Assigning Lamport & Vector Timestamps

By Paul Krzyzanowski
September 29, 2017

Introduction

This is a brief example of assigning Lamport Timestamps and Vector Timestamps.

Lamport Clocks

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 messasge, 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.

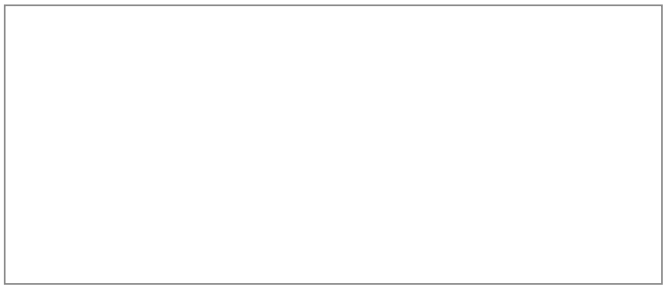
Clock Assignment

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 has 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 you to compare the current value of the process' timestamp counter (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 $b \rightarrow i$, 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 Assignments

With Lamport timestamps, we're 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 $h \rightarrow c$, $c \rightarrow d$, and $d \rightarrow m$. Since the *happened-before* relationship is transitive, we know that $h \rightarrow m$ (*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, 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 need need to employ a different technique to be able to make that determination. That technique is the use of *vector timestamps*.

Vector Clocks

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).

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. 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 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:

1. increment the counter for the process' position in the vector, just as it would prior to timstamping any local event.
2. Peform an element-by-element comparison of the received vector with the process' timestamp vector. Set the process' timestamp vector to the higher of the values:

```
for (i=0; i < num_elements; i++)
        if (received[i] > system[i])
                system[i] = received[i];
```

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 Assignments

To determine if two events 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*$rarrj

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 $f = j$ ($1 = 1$). Finally, when we compare the third element, we see that $f < j$ ($2 < 3$). Because of this, we cannot say that the vector for *f* is either less than or greater than the vector for *j*.

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:

$$(\{ P_0, 6 \}, \{ P_1, 3 \}, \{ P_2, 2 \}, \{ P_3, 8 \})$$
