Logical clocks

Causality and concurrency

Paul Kryzanowski
February 14, 2021 (Updated September 28, 2023)

> ***Goal:** Allow processes on different systems to identify causal relationships and their ordering among events, particularly among messages sent between different systems.*

Introduction

A computer's time-of-day clocks (known as *wall clocks*) rely on real-world time. They are usually set by contacting a server that has the time, often via NTP (the Network Time Protocol). These clocks have some inaccuracies due to the imperfect physical nature of the quartz crystals that make them oscillate. These inaccuracies create clock drift, leading to the likelihood that two computers will report slightly different time values if queried at the same time. To correct for drift, computers periodically re-sync their clocks via NTP. However, even NTP synchronization does not yield perfectly accurate time and the clock continues to drift between synchronizations. Because of this, we cannot definitively determine that one event occurred before another event by simply comparing the timestamps of the events. In a large environment (think of thousands of servers processing thousands of requests per second), it is also highly likely that multiple events may share the same timestamp. Moreover, time-stamping events with time-of-day values fails to account for concurrent events, which are events unrelated by cause-effect, and might incorrectly order them.

The goal of logical clocks is to identify the ordering of events. We are not interested *when* an event happens but rather if an event happened before some other event. The concept of logical clocks was created by Leslie Lamport in his paper Time, Clocks, and the Ordering of Events in a Distributed System.

Lamport clocks enable processes to assign sequence numbers ("timestamps") to events such as messages so that all cooperating processes can agree on the order of related events. There is no assumption of a central time source and thus no concept of *when* events took place.

Events are **causally** related if one event may potentially influence the outcome of another event. For instance, a sequence of events in one process are causally related: there is a clear progression from one event to another. If one process sends a message to another, the event of sending the message causally precedes the event of the other process receiving the message. If process A sends a message to process B, all events that occurred on process A before that message was sent causally precede all the events that occur on B after B received the message.

The central concept with Lamport's logical clocks is the **happened-before** relation: ***a⟶b*** represents that event *a* occurred before event *b*. This order is imposed upon consecutive events at a process and also upon a message being sent before it is received at another process. Beyond that, we can use the transitive property of the relationship to determine causality: if *a⟶b* and *b⟶c* then *a⟶c*.

If the *happened-before* property applies to a pair of events, then they are causally related. If there is no **causal** relationship between two events (e.g., they occur on different processes that do not exchange messages or have not yet exchanged messages, even indirectly), then the events are said to be **concurrent**.

Lamport Clocks

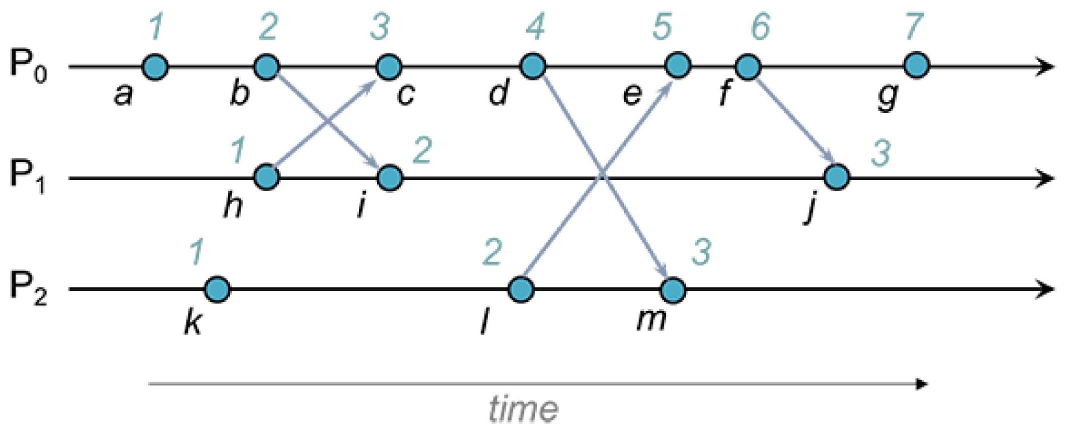
Lamport's clock algorithm relies on a per-process clock. This clock is simply a counter (e.g., a global variable) and is always incremented before each event. A timestamp is just a local sequence number.

The algorithm states that every event is timestamped (assigned a sequence number from the clock). Each message carries the timestamp of the sender's clock (sequence number). A message comprises two events: (1) at the sender, we have the event of sending the message and (2) at the receiver, we have the event of receiving the message.

When a message arrives, if the receiver's clock is less than or equal to the message's timestamp, the clock is set to the *message timestamp + 1*. This ensures that the timestamp marking the event of a received message will always be greater than the timestamp of that sent message.

