P4 Synchronization

Project report

A Distributed Systems Report

by Group 2:

Andreas Kaag Thomsen, 202105844

Buster Salomon Rasmussen, 202107215

Supervisors: Christian Fischer Pedersen & Christian Marius Lillelund

AARHUS UNIVERSITET

December 6, 2023

Contents

1 Introduction

One of the major problems arising from distributed systems, as compared to centralized systems, is synchronization. The objective of synchronization is to agree on 'what time it is' in a logical or physical sense, to be able to schedule or sort events in the right order. In many situations, such as load balancing, resource sharing and real-time processing in areas like factories, aircraft, space vehicles and military, synchronization is important [7].

This job is possible handled by use of physical- or logical clocks, or a hybrid of the two [3]. Physical clocks utilizes the build-in computer timer of some computing systems, and gives a global representation of time, which can be converted to UTC time. Logical time on the other hand, deals with relative time, that is timing of events relative to each other, not an absolute global standard like UTC. This is especially useful in determining the order of events by capturing causality [9].

This project will focus on the latter of the two, synchronization of events using logical time. More precisely it will delve into two methods; Lamport timestamps and Vector clocks, and the underlying mathematics.

2 Methods and tools

In this project a process has been implemented in object oriented Python. We have used Visual Studio Code as our integrated development environment (IDE) and Github for code sharing and version control. In the design phase we have used the unified modelling language (UML) to showcase the design and architecture of our implementation.

3 Theory

Due to the nature of this project, we find it relevant to introduce some important theory on logical clock synchronization.

3.1 Terminology

Throughout the report, we apply the following terminology; We define a distributed system to consist of a finite set of processes $\mathbb{P}$. A process $p_i \in \mathbb{P}, i = 0 \dots N-1$, is characterized by a finite set of events $\mathbb{E}$. An event $e_i \in \mathbb{E}, i = 0 \dots M-1$, is either an internal event, or the sending by p_i, of a message to another process p_j, or the receipt of a message sent to p_i, by another process p_j.

3.1.1 Partial and total order

When discussing logical clocks, the terms *partial ordering* and *total ordering* often come up. For this reason, we explain these terms from a mathematical point of view here. In general, we define that a relation on a nonempty set A is called a partial ordering or a partial-order relation if it is reflexive, anti-symmetric, and transitive. The partially ordered set is also called a poset and is denoted with '$(A, \preceq)$' where '$\preceq$' is the relation. Note that a partial order relation can also be irreflexive if the first property is not held and is then referred to as a *strict partial order*. For a poset A, it may be true that $\exists a, b \in A : a \neq b, a \npreceq b \vee b \npreceq a$. That is, every pair of distinct elements is not necessarily comparable. On the other hand, if it is true that $\forall a, b \in A : a \prec b \vee b \prec a$ (every pair of distinct elements comparable) we

call the relation a total ordering and the totally ordered set a *chain* [5]. To summarize, the key difference between a poset and a chain is that in the first, all elements are **not** necessarily comparable whereas in the latter all elements are comparable.

3.1.2 The happened before relation

The *happened before relation* " $\rightarrow$ ", introduced by Leslie Lamport in 1978 [4], is defined as a strict partial order on $\mathbb{E}$. The relation satisfies these conditions:

- If e_1 and e_2 are events in the same process, and e_1 comes before e_2, then $e_1 \rightarrow e_2$ (symmetric)
- If e_1 is the sending of a message by one process and e_2 is the receipt of the same message by another process, then $e_1 \rightarrow e_2$ (symmetric)
- If $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$, then $e_1 \rightarrow e_3$ (transitive)
- For any event e_i, $e_i \nrightarrow e_i$ (irreflexive).

The intuitive understanding is that the relation expresses causality between two events, that is $e_1 \rightarrow e_2$ means that e_1 can causally effect e_2. If two events are not casual, they are called *concurrent*, which is denoted by $(e_1||e_2)$. That is, if $(e_1 \nrightarrow e_2 \wedge e_2 \nrightarrow e_1)$.

3.2 Lamport's algorithm

Lamport's algorithm, introduced by Leslie Lamport in 1978 [4], enforces the partial order '$(\mathbb{E}, \preceq)$', in a strict manner. Firstly the *happened before* relation will be explained. Next logical clocks will be explained. They are the means of which the relation is obeyed. Finally it will be explained exactly how logical clocks are used to enforce the happened before relation.

