Fair timestamp allocation in distributed systems

by SAID K. RAHIMI
Honeywell Corporate Computer Sciences Center
Bloomington, Minnesota
and
WILLIAM R. FRANTA
University of Minnesota
Minneapolis, Minnesota

ABSTRACT

Many researchers have addressed the problem of uniquely identifying updates in a distributed database system in the literature.[1,5,6,7,11] Primitive identification schemes that generate globally unique update IDs have also been suggested. These IDs are usually used as priority among updates as well. When used as such, these schemes do not distribute priority evenly across the nodes. This paper presents a numbering scheme that generates unique update IDs and, if used as a priority scheme, is fair.

1. INTRODUCTION

Many authors have addressed the problem of global identification of updates in a distributed database system.[1,5,6,7,10,11] To solve the problem, these researchers have suggested primitive ID generation schemes that globally identify all updates. Almost all these suggestions assume an ID to be a combination of two parameters: a local physical-clock parameter to provide for local identification and node numbers to provide for global identification. Thomas' algorithm for concurrent update problem of distributed database systems[11] assumes that an update ID number is a combination of a node number and readings of a physical clock at that node at the time of update generation. (Our model of a distributed system consists of a set of cooperating nodes connected by a communication facility.) Physical clocks, kept at every node of the system, tend to require resynchronization periodically. If physical clocks are skewed with respect to one another or run at different rates, certain anomalies may occur.[11] To solve the synchronization problem, Lamport[5] has suggested a rather expensive mechanism to resynchronize drifted clocks.

Away from physical-clock problems, these schemes are able to generate globally unique identification numbers for updates. The ID numbers generated are also used as priority numbers among updates.[7, 11] A priority scheme as such does not distribute priority evenly across the nodes. The reason is the fixed node number assignment that biases the priority among updates from different nodes.

Section 2 explains the update numbering schemes and their problems. To solve some of the problems, Section 3 presents the MOD numbering scheme. This scheme is solely based on the use of logical clocks and therefore does not have the problems associated with physical clocks. If ID numbers generated by this scheme are used as priority, one can be sure that this priority scheme is fair. The MOD numbering scheme achieves fairness by dynamically changing the node numbers. The problem associated with varying node numbers and a solution to this problem are also presented.

2. UPDATE NUMBERING SCHEMES/PROBLEMS

An update ID number is generated and assigned to an update by the initiating node at the time of update generation. It is assumed that each node has a logical clock (instead of a physical clock in similar schemes). A logical clock at a node simply counts the number of updates generated at that node. This means that a logical clock at a node is incremented by 1 for every update generated at that node. Another update at a node cannot be generated before the clock at that node is incremented (it is assumed that all logical clocks are set to 0 at the system initiation time). Using the logical-clock readings (LCR) at every node, therefore, solves the problem of locally identifying the updates and orders the updates by their generation. This, on the other hand, does not provide for global identification of updates, because LCRs at different nodes may be the same.

To ensure a global identification, node numbers are used as the second part of update IDs. It is assumed that the N nodes of the system are uniquely numbered 0 to $N-1$. If NN is node number, the tuple (LCR, NN) is a unique ID throughout the system. Two IDs, $ID_i = (LCR_i, NN_i)$ and $ID_j = (LCR_j NN_j)$ are said to be different ($ID_i \neq ID_j$) if and only if $LCR_i \neq LCR_j$ or $NN_i \neq NN_j$.

It is easy to show that for any two different updates i and j with $ID_i = (LCR_i, NN_i)$ and $ID_j = (LCR_j, NN_j)$, $ID_i \neq ID_j$. To see this, suppose updates i and j are generated at the same node; i.e., $NN_i = NN_j$. According to the above discussion, LCR has to be incremented after it is read for one update, and hence $LCR_i \neq LCR_j$. If the updates are from two different nodes, then $NN_i \neq NN_j$, which implies that $ID_i \neq ID_j$.

ID numbers are used in two different ways: for identification and for priority purposes.