An example

Each process maintains a single Lamport timestamp counter. Each event in the process is tagged with a value from this counter. The counter is incremented before the event timestamp is assigned. If a process has four events, *a, b, c, d*, the events would get Lamport timestamps of *1, 2, 3, 4*, respectively. Let's look at an example. The figure below shows a bunch of events on three processes. Some of these events represent the sending of a message, others represent the receipt of a message, while others are just local events (e.g., writing some data to a file). With these per-process incrementing assignments, we get the clock values shown in the figure.



This simple incrementing counter does not give us results that are consistent with causal events. If event *a* happened before event *b* then we expect *clock(a)* < *clock(b)*.

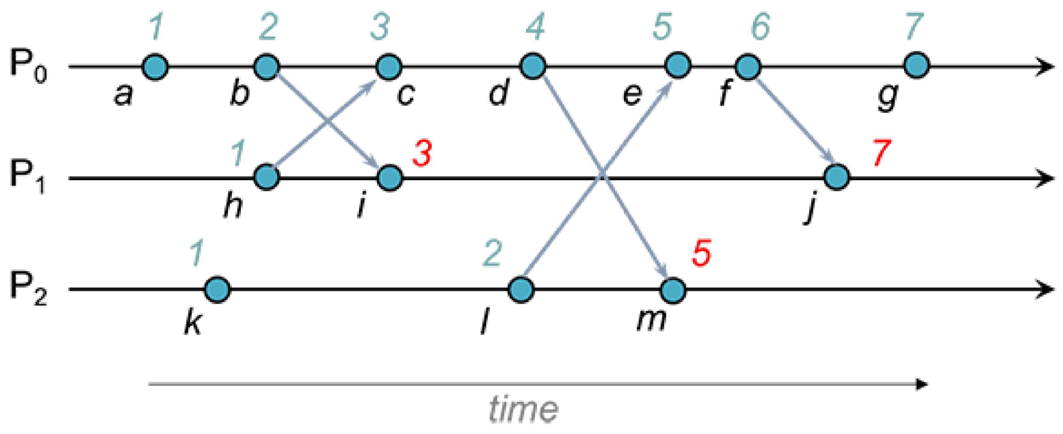
To make this work, Lamport timestamp generation adds an extra step:

If an event is the sending of a message then the timestamp of that event is sent along with the message. If an event is the receipt of a message then the the algorithm instructs the process to compare the current value of the process' clock (which was just incremented before this event) with the timestamp in the received message. If the timestamp of the received message is greater than or equal to that of the event, the event and the process' timestamp counter are both updated with the value of the timestamp in the received message plus one. This ensures that the timestamp of the received event and all further timestamps on that process will be greater than that of the timestamp of the event of sending the message as well as all previous messages on that process.

In the figure below, event *i* in process P_1 is the receipt of the message sent by event *b* in P_0. If event *i* was just a normal local event, the P_1 would assign it a timestamp of 2. However, since the received timestamp is 2, which is greater than or equal to 2, the timestamp counter is set to 2+1, or 3. Event *i* gets the timestamp of 3. This preserves the relationship *b*⟶*i*, that is, b happened before i. A local event after *i* would get a timestamp of 4 because the process P_1's counter was set to 3 when the timestamp for *i* was adjusted.

Event *c* in process P_0 is the receipt of the message sent at event *h*. Here, the timestamp of *c* does not need to be adjusted. The timestamp in the message is 1, which is less than the event timestamp of 3 that P_0 is ready to assign to *c*.

If event *j* was a local event, it would get the next higher timestamp on P_1: 4. However, it is the receipt of a message that contains a timestamp of 6, which is greater than or equal to 4, so the event gets tagged with a timestamp of 6+1 = 7.



Lamport Clock Assignment

With Lamport timestamps, we are assured that two causally-related events will have timestamps that reflect the order of events. For example, event *h* happened before event *m* in the Lamport causal sense. The chain of causal events is *h*⟶*c*, *c*⟶*d*, and *d*⟶*m*.

Since the *happened-before* relationship is transitive, we know that *h*⟶*m* (*h* happened before *m*). Lamport timestamps reflect this. The timestamp for *h* (1) is less than the timestamp for *m* (7). However, just by looking at timestamps we cannot conclude that there is a causal happened-before relation.

For instance, just because the timestamp for *k* (1) is less than the timestamp for *i* (3) does not mean that *k* happened before *i*. Those events happen to be concurrent but we cannot discern that by looking at Lamport timestamps. We will need to employ a different technique to be able to make that determination. That technique is the use of *vector clocks*.

Partial ordering vs. total ordering

One result of Lamport timestamps is that multiple events on different processes may all be tagged with the same timestamp. This defines a **partial ordering** of events. If we consider multiple processes looking at a log of all the events along with their associated Lamport timestamps, events that are causally related can all be sorted consistently. Other events across all the processes may share those same timestamps. We cannot be assured that each process will sort events in the same sequence. The sequencing of non-causal events does not affect the validity of the overall system.