3.2.1 Logical clocks in Lamport's algorithm

The following theory is primarily based on [4]. The way the happened before relation is sustained, is through the notion of logical clocks. Clocks are a way of assigning numbers to events. In the case of Lamport timestamps it is essentially a function from the set of events $\mathbb{E}$ to the set of natural numbers $\mathbb{N}$, $C : \mathbb{E} \rightarrow \mathbb{N}$, meaning that each event is defined by a natural number.
The most important property we desire the algorithm to fulfill is the *clock condition*, defined as

$$e_1 \rightarrow e_2 \Rightarrow C(e_1) < C(e_2)$$

where $e_1, e_2 \in \mathbb{E}$. That is, if event e_1 can causally affect event e_2, then it must be the case that e_1 has a lower clock value than e_2. Note that this cannot be stated as a an equivalence (iff, $\Leftrightarrow$), because Lamport timestamps do not capture concurrent events. To clarify, it might be the case for two concurrent events $e_1, e_2 \in \mathbb{E}$, that $C(e_1) < C(e_2)$, but because the events are concurrent the happened-before relation does not hold, i.e. $(e_1 \nrightarrow e_2 \wedge e_2 \nrightarrow e_1)$. This is referred to as *weak clock consistency.*

For two distinct events $e_1, e_2 \in \mathbb{E}$ it can also be the case that $C(e_1) = C(e_2)$, if they are concurrent, which makes the two events incomparable, leading to a (strict) partial ordering of events.

If we wish to extend Lamport clocks such that all events are comparable, we can extend the logical clock value with the unique identifier of the process to break ties such that a logical value is a tuple: $(C_i(e), p_i)$. Note that this extension results in a total ordering of events since all events are now comparable [4]. This

total order is defined as follows; consider two events a and b from p_i and p_j respectively. Then the total order on $\mathbb{E}$ is defined as:

$$(a \prec b) \iff (C_i(a) < C_j(b) \vee (C_i(a) = C_j(b) \wedge p_i < p_j)) \tag{1}$$

The total order in Equation 1 is **consistent with causality** meaning that $(a \rightarrow b) \Rightarrow (a \prec b)$ [2].

3.2.2 Lamport's algorithm overview

Every process $p \in \mathbb{P}$ has a variable $c = C(e)$ to represent the clock value associated with the latest event $e \in \mathbb{E}$. The algorithm can be stated simply:

- For any internal event e on p, its counter is incremented such that $c = C(e) = c + 1$
- When a process p_i sends a message m to p_j, corresponding to event e, it increments its counter such that $c_i = C(e) = c_i + 1$, and transmits this counter with the message.
- When the process p_j receives the message m with clock value c_i, it increments its own counter such that $c_j = \max(c_j + 1, c_i + 1)$

This implementation satisfies the implementation rules *IR1* and *IR2* given in the article by Lamport. Thus, it also ensures that the clock condition is satisfied [4].

3.3 Vector clocks

As mentioned, Lamport clocks has weak clock consistency. That is, we cannot say anything about the relationship between two events e_i and e_j merely by comparing their clock values $C(e_i)$ and $C(e_j)$. More importantly, given just $C(e_i)$ and $C(e_j)$ with $C(e_i) < C(e_j)$ we cannot tell if $(e_i \rightarrow e_j)$ *nor* $(e_i || e_j)$. What we would like is to say that *if* $C(e_i) < C(e_j)$ then e_i happened before e_j. The main point is finding out if one event can have affected the other - i.e. the causality of events. This can be obtained using *causal histories*. The following definition is based on [1]. In general, this works by assigning each event a unique name and then keep joining the old causal history with new events. That is, for each new event, the process creates a new unique name and the causal history consists of the union of this name and the causal history of the previous events. Then, each time a process sends a message, its causal history is sent together with the message and on receipt the received causal history is joined with the local causal history. Now, we can check causality between two events e_i and e_j simply by set inclusion:

