1997

Timestamp Based Approach For The Detection and Resolution of Mutual Conflicts in Real-Time Distributed Systems

Sanjay Kumar Madria

Report Number:
97-030

TIMESTAMP BASED APPROACH FOR THE DETECTION AND RESOLUTION OF MUTUAL CONFLICTS IN REAL-TIME DISTRIBUTED SYSTEMS

Sanjay Kumar Madria

CSD-TR 97-030
May 1997

Timestamp Based Approach For The Detection And Resolution Of Mutual Conflicts In Real-Time Distributed Systems*

Sanjay Kumar Madria
School of Computer Science
University Sains Malaysia, 11800 Minden
Penang, Malaysia
skm@cs.usm.my
and
Department of Computer Sciences
Purdue University, West Lafayette, IN-47907

* This research is partly supported by a grant from NSF under NCR-9405931.

Abstract

In real-time distributed systems, it is desirable to allow read and write accesses to occur on replicated copies of database files in case of network partitions to increase availability. However, the system should detect mutual conflicts among the copies of the database files when sites from different partitions merge to form one partition. In this paper, we present a timestamp-based algorithm for the detection of both write-write and read-write conflicts for a single file in distributed systems during network partitions. Our algorithm allows operations to occur in different network partitions simultