Event Handling:** Testing how the logical clock is incremented during internal events within a process.
- **Message Sending:** Assessing the behavior of the logical clock when a process sends a message to another process.
- **Message Reception:** Evaluating the logical clock's update upon receiving a message..

3.3 Results

Pytest gives a simple overview of the test results, including the number of tests run, the number of tests passed, and the number of tests failed. The test results for the Lamport Timestamp and Vector Clock implementations are shown in Figure 6.

3.4 Discussion

This discussion section delves into interpreting the test results of the Lamport Timestamp and Vector Clock implementations, considering their practical applications, efficiency, and potential areas for further research.



```
(base) asgerpoulsen@MacBook-Pro-4 Distributed-Systems-P4 % pytest
========================================================================= test session starts ==========================================================================
platform darwin -- Python 3.9.18, pytest-7.4.0, pluggy-1.0.0
rootdir: /Users/asgerpoulsen/Desktop/Distribuerede systemer/Distributed-Systems-P4
configfile: pytest.ini
testpaths: Test
plugins: cov-2.12.1, anyio-3.5.0, mock-3.10.0
collected 13 items

Test/test_logical_clocks.py .............                                                                                                                        [100%]

---------- coverage: platform darwin, python 3.9.18-final-0 ----------
Name                                     Stmts   Miss  Cover   Missing
----------------------------------------------------------------------
Lamport_Timestamps/lamport_process.py       64      7    89%   41-44, 87-88, 115
Vector_Clocks/vector_process.py             41      0   100%
----------------------------------------------------------------------
TOTAL                                      105      7    93%


========================================================================== 13 passed in 3.95s ==========================================================================
```

Figure 6: Pytest results

3.4.1 Correctness

The results in figure 6 show that all tests passed for both implementations, with an acceptable code coverage percentage, indicating that the implementations are correct. This is expected, as the implementations follow the theoretical underpinnings of the algorithms. Note that even without a 100% code coverage, the implementations can still be considered correct, as the untested code is either trivial or not part of the core functionality of the algorithms e.g. exeption handling. The correct functioning of these algorithms is crucial in distributed systems, where precise event ordering and synchronization are fundamental.

3.4.2 Limitations of the Study

This study, while informative, has certain limitations. Primarily, it relies on simulations, which may not fully replicate the complexities of real-world distributed systems, where factors like network latency and hardware differences play a significant role. Moreover, scalability tests were not conducted. The implementations worked well for a small number of processes, but their efficiency in larger systems remains untested. The study also didn't explore advanced optimizations or hybrid approaches, such as sparse vector clocks[3], that could potentially improve efficiency and applicability in specific contexts. Lastly, the focus was mainly on the correctness of the algorithms, with limited evaluation metrics. Future studies could benefit from a broader range of metrics, including fault tolerance and real-time processing capabilities, for a more comprehensive assessment of these algorithms in diverse scenarios.

4 Conclusion and perspectives

This section concludes the project by summarizing the key findings and discussing potential areas for future work.

4.1 Conclusion

This project's exploration of Lamport Timestamps and Vector Clocks in distributed systems has yielded critical insights into the mechanisms of these logical clock algorithms. The successful implementation and testing of both algorithms have demonstrated their fundamental correctness and practical applicability in a simulated environment. This study underscores the relevance of these algorithms in ensuring accurate event ordering and synchronization in distributed systems, which is crucial for maintaining consistency and coordination across different system nodes.

Theoretical Foundation: A deep understanding of the theoretical aspects of any algorithm is crucial before moving to implementation. This foundational knowledge guides effective and accurate coding practices. **Testing and Validation**: Rigorous testing is essential to validate the correctness of an implementation, especially in systems where accuracy is critical. **Adaptability and Scalability**: While the algorithms performed well in controlled test scenarios, real-world applications require adaptability to various and often unpredictable environmental factors.

4.2 Future Work

For future work, several paths can be explored:

Real-World Testing: Implementing these algorithms in real-world scenarios to assess their performance and robustness under various network conditions and system loads. **Scalability Assessment**: Testing the algorithms in larger distributed systems to understand their scalability and identify potential optimization opportunities. **Hybrid Approaches**: Investigating hybrid or optimized versions of these algorithms, such as Sparse Vector Clocks, to enhance efficiency and applicability in specific use cases. **Advanced Metrics Evaluation**: Employing a broader range of metrics, including fault tolerance and response to network partitions, to evaluate these algorithms comprehensively. This project's outcomes contribute to the ongoing discourse in distributed systems, particularly in the domain of logical clocks, and pave the way for future innovations and enhancements in this field.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, 1978.

[2] M. Raynal and M. Singhal, "Logical time: Capturing causality in distributed systems," *Computer*, 1996.

[3] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," 01 2004.

[4] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed. Maarten van Steen, 2023, ch. 5.