$$e_i \rightarrow e_j \iff H_{e_i} \subset H_{e_j}$$

where H_e is the causal history of event e. This follows from the definition of causal histories since the causal history of an event will be included in the causal history of the following event [1]. Sending the entire history causes a large amount overhead and for this reason the more compact notation of *vector clocks* was developed. Here, each process P_i holds a vector V_i of length N where N is the number of nodes in the distributed system. One way of thinking of one such vector is by interpreting it as a *set of events*, containing the current event $e \in \mathbb{E}$ and its causal dependencies (i.e. the events before e) denoted by: $\{e\} \cup \{a \in \mathbb{E} : a \rightarrow e\}$ [2]. The creation of a unique event corresponds to incrementing the entry in the vector for the node where the event is created such that $V_i[i] = V_i[i] + 1$. The union of two causal histories corresponds to taking the elementwise maximum of the corresponding two vectors such that $\forall k : V_i[k] = \max(V_i[k], V_j[k])$. Checking that $e_i \rightarrow e_j$ corresponds to:

$$e_i \rightarrow e_j \iff \forall i : V_{e_i}[i] \leqslant V_{e_j}[i] \wedge \exists j : V_{e_i}[j] < V_{e_j}[j] \tag{2}$$

Or simply $e_i \rightarrow e_j \iff V_{e_i} < V_{e_j}$ for short. Note the ' $\iff$ ' implying *strong clock consistency*. If the above does *not* hold the two events are concurrent. Thus, by using vector clocks the causal dependency can be checked by just inspecting the vector clocks.

4 Results

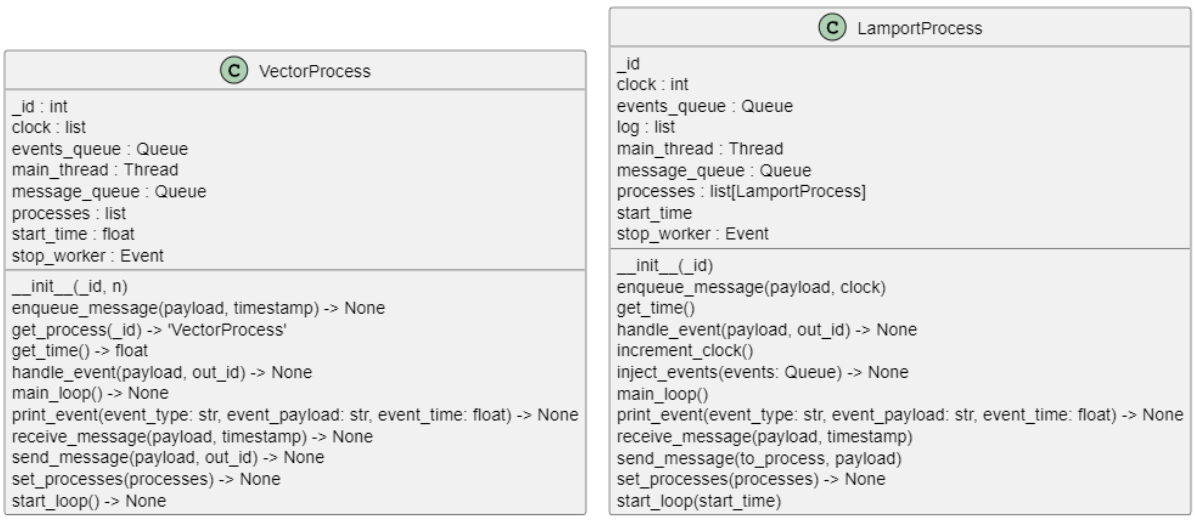
In this section we present our results of the project. In subsection 4.1 we present the architecture of the implementation using class- and sequence diagrams and in subsection 4.2 we present selected parts of the source code.

4.1 Architecture

We have chosen to implement a process as a class. In order to simulate events happening in a process, we have created a queue of tuples that holds events on the form `(to_id, payload, time)`. `to_id` is the receiver of that event, which can also be the process itself, `payload` is a string and `time` is the time when that event should be executed. Each instance of a process runs a thread that checks for both events as well as incoming messages from other processes. Below, we present the logical and procedural view of the implementation.

4.1.1 Logical view

We have implemented two different classes each using one of the two logical clocks and their respective class diagrams are shown in Figure 1.



(a) UML Class diagram for the `VectorProcess` class (b) UML Class diagram for the `LamportProcess` class

Figure 1: Class diagrams

The main functionality is within the `main_loop()` method and is presented in subsection 4.2.