The uniqueness property of update IDs, generated this way, gives us confidence in using these tuples as identification of updates. When used as priority, however, this scheme raises some questions. Note that priority here is concerned with ordering conflicting updates and does not have anything to do with user-defined or external priority. For two updates i and j it is usual to say

update i is of equal priority to update j if $ID_i \doteq ID_j$,
update i is of higher priority than update j if $ID_i < ID_j$,
and
update i is of lower priority than update j if $ID_i > ID_j$.

Since there are two different elements (LCR and NN) constituting each update ID, there are two possible ways of defining relations $=$, $>$, and $<$ for two updates i and j:
First,

$ID_i = (LCR_i, NN_i)$
and
$ID_j = (LCR_j, NN_j)$

which means that

$ID_i = ID_j$ if and only if $LCR_i = LCR_j$ *and* $NN_i = NN_j$,
$ID_i > ID_j$ if and only if ($LCR_i > LCR_j$) *or* ($LCR_i = LCR_j$ *and* $NN_i > NN_j$)
$ID_i < ID_j$ if and only if ($LCR_i < LCR_j$) *or* ($LCR_i = LCR_j$ *and* $NN_i < NN_j$),

as used in Rosenkrantz et al.,[9] Thomas,[11] and Traiger et al.[12]

A priority scheme is said to be fair if it distributes priority evenly among the updates from different nodes. A scheme that gives high priority to updates from one node all the time is not fair.

As far as priority is concerned, the scheme given above is fair if different nodes are generating updates at a close rate or if LCRs are not skewed. To see this, suppose that a node is generating updates at a much higher rate than the other nodes. Soon the LCR at this node becomes much greater than LCRs at the other nodes. Therefore, updates generated at this node get the lowest priority among the updates generated in the system. Some authors have suggested means of controlling this situation by proposing synchronizing LCRs,[5,12] which tends to be expensive.

Second,

$$ID_i = (NN_i, LCR_i) \text{ and } ID_j = (NN_j, LCR_j)$$

which means that

$ID_i = ID_j$ if and only if $NN_i = NN_j$ *and* $LCR_i = LCR_j$,
$ID_i > ID_j$ if and only if $(NN_i > NN_j)$ *or* $(NN_i = NN_j$ *and* $LCR_i > LCR_j)$,
$ID_i < ID_j$ if and only if $(NN_i < NN_j)$ *or* $(NN_i = NN_j$ *and* $LCR_i < LCR_j)$.

This scheme solves the problem of skewed clocks but has another potential drawback. Since in this scheme dominance is given to node number NN, all updates generated from the node numbered $N-1$ have lower priority than updates from the node numbered $N-2$, updates generated at Node $N-2$ have lower priority than updates from Node $N-1, \ldots,$ and updates from Node 1 have lower priority than updates from Node 0. According to the above definition, this scheme is not fair either. To solve the fairness problem of this scheme, we suggest the MOD numbering scheme.

THE MOD NUMBERING SCHEME

As before, the MOD numbering scheme assumes that the nodes of the system are numbered 0 to $N-1$ at system initiation time. Since the problems mentioned above stem from fixed node numbers, the MOD scheme suggests that the node numbers be changed periodically and dynamically, as follows:

$$\text{New NN} = (\text{old NN} + 1) \text{ MOD } N$$

which means that Node 0 becomes 1 and Node 1 becomes 2, ..., and node number $N-1$ becomes 0. Changing the node numbers this way solves the problem of having a biased priority scheme but creates the problem of having two or more updates with the same ID numbers. For example, assume that the LCR at Node 2 is 4 and the LCR and Node 3 is 5. This means that Node 3 has already generated an update numbered (3,4). Now assume that Node 2 changes its node number to 3. The very next update generated at this node will also be numbered (3,4).

This problem can be solved by using a node sequence number, SN, as a third part of the update ID numbers. Using SN concatenated with NN, update ID numbers become

$$ID = (SN!NN, LCR)$$

where ! denotes concatenation.

