Incorporating the Concept of Priority into Lamport Timestamps to Prevent Starvation in Systems That Use Timestamps for Concurrency Control

Gianluca Rombolà and Loredana Vigliano

April 1, 2020

Incorporating the concept of priority into Lamport timestamps

to prevent starvation in systems that use timestamps for concurrency control

Gianluca Rombolà
University of Rome Tor Vergata
rombola.gianluca@gmail.com

Loredana Vigliano
University of Rome Tor Vergata
vigliano@mat.uniroma2.it

ABSTRACT

Lamport timestamps are an elementary tool that can be used to maintain system-wide temporal consistency in a distributed system. By making all processes involved able to agree on the order of any two or more events (although not necessarily on their causal relation) they can be used as building block for many more complex algorithms intended for distributed systems.

Without excluding other applications of such timestamps, we are interested in how these can be used for concurrency control in transactional databases through timestamp ordering algorithms, especially with less conservative algorithms that while being usually more efficient, are also prone to starvation.

In this paper we propose an extension to Lamport timestamps that can work with any existing algorithm, by taking into account priority in order to prevent starvation: with priority we intend a dynamic property of a process that depends by its transaction failure rate, so that higher priority is symptom of more transaction rejections.

KEYWORDS

Lamport timestamps, concurrency control, timestamp ordering, starvation, priority

1 Introduction to Lamport timestamps

Lamport timestamps are logical sequence numbers used to determine event order in a distributed system. Lamport builds on concept of "happens before", such that when writing A→B we can tell that A has happened before B.

Each process involved has its own logical clock which will increase at least when sending and receiving messages to other processes, but at the same time when receiving timestamped messages it will compare its local timestamp with the one received: should the received one be higher in value, adapt itself to use this new timestamp. Regardless it will then increase its timestamp as it's supposed to do when receiving messages: doing so will ensure that for each message exchanged in the system, it's sending timestamp will always be lower than the receiving one, which means SEND→RECEIVE (after all a message can't be received before it's sent).

In addition to the above, in order to avoid ambiguities, to each process is also given a unique identifier which belongs to a totally ordered set of values. This identifier, when incorporated in a timestamp, will then make it possible to order events even when their sequence numbers are the same and would not yield a distinct answer. Let's then formalize the above concept to set a basis for this paper.

Let's consider a distributed system with N processes, let P_i be a process in such system, and let's denote with M_{ij} a message from process P_i to process P_j. When discussing timestamps, $TS(P_i)$ will be P_i local timestamp and $TS(M_{ij})$ will be P_i's timestamp when it sent M_{ij} (which is the timestamp carried by M_{ij}).

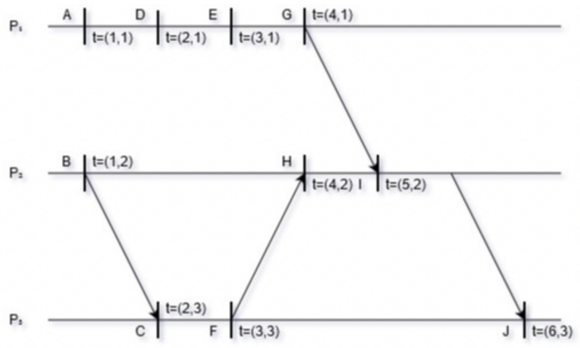

Figure 1: **We can see for each process its timestamp when sending or receiving a message, and how they make a total ordering of events.**

Timestamps issued by P_i will be pairs $(t_i, id(P_i))$ where t_i is a sequence number and $id(P_i)$ is P_i's unique identifier. We define comparison between timestamps $T_{a,i}$ = (t_a,id(P_i)) and $T_{b,j}$ = (t_b,id(P_j)) as follows:

$$T_{a,i} > T_{b,j} \Leftrightarrow t_a > t_b \vee \left(t_a = t_b \wedge id(P_i) > id(P_j)\right) \qquad (1)$$

We also define a + operator for timestamps, so that given a timestamp T_i = (t, id(P_i)) and an integer n, (T_i+n)= (t+n, id(P_i)).

So whenever P_i wants to send a message M_{ij}, it will increase it's timestamp by one TS(P_i)=TS(P_i)+1, then send it with TS(M_{ij}) = TS(P_i).

Whenever P_i receives a message M_{ji}, it will compare and pick the maximum between TS(M_{ji}) and TS(P_i) as new timestamp. It will then increase its timestamp by one as it did when sending: TS(P_i) = max(TS(P_i), TS(M_{ji})) + 1.

2 Possibility of starvation when using timestamps for concurrency control

Lamport timestamps as defined above bring an inherent bias, which is unique identifiers and their static nature. Timestamps issued with lower unique identifiers will more often result to be lower than others as a result of this: although this will be a rare issue in most systems (due to the rapid increase nature of sequence numbers in such systems), it can't be totally excluded: rather, as we will see, increasing the odds of such occurrence can be beneficial with the right approach.