4.1.2 Process view

In this view the sequence diagram in Figure 2 is presented. This diagram illustrates the main functionality in how one process P_1 processes an internal event, and how P_1 and P_2 processes send and receive events, in the Lamport implementation. The structure is the same for the vector clock implementation, since the only difference is how the clock is updated on a receive event.
As evident from the figure, all business logic runs within `main_loop`. This function first calls `handle_event`, which in turn handles the event depending on the type. This can either be an internal event or a send event. Processes periodically check their message queues, and calls `receive_message` on new messages, which in turn updates the clock, based on the specific implementation.

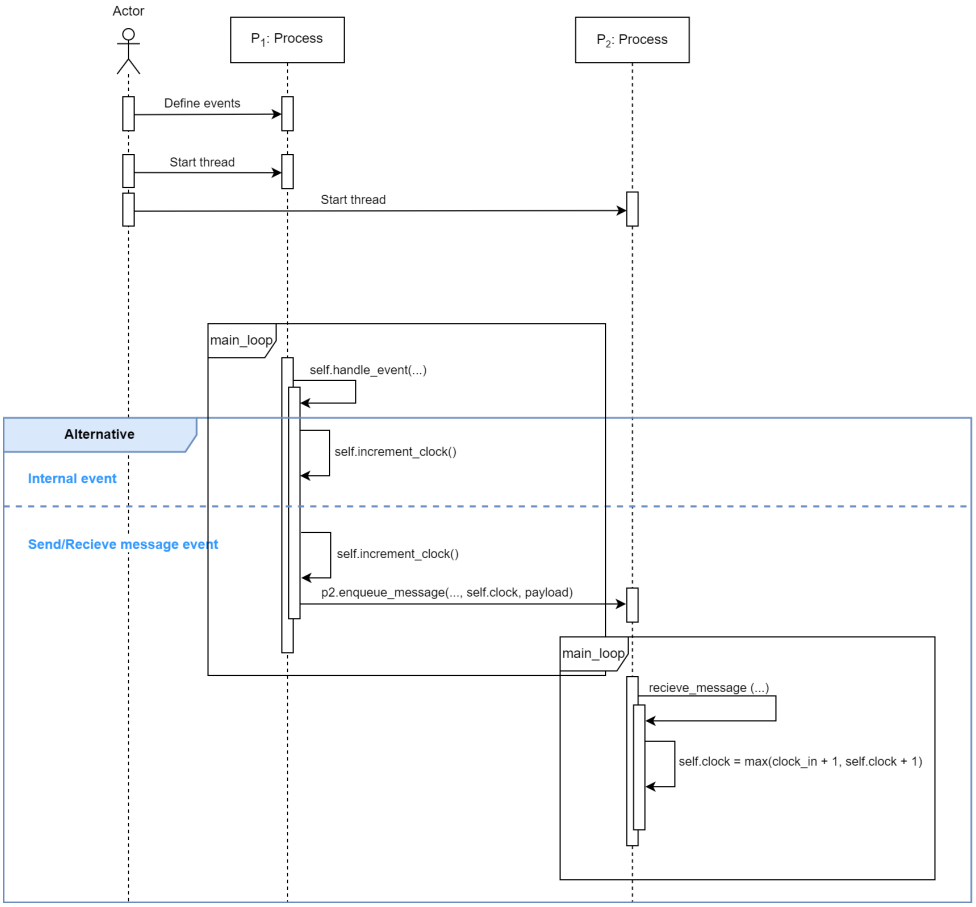


Figure 2: Sequence diagram illustrating how different types of events are handled in the Lamport implementation

4.2 Implementation

In this section we delve into the implementation details. As depicted in the sequence diagram in Figure 2, all processes have a daemon `main_loop`, which executes the business logic. The logic of this method is listed in Listing 1. This method handles events in the events queue, if the time has come to execute that

event, as it can be seen on line 7-11. After that it handles incoming messages, on line 13-20.

```
1  def main_loop(self):
2      while not self.stop_worker.is_set():
3          # Check event queue for events
4          if not self.events_queue.empty():
5              time_delta = self.get_time()
6              event_time, event_payload, out_id = self.events_queue.queue[0]
7              if time_delta >= event_time:
8                  # Remove element
9                  self.events_queue.get()
10                 # Call send message with event and timestamp
11                 self.handle_event(event_payload, out_id)
12         # Check for incomming messages
13         try:
14             payload, clock_in = self.message_queue.get(timeout=0.1)
15         # Empty
16         except Empty:
17             pass
18         # Not empty
19         else:
20             self.receive_message(payload, clock_in)
```