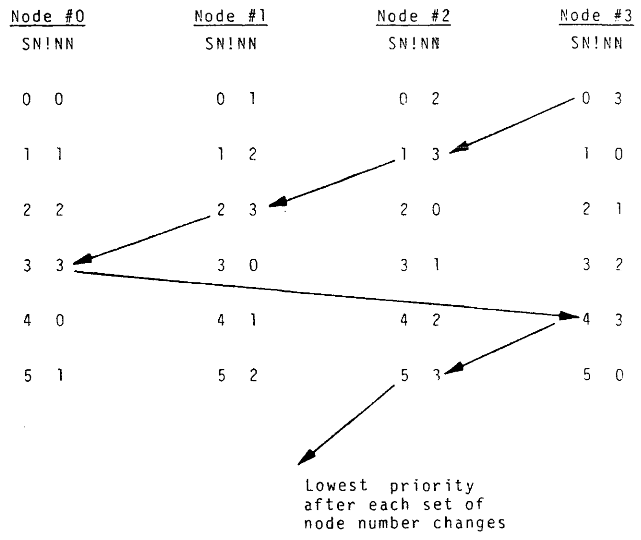

Figure 1—SN!NN for a 4-node system

SN is set originally to 0 at each node and is incremented each time the node changes its node number.

Figure 1 shows SN!NN for a system of four nodes for the first five node number changes. The dominant factor in this scheme is SN!NN; i.e., for two updates i and j with $ID_i = (SN_i!NN_i, LCR_i)$ and $ID_j = (SN_j!NN_j, LCR_j)$,

$ID_i = ID_j$ if and only if $SN_i!NN_i = SN_j!NN_j$ *and* $LCR_i = LCR_j$,
$ID_i > ID_j$ if and only if $(SN_i!NN_i > SN_j!NN_j)$
or
$(SN_i!NN_i = SN_j!NN_j$ and $LCR_i > LCR_j)$,
$ID_i < ID_j$ if and only if $(SN_i!NN_i < SN_j!NN_j)$
or
$(SN_i!NN_i = SN_j!NN_j$ *and* $LCR_i < LCR_j)$.

To show that IDs generated this way are unique, it is sufficient to show that SN!NN for any given node is unique over the system. To do this, we have to show that for any two nodes i and j at any time either $SN_i \neq SN_j$ or $NN_i \neq NN_j$.

Suppose $SN_i!NN_i = SN_j!NN_j$ for two different nodes i and j. This means that $SN_i = SN_j$ and $NN_i = NN_j$. Let us assume that $SN_i = SN_j = s$, which is the number of times that these nodes have changed their numbers (see definition of SN). If the node number of node i at the system initiation time is ni and the node number of node j at the system initiation time is nj, then

$$NN_i = (ni + s) \text{ MOD } N$$

and

$$NN_j = (nj + s) \text{ MOD } N$$

If NN_i is to be equal to NN_j, then

$$(ni + s) \text{ MOD } N = (nj + s) \text{ MOD } N$$

The only way that this equality can hold is that if

$$ni + s = nj + s + KN \qquad \text{for } K \geq 0$$

or if

$$ni = nj + KN$$

Since $0 \leq ni < N$ and $0 \leq nj < N$ (see initial numbering of the nodes), the only value that K can have is 0, and therefore $ni = nj$, which contradicts the fact that all nodes are *uniquely* numbered at the system initiation time. Hence $ni \neq nj$, which means $NN_i \neq NN_j$ or $SN_i!NN_i \neq SN_j!NN_j$.

Note that besides being unique, SN!NN, generated as above, evenly distributes priority among the nodes of the system. In Figure 1, Node 3 (at the first row) has the highest SN!NN, whereas after the first node number change its SN!NN drops to the lowest (at the second row). Node 2, which had the second highest SN!NN at the beginning, will have the highest SN!NN after the first change (second row). Figure 1 shows how the highest SN!NN or lowest priority is passed from one node to another in a round-robin fashion.