In some cases, however, we want to ensure that all processes in a group can compare two different timestamps and make the same decision. That means the timestamps cannot be equivalent for two different events. Even if the events are concurrent, we would like all processes to reach the same decision (e.g., consider a decision of "who gets to go first" that chooses the smaller of two timestamps).

Wecan force each timestamp to be unique by suffixing it with a globally unique process number, such an IP or MAC address appended with a process ID or thread ID). While these new timestamps will not relate to real time ordering, each will result a unique number that can be used for consistent comparisons of timestamps among multiple processes. Making every timestamp unique allows a system to achieve **total ordering**.

A second deficiency with Lamport timestamps is that, by looking at timestamps, one cannot determine whether there is a causal relationship between two events. For example, just because event *a* has a timestamp of 5 and event *b* has a timestamp of 6, it does not imply that event *a* happened before event *b*.

Vector Clocks

A way to create timestamps that allows us to discern causal relationships is to use a vector clock. A **vector clock** is no longer a single value but rather a vector of numbers, with each element corresponding to a process.

With vector clocks, we assume that we know the number of processes in the group (we will later remove this restriction). Instead of a single number, our timestamp is now a vector of numbers, with each element corresponding to a process. Each process knows its position in the vector. For example, in the example below, the vector elements correspond to the processes (P_0, P_1, P_2).

Vector clock assignment

The rules for updating vector clocks are as follows:

1. Before affixing a vector timestamp to an event, a process increments the element of its local vector that corresponds to its position in the vector. For example, process 0 increments element 0 of its vector, process 1 increments element 1 of its vector, and so on.

2. When a process receives a message, it also first increments its element of the vector (i.e., it applies the previous rule). It then sets the vector of the *received* event to a set of values that contains the higher of two values when doing and element-by-element comparison of the original event's vector and the vector received in the message.

This new vector timestamp becomes the new per-process clock vector from which future timestamps for events on that process will be generated. Think of a vector timestamp as a set of version numbers, each representing a different author.

As with Lamport's algorithm, the element corresponding to the processor in the vector timestamp is incremented prior to attaching a timestamp to an event.

For example, in an environment of four processors, P_0, … P_3, P_1 will "own" the second element of the vector.

If a process P_0 has four sequential events, *a, b, c, d*, they would get vector timestamps of *(1,0,0), (2, 0, 0), (3, 0, 0), (4, 0, 0)*. If a process P_2 has four sequential events, *a, b, c, d*, they would get vector timestamps of *(0,0,1), (0, 0, 2), (0, 0, 3), (0, 0, 4)*. If one event on P_1 is (2, 4, 0, 1) then the next event on P_1 will be (2, 5, 0, 1).

If the event is the sending of a message, the entire vector associated with that event is sent along with the message. When the message is received by a process (which is an event that will get assigned a timestamp), the receiving process does the following:

We can illustrate this with the following pseudocode where `system` is the system's vector counter and `received` is the received vector timestamp:

```
    /* receive message with vector timestamp received */
for (i=0; i < num_elements; i++)
        if (received[i] > system[i])
                system[i] = received[i];
```

Comparing vector timestamps

We can determine if two events are concurrent or causally related by comparing their timestamps. Vectors are compared by comparing their values element by element. That is, we compare the values of P_0, then P_1, etc.

Two vector timestamps are equal if each corresponding element of one vector is the same as the other.

A vector timestamp `v` is less than a vector timestamp `w` if `v` and `w` are not equal and each element of `v` is less than or equal to the corresponding element of `w` :

```
is_smaller(int v[], w[]) {  // is v smaller than w, assuming v != w
        result = true;
        for (i=0; i < num_elements; i++)
                if (v[i] > w[i])
                        smaller = false;
        return result;
```

Two events are concurrent if one vector timestamp is neither greater than nor less than the other element when doing an element-by-element comparison.

For example, events (2, 4, 6, 8) and (3, 4, 7, 9) are *not* concurrent (that is, they are causally related) because every element of the first vector is less than or equal to the corresponding element of the second vector.

The vectors (2, 4, 6, 8) and (1, 5, 4, 9) represent concurrent events. Neither vector is less than or greater than the other. For instance, 2 > 1 (first element of the first vector is greater than the first element of the second vector) but 4 < 5 (second element of the first vector is less than the second element of the second vector).