Listing 1: The body of the method `main_loop()` in both Lamport and Vector Clock

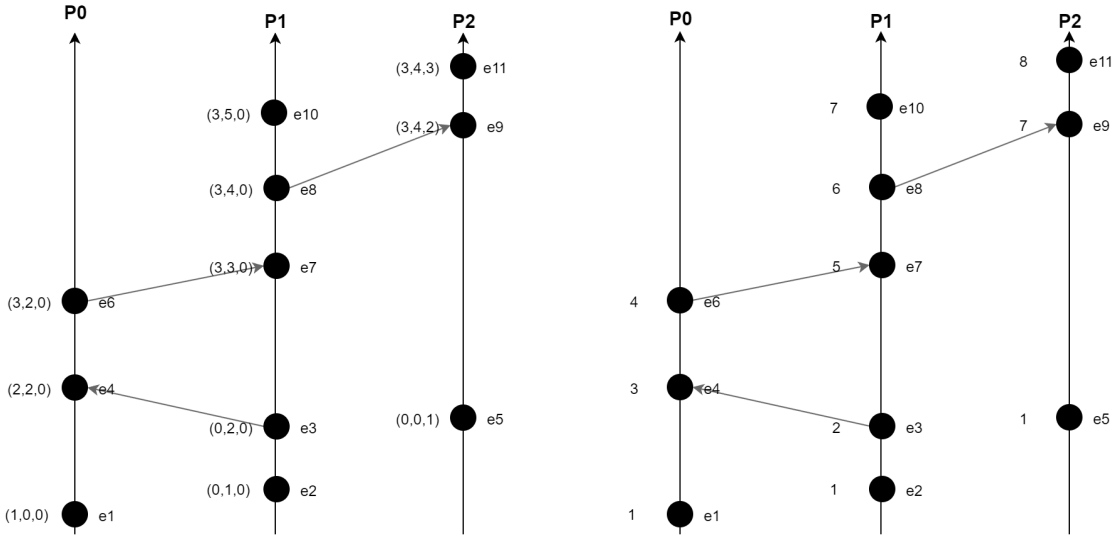
We will leave out the implementation details of `handle_event` and `receive_message`, as their logic is described sufficiently by the sequence diagram and in the theory section.

5 Experiments

In this section we elaborate on the experiments we have conducted to test our implementation.

5.1 Experiment setup

In order to test our implementation, we wish to test the scenarios depicted in the space-time diagrams in Figure 3. The scenarios are identical, but we see there is a difference in the clocks of the processes since they use the two different logical clocks.



(a) Space-time diagram using Vector clocks

(b) Space-time diagram using Lamport clocks

Figure 3: Space-time diagrams using the different algorithms

Focusing on the events `e10` and `e11`, using the (extended) Lamport clock, we would *enforce* a total ordering, but still get a weak clock consistency. Thus, we are not able to infer causality or concurrency from the clocks alone. What we are able to say is that `e11` did **not** happen before `e10` since its timestamp would then be lesser. But we cannot actually say that $e_{10} \rightarrow e_{11}$ nor $(e_{10}||e_{11})$ merely by inspecting the clock values. On the other hand, when using vector clocks we can infer the concurrency $(e_{10}||e_{11})$, by comparing their values $[3, 5, 0]$ and $[3, 4, 3]$ using Equation 2.