If we then consider using timestamps as main source of concurrency control in a transactional environment, other problems might arise which can lead to a process having its transactions failing (of course assuming our systems allows for a transaction to fail), such as delayed messages due to external problems (which might as well be expected due to network conditions).

We are going to consider only systems were transactions are allowed to fail: in such systems, it is reasonable to take into account starvation due to a process timestamps being constantly discarded for being late.

3 Introducing priority to timestamps

While there are timestamp ordering algorithms more conservative that would prevent this problem (by simply removing the possibility for transactions to fail for example), we want to expand the concept of timestamp to include information about its issuers situation, to prevent starvation without resorting to conservative algorithms.

We do this by introducing the concept of priority to Lamport timestamps, so that each timestamp issued by P_i will be in the form $T_{a,i}$(P_i) (t_a,p_i,id(P_i)) where p_i is a priority value (integer) that is integrated into the timestamp. We then redefine our comparison operator as follows:

$$T_{a,i} > T_{b,j} \Leftrightarrow t_a > t_b \vee \left(t_a = t_b \wedge p_i > p_j\right) \vee \left(t_a = t_b \wedge p_i = p_j \wedge id(P_i) > id(P_j)\right) \qquad (2)$$

This way priority is used preferentially instead of unique identifiers when ambiguity occurs between sequence numbers in timestamps. Each process is then allowed to increase its priority value whenever it sees its transaction rejected. Also, each process will retain its priority value as it does for its unique identifier: as in the standard definition of Lamport timestamp, when receiving a message only the t component (sequence number) will be adopted if higher.

First of all we observe the set of all timestamps allowed in this system is still totally ordered, as it still inherits this property from the set of unique identifiers and nothing has changed in this regard. Also it has to be noted that this variant of timestamp behaves differently only when both t_a = t_b and p_i != p_j: in all other situations it behaves exactly the same. What this means is that the effectiveness of introducing priority this way is dependant by how often timestamp collisions of this kind happen: if having two timestamps with the same sequence number for different events is common (in which case Lamport timestamps would more often show their bias towards higher ordered unique identifiers), then priority will be meaningful.

Let's assume collisions do happen often enough to make the priority value we introduced to timestamps significant. For each process P_i, the value of p_i can increase but never decrease. Eventually each process' priority value, compared to others, will reflect that process tendency to failure, and will directly counteract to that by increasing its odds of success on each interaction that results in a collision. If collisions never happen, priority won't matter and the whole system will keep working as it would if priority wasn't introduced, which while not optimal for our purposes, doesn't bring any quantifiable performance deterioration .

3.2 Increasing timestamp collisions

In our introduction when describing Lamport timestamps we were using them to order messages exchanged in a distributed system, which is the way they are usually defined. We are not forced to follow this kind of usage though: for our purpose, we can reduce our timestamps granularity in order to have each new timestamp issued be more closely related to a transaction.

By definition, Lamport timestamps allow us to order messages exchanged in a distributed system by identifying for each message a start event (being sent) and an end event (being received) which are also causally related. Instead we want to shift our focus on transactions[4] rather than single messages, therefore we identify a start event and an end event in a transaction's own beginning and end (whether being executed or rejected): both these events are verified locally by the process issuing the transaction, as is their causal relationship, differently for what happens with sending/receiving messages.

We want then to issue a new timestamp whenever a new transaction is started, rather than whenever a message is sent. This means we lose our ability to order single messages, but we are still able to look at two transactions and pick which one happened before the other: in most timestamp ordering algorithms, transaction timestamps are the only ones used in actual decision making after all.

To formalize, we say that a new timestamp is issued whenever a process increases its local timestamp's sequence number, so that if a process receives a message with a higher timestamp and adopts it without increasing it, we say no new timestamp has been issued. In other words issuing a new timestamp could eventually (but will not necessarily) contribute to the systemic growth of timestamps across all processes. Since some timestamps might be discarded by the system (they could be lagging behind compared to other processes and discarded on reception for example), we can safely say that at any time the highest timestamp will be at most equal to the amount of timestamps issued system-wide. To further prove this, we can consider a system with only one process: in this scenario we have that no timestamps will be discarded, and each timestamp issued will increase our process' local timestamp's sequence number by one, so that after n issues, our timestamp's sequence number will be n. if we add more processes and across those we issue again m timestamps, we can expect at best to have none discarded and achieve the same results, but we can't expect to have a timestamp with its sequence number >n anywhere in the system.