There are two ways of initiating the node number changes. The first scheme calls for a timer at each node. A node changes its node number, according to the above scheme, when its interval timer expires. At this time the timer is reset and the sequence number is also incremented. The problem with interval timers is similar to the problem with physical clocks. To avoid this problem the second scheme can be used. In this scheme every node changes its node number after it generates M (a predefined integer number of) updates. The problem with this scheme is that lightly loaded nodes change their numbers more slowly than heavily loaded nodes. The tradeoffs between the two schemes must be investigated with regard to a specific application.

After a node changes its number, the LCR at that node can

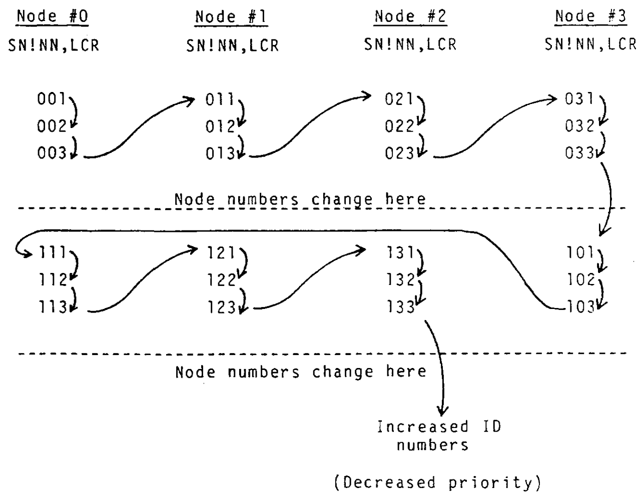

Figure 2—SN!NN,LCR for a 4-node system with M = 3 and LCR restart

	Node #0 SN!NN,LCR		Node #1 SN!NN,LCR	Node #2 SN!NN,LCR
	001		011	021
	002		012	022
	111		121	101
	112		122	102
A		improper SN_reset		
	221	021	201	211
	222	022	202	212
B		proper SN_reset		
	301	001	311	321
	302	002	312	322

Figure 3—Resetting SNs

be reset to 1 without threatening the local ordering of updates from the same node. This is necessary because otherwise LCRs may become undesirably large. Figure 2 shows some of the update ID numbers generated for a system of four nodes when each node changes its NN after generating $M = 3$ updates. This figure shows that updates generated from the same node are numbered in order of their generation. Therefore, even though LCRs are reset for each node number change, the local ordering is still preserved.

SNs similar to LCRs can grow large (although at a slower rate) and therefore require periodic resetting. Resetting SNs has to be done so that the properties of uniqueness, local ordering, and total relative ordering of the MOD numbering scheme are preserved. One has to be careful about when to reset a nodal SN. Since NNs and LCRs change in a circular manner, it is possible to generate two or more updates with the same ID when SNs are reset carelessly.

An example of a three-node system that changes a node's number after the node generates two updates is given in Figure 3. This figure shows that if SN of node number 0 is reset to 0 at Point A, the very next update generated at this node is numbered (0!2,1), which was first assigned to another update by Node 2 (first row, last column of Figure 3). In order to avoid this (and possible message transfer for synchronization), Node 0 can wait and attempt to reset its SN at Point B. There are two important properties associated with Point B. First, at this point, the new node number of every node is the same as its original number assigned at the system initiation time (0 for Node 0, etc). Second, if resetting is done at this point, the only possible conflicts are local conflicts: Node 0 might generate ID = (0!0,1), which was first generated by this node. This property eliminates the need for message transfers and synchronization with other nodes. Therefore, resetting SNs at this point will not cause uniqueness destruction of IDs if each node is only assured that all updates it has generated since its last SN reset are finished and out of the system. This might delay the numbering of updates if all previous updates are not finished. Considering the facts that each update is executed in a finite period and that resetting of SNs does not occur very often, the delay is not substantial. Note that SNs can be reset independently for each node and do not have to be reset for all nodes at the same time. Note also that resetting a SN at a node means starting IDs from the lowest possible number at that node. As far as the local ordering of updates is concerned, this does not matter, because all previous updates at this node are out of the system when resetting occurs.