5.2 Experiment outputs

Injecting the events as they occur in Figure 3, and executing our test code in `test.py` we obtain the results shown in Listing 2 and Listing 3.

```
---- LAMPORT TIMESTAMPS TEST ----
LOCAL event [PROCESS_ID: 0], [CLOCK: 1], [PAYLOAD: e1], [TIME: 1.06]
LOCAL event [PROCESS_ID: 1], [CLOCK: 1], [PAYLOAD: e2], [TIME: 2.0]
SEND event [PROCESS_ID: 1], [CLOCK: 2], [PAYLOAD: e3], [TIME: 3.12]
RECEIVE event [PROCESS_ID: 0], [CLOCK: 3], [PAYLOAD: e3], [TIME: 3.14]
LOCAL event [PROCESS_ID: 2], [CLOCK: 1], [PAYLOAD: e5], [TIME: 5.12]
SEND event [PROCESS_ID: 0], [CLOCK: 4], [PAYLOAD: e6], [TIME: 6.03]
RECEIVE event [PROCESS_ID: 1], [CLOCK: 5], [PAYLOAD: e6], [TIME: 6.03]
SEND event [PROCESS_ID: 1], [CLOCK: 6], [PAYLOAD: e8], [TIME: 8.13]
RECEIVE event [PROCESS_ID: 2], [CLOCK: 7], [PAYLOAD: e8], [TIME: 8.13]
LOCAL event [PROCESS_ID: 2], [CLOCK: 8], [PAYLOAD: e11], [TIME: 9.11]
LOCAL event [PROCESS_ID: 1], [CLOCK: 7], [PAYLOAD: e10], [TIME: 11.02]
```

Listing 2: Experiment results using Lamport timestamps

```
---- VECTOR CLOCKS TEST ----
LOCAL event [PROCESS_ID: 0], [CLOCK: [1, 0, 0]], [PAYLOAD: e1], [TIME: 1.02]
LOCAL event [PROCESS_ID: 1], [CLOCK: [0, 1, 0]], [PAYLOAD: e2], [TIME: 2.04]
SEND event [PROCESS_ID: 1], [CLOCK: [0, 2, 0]], [PAYLOAD: e3], [TIME: 3.04]
RECEIVE event [PROCESS_ID: 0], [CLOCK: [2, 2, 0]], [PAYLOAD: e3], [TIME: 3.05]
LOCAL event [PROCESS_ID: 2], [CLOCK: [0, 0, 1]], [PAYLOAD: e5], [TIME: 5.11]
SEND event [PROCESS_ID: 0], [CLOCK: [3, 2, 0]], [PAYLOAD: e6], [TIME: 6.1]
RECEIVE event [PROCESS_ID: 1], [CLOCK: [3, 3, 0]], [PAYLOAD: e6], [TIME: 6.1]
SEND event [PROCESS_ID: 1], [CLOCK: [3, 4, 0]], [PAYLOAD: e8], [TIME: 8.07]
RECEIVE event [PROCESS_ID: 2], [CLOCK: [3, 4, 2]], [PAYLOAD: e8], [TIME: 8.09]
LOCAL event [PROCESS_ID: 2], [CLOCK: [3, 4, 3]], [PAYLOAD: e11], [TIME: 9.06]
LOCAL event [PROCESS_ID: 1], [CLOCK: [3, 5, 0]], [PAYLOAD: e10], [TIME: 11.06]
```

Listing 3: Experiment results using Vector clocks

We observe that the vectors are identical to those expected. Since there is no global event counter in the system, the event numbers are slightly different than those in Figure 3. For instance the `RECEIVE` event in line 4 in Listing 3 with payload `e3` corresponds to event `e4` in Figure 3a.

6 Discussion

This section delves into how logical clocks can be optimized as well as alternatives to logical clocks.

6.1 Vector clocks vs. Lamport clocks

Vector clocks and Lamport clocks are both logical clocks used to determine the order of events in a distributed system. Lamport clocks are simply integers[1] making the space requirement $O(1)$. But they are not able to capture concurrent events in the distributed system. For this reason, vector clocks exist, which *are* able to capture concurrent events. Unfortunately, the space requirement for vector clocks is in the order of nodes in the system $O(N)$ and is prohibitive for large systems [3]. Regarding message complexity, the two algorithms perform identically with $O(1)$ as they only send out one message on each `SEND` event though vector clocks uses more network bandwidth when sending the entire vector. When it comes to time complexity, the two implementations differ only in the `receive_message()` method. Here, the vector clock implementation has to loop over its vector of size N whereas the Lamport clock implementation simply takes the maximum of the two integer clock values.

6.2 Alternative algorithms

Vector clocks have clear benefits over Lamport clocks, but optimizations can be made regarding the message overhead. This can be done by observing that in between message sends to the same node, only a few entries in the vector clock are likely to change as exploited in Singhal-Kshemkalyani's differential technique [8]. Without going into too much detail, the idea is generally that a process p_i sending a message to p_j need only send a subset of the entries in its vector clock. Moreover, p_i never needs to send the vector clock entry corresponding to p_j. This technique is especially useful for large systems since only few of the processes are likely to interact resulting in updates in few entries of the vector clock [8]. Though the space complexity at each node is the same as for normal vector clocks ($O(N)$), they argue

[1] Or tuples when using the extended Lamport clock

that the traffic reduction on the communication network is more desirable, since this has limited capacity and is often a bottleneck.