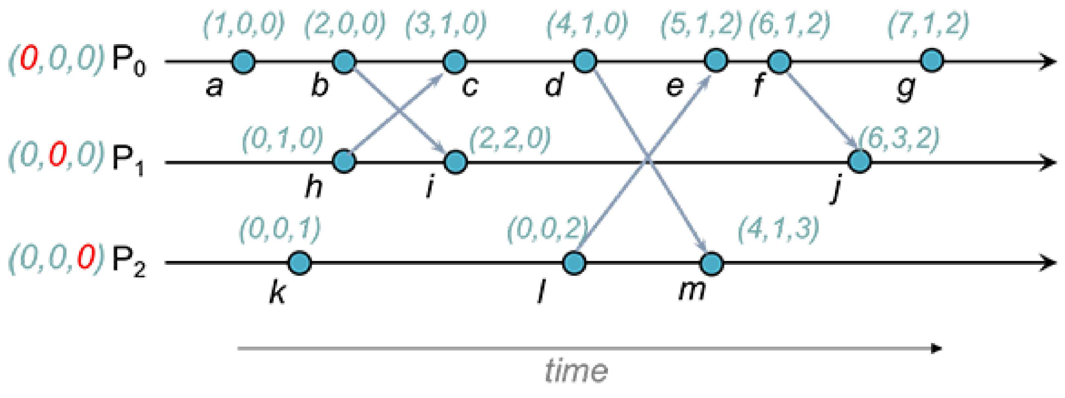
Vector clock example

The figure below shows the same set of events that we saw earlier but with vector clock assignments. Event *b* is the sending of a message to P_1. That message contains event *b*'s timestamp: (2, 0, 0). Event *i* is the receipt of that message. If *i* was a local event, it would get the timestamp (0, 2, 0). Since it is the receipt of a message, we do an element-by-element comparison of values in the received timestamp and the local timestamp, picking the highest of each pair of numbers:

Compare (2, 0, 0) with the received timestamp of (0, 2, 0).

- First element: 2 vs. 0: 2 is greater and wins.
- Second element: 0 vs. 2: 2 is greater and wins.
- Third element: 0 vs. 0: tie, so 0 wins.

The resulting vector is hence (2, 2, 0) and is assigned to event *i* as well as to the system clock. The next local event on P_1 would be tagged with (2, 2+1, 0), or (2, 3, 0).



Vector Clock Assignment

To determine if two events, V and W, are **concurrent**, do an element-by-element comparison of the corresponding timestamps. If each element of timestamp V is less than or equal to the corresponding element of timestamp W then V causally precedes W and the events are not concurrent. If each element of timestamp V is greater than or equal to the corresponding element of timestamp W then W causally precedes V and the events are not concurrent. If, on the other hand, neither of those conditions apply and some elements in V are greater than while others are less than the corresponding element in W then the events are concurrent. We can summarize it with the pseudocode:

```
isconcurrent(int v[], int w[])
{
        bool greater=false, less=false;

        for (i=0; i < num_elements; i++)
                if (v[i] > w[i])
                        greater = true;
                else (v[i] < w[i])
                        less = true;
        if (greater && less)
                return true;    /* the vectors are concurrent */
        else
                return false;   /* the vectors are not concurrent */
}
```

In the figure above, the timestamp for *e* is less than the timestamp for *j* because each element in *e* is less than or equal to the corresponding element in *j*. That is, $5 \le 6$, $1 \le 3$, and $2 \le 2$. The events are causally related and $e \longrightarrow j$

Events *f* and *m*, on the other hand, are concurrent. When we compare the first element of *f* versus *m*, we see that $f > m$ $(6 > 4)$. When we compare the second element, we see that $f = m$ $(1 = 1)$. Finally, when we compare the third element, we see that $f < m$ $(2 < 3)$. Because of this, we cannot say that the vector for *f* is either less than or greater than the vector for *m*.

Generalizing the timestamp

Recall that we had to assume that we knew the number of processes in the group so that we could create a vector of the proper size. This is not always the case in real implementations. Moreover, all processes may not be involved in communication, resulting in an unnecessarily large vector.

We can replace the vector with a set of tuples, each of which represents a process ID and its counter:

$(\{ P_0, 6 \}, \{ P_1, 3 \}, \{ P_2, 2 \})$

When a process sends a vector, it sends the entire set of tuples that it has. When it receives a vector and performs a comparison, it compares each related pair. For instance, the value of P_0 will be compared against a tuple containing P_0 in the received vector. If any process IDs are missing in one of the sets, they are implicitly given a value of 0 for comparison. The resulting vector contains the superset of all tuples.

For example, if a process has a system vector clock of:

$(\{ P_0, 6 \}, \{ P_1, 3 \}, \{ P_2, 2 \})$

and receives a value of

$(\{ P_1, 1 \}, \{ P_2, 5 \}, \{ P_3, 8 \})$

The resulting vector will be the set of all process IDs and their largest values:

$(\{ P_0, 6 \}, \{ P_1, 3 \}, \{ P_2, 5 \}, \{ P_3, 8 \})$

Last modified September 28, 2023.