A virtual ring among the nodes and a token circulating in this ring, similar to the scheme explained in Lelann,[6] can also be used instead of the numbering scheme presented above. In this scheme a token (or a sequencer [Reed[8]]) is circulating in a prespecified virtual ring among the nodes of the system. The token is given a token round number, TRN, that is set to 0 at system initiation time and is incremented for each complete rotation of the token in the ring. A node will change its node number every time it receives the token. The TRN is attached to update ID numbers instead of SNs; i.e.,

$$ID = (TRN!NN,LCR)$$

This scheme also provides for a unique identification and a fair priority scheme among the updates. One drawback to this scheme is the problem of token loss, which may occur if a node that has the token fails. Loss of the token, even though soluble,[6] can delay the numbering procedure and hence contribute to delay in the execution of the updates. Link failures can cause similar problems. The MOD numbering scheme, on the other hand, does not require communication among the nodes to generate timestamps. This means that link failures do not affect timestamp generation. As far as node failures are concerned, a node can fail without interrupting other nodes' timestamp generation. After a node recovers, it can resume its timestamp generation where it left off. Because of the problem associated with the scheme using circulating tokens, it is preferable to use the MOD numbering scheme for numbering the events (updates) in a distributed system.

In summary, the scheme has four properties:

1. IDs generated using this scheme are unique.
2. For a given node, update IDs increase monotonically, and therefore updates generated from a node preserve the order of their generation (local ordering).
3. As discussed above, priority of nodes changes in a round-robin fashion and is not pre-fixed.
4. Control is local and therefore communication cost is low.

CONCLUSION

A numbering scheme that generates globally unique update IDs has been presented. The scheme dynamically changes the node numbers; this change results in an even distribution of priority across the nodes. The scheme does not require physical clocks and therefore avoids all the problems associated with synchronizing them. The MOD numbering scheme could be employed by update algorithms[1,7,9,11] in place of numbering schemes using fixed node numbers and physical clocks. This is expected to improve the performance of these algorithms.

REFERENCES

1. Bernstein, P. A., J. B. Rothnie, N. Goodman, and C. A. Papadimitriou. "The Concurrency Control Mechanism of SDD-1: A System for Distributed Data Bases (The Fully Redundant Case)." *IEEE Transactions on Software Engineering,* SE-4, (1978).
2. Date, C. J. "An Introduction to Database Systems." Addison-Wesley, Reading, Massachusetts, 1977.
3. Everest, G. C. "Concurrent Update Control and Database Integrity." In J. W. Klimbie and K. L. Koffeman (eds.), *Database Management.* Amsterdam: North-Holland, 1974, pp. 241-268.
4. Eswaran, K. P.; J. N. Gray, R. A. Lorie, and I. L. Traiger. "The Notions of Consistency and Predicate Locks in a Database System." *Communications of the ACM,* 19 (1976).
5. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM,* 21 (1978).
6. Lelann, G. "Algorithms for Distributed Data-Sharing Systems Which Use Tickets." *Proc. Third Berkeley Workshop on Distributed Data Base and Computer Networks,* University of California, Berkeley, CA, August 1978.
7. Rahimi, S. K., and W. R. Franta, "A Posted Update Approach to Concurrency Control in Distributed Data Base Systems." *Proc. 1st Intl. Conf. on Distributed Computing Systems,* IEEE, Oct. 1979.
8. Reed, D. P. "Naming and Synchronization in a Decentralized Computer System." Ph.D. thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, September 1978.
9. Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis, II. "System Level Concurrency Control for Distributed Database Systems." *ACM Transactions on Database Systems,* 3 (1978).
10. M. Stonebraker. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES." *3rd Berkeley Workshop on Distributed Data Management,* 1978.
11. Thomas, R. H. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases." *ACM Transactions on Database Systems,* 4 (1979).
12. Traiger, I. L., J. N. Gray, C. A. Galtieri, and B. G. Lindsay. "Transactions and Consistency in Distributed Data Base Systems." IBM Report RJ2555 (33155), May 1979.