Another alternative is the use of Bloom clocks as put forward by Ramabaja in 2019 [6]. This technique is applicable for highly distributed systems (i.e. large N), since the space complexity does not depend on the number of nodes in the system but rather on a set of chosen parameters, which determine the *confidence* of the clock. It does this by introducing the possibility for false positives, i.e. the possibility of concluding that $e_1 \rightarrow e_2$ even though $e_1 \nrightarrow e_2$. This is a probabilistic and space-efficient approach with the same benefits of the vector clock without the space overhead and gives a partial ordering of the events.

Event ordering can also be accomplished by using physical clocks, though there can be situations where they do not capture causality [2]. One technique, called centralized clock synchronization, uses a client-server model, where one central time server acts as a leader. Examples of such algorithms are Christian's algorithm and Berkeley algorithm. Christian's algorithm assumes that the central time server has a perfect representation of the time. When clients wish to synchronize their time, they request the server's time, and calculate the new time by including propagation delay. Berkeley's algorithm on the other hand, makes no such assumption, but works by polling every client including itself for their local time, and calculates the new time to be the average, and then sends the adjustments to the clients [7].

7 Conclusion

In this report, we have explored the concept of logical clocks and their applications in distributed systems. We have implemented two algorithms for logical clocks: Lamport clocks and vector clocks. We have compared and contrasted their advantages and limitations in terms of space, time, and message complexity, as well as their ability to capture causality and concurrency among events. We have also conducted experiments to demonstrate the behavior of the two algorithms in a simulated distributed system.

Our main findings are that vector clocks are more expressive than Lamport clocks, as they can distinguish between concurrent and causally related events. However, vector clocks also require more space and computation, as they need to store and update a vector of size N for each process, where N is the number of processes in the system. Lamport clocks, on the other hand, are simpler and more efficient, as they only use a single integer value for each process. However, when extending Lamport clocks they are also more ambiguous and conservative, as they enforce a total ordering among events that may not reflect the actual causal order.

Furthermore, we have explored alternative methods for event synchronization, including improvements of the vector clock algorithm and some physical clock methods. We conclude that there is no definitive answer to which clock algorithm is better, as it depends on the specific requirements and trade-offs of the distributed system.

8 References

[1] Carlos Baquero and Nuno Preguiça. "Why Logical Clocks Are Easy". In: *Commun. ACM* 59.4 (Mar. 2016), pp. 43–47. ISSN: 0001-0782. DOI: 10.1145/2890782. URL: https://doi.org/10.1145/2890782.

[2] Martin Kleppmann. *Distributed Systems 4.1: Logical time*. Youtube. 2020. URL: https://www.youtube.com/watch?v=x-D8iFU1d-o&t=184s. (accessed 2023-12-05).

[3] Sandeep S. Kulkarni et al. "Logical Physical Clocks". In: *Principles of Distributed Systems*. Ed. by Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro. Cham: Springer International Publishing, 2014, pp. 17–32. ISBN: 978-3-319-14472-6.

[4] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[5] *Partial and Total Ordering*. July 2021. URL: https://math.libretexts.org/@go/page/8424. (accessed 2023-11-30).

[6] Lum Ramabaja. *The Bloom Clock*. 2019. arXiv: 1905.13064 [cs.DC].

[7] Amritha Sampath and C. Tripti. "Synchronization in Distributed Systems". In: *Advances in Computing and Information Technology*. Ed. by Natarajan Meghanathan, Dhinaharan Nagamalai, and Nabendu Chaki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 417–424. ISBN: 978-3-642-31513-8.

[8] Mukesh Singhal and Ajay Kshemkalyani. "An efficient implementation of vector clocks". In: *Information Processing Letters* 43.1 (1992), pp. 47–52. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(92)90028-T. URL: https://www.sciencedirect.com/science/article/pii/002001909290028T.

[9] M van Steen and A.S. Tanenbaum. *Distributed Systems*. 4. Maarten van Steen, 2023. URL: distributed-systems.net.