Since the amount of timestamps issued across the whole system is an upper bound for sequence number, if you can reduce that quantity per amount of transactions attempted, we can expect to have an increase in the frequency of collisions. In our general definition we issued a new timestamp whenever we sent or received a message, therefore our upper bound was twice the amount of messages sent across the system. Assuming that each transaction involves and exchange of multiple messages, we can change our definition to issue a new timestamp only when a transaction is started: having the amount of transactions attempted system-wide as upper bound for highest timestamp will contribute to increasing the amount of collisions.

The question now is: can we do this while still being able to order events and allow every process to do so and make decisions autonomously in a consistent way? Well, assuming we don't care about being able to order individual messages, we can safely apply this approach and retain our ability to execute transactions correctly.

As we briefly noted earlier, most timestamp ordering algorithms only care about transaction timestamps, while making no use of any timestamp that isn't related to a transaction: as long as each transaction has got its unique timestamp and those can be correctly ordered, it has no effect on the execution of timestamp ordering algorithms.

3.3 Priority flagging

As we have seen, having a priority value built into timestamps will reduce starvation and will help balance out performance across the system.

We wonder if we can introduce a stronger concept of priority, akin to a priority flag, to allow a process to "make a reservation" to guarantee itself success, rather than simply increasing its chances. Is it possible to implement such solution? Is it practical?

First of all we consider a simple priority flag. Considering our definition of timestamp with priority value from before $TS(P_i) = (t_i,p_i,id(P_i))$, could we make it so that p_i is just a true/false flag that takes precedence over sequence number? For all practical purposes we would then have:

$$\begin{aligned} T_{a,i} > T_{b,j} \Leftrightarrow & (p_i = true \wedge p_j = false) \vee \\ & (p_i = p_j \wedge t_a > t_b) \vee \\ & (p_i = p_j \wedge t_a = t_b \wedge id(P_i) > id(P_j)) \end{aligned} \quad (3)$$

Therefore having p_i = true would give precedence over all timestamps, bar those that have their priority set to true as well. This solution won't work as it breaks the requirement that timestamps can only increase and not decrease. We can show this by considering two timestamps issued by P_i, T_1=(t_1, true, id(P_i)) and T_2=(t_2, false, id(P_i)), such that T_2 is issued after T_1 and as such t_2>t_1. It's easy to see that with the comparison operator we just defined, we have T_1>T_2, even though they were issued by the same process in different order.

To solve this problem we can extend p_i to use integers that can never decrease. We end up with something very similar to

(2), but with priority having precedence over sequence numbers:

$$T_{a,i} > T_{b,j} \Leftrightarrow p_i > p_j \lor (p_i = p_j \land t_a > t_b) \lor (p_i = p_j \land t_a = t_b \land id(P_i) > id(P_j)) \quad (4)$$

This can correctly work as a priority flagging implementation. We assume that all process have the same starting priority value, and in a balance situations end up having the same priority value as well. Whenever a process increase its priority value, exactly as it would happen were it simply a flag, it will gain priority above all other processes causing its next request (or transaction assuming multiple requests will share that transaction timestamp) to succeed.

While this can help a starving process, it has its fair share of cons. Increasing priority for a process will mean for all other processes to fail their current/next request (and this is by design, since it's to be expected with a priority flagging approach), which in response will increase their priority values, leading to a self stabilizing outcome: in the end it's an expensive price to pay to reduce starvation.

An approach like this might be seen as viable in particular circumstances, but we feel it damages the overall systemic performance to be seriously taken in consideration, especially considering we proposed an alternative that brings no side effects while directly reducing starvation.

4 **Conclusion**

We have seen how introducing priority to timestamps can provide a layer to help reduce starvation without adding any real overhead, and without having to adapt any algorithm to it. Although its real efficiency in the way we proposed it is reliant on collision rate, we think that having no side effects compared to priority-less timestamps and requiring minimal additional processing, such approach can only be beneficial in distributed systems.

Some preliminary tests on a stripped down implementation seem to confirm what we proposed in theory in this paper, showing statistically better results in scenarios where transactions failed more often. Although trying to prevent failure is generally better than fix it once happened, we feel that having a lightweight system in place to take care of it should it happen can't do harm and should be taken in consideration.

We focused our discussion on transactional distributed environments, but there's no reason to infer it wouldn't bring benefits in a more generic scope, as long as request failure is an option.

REFERENCES

(1) Leslie Lamport (1978), "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM 21,7 558-565

(2) Philip A. Bernstein and Nathan Goodman (1981), "Concurrency Control in Distributed Database Systems", Computing Surveys 13,2 185-221

(3) R. H. Thomas, "A solution to the concurrency control problem for multiple copy databases", Proc. 1978 COMP- CON Conf. (IEEE), New York.

(4) Daniel J. Rosenkrantz, Richard E. Stearns and Philip M. Lewis (1978), System level concurrency control for distributed database systems, ACM Transactions on Database Systems (TODS), v.3 n.2, p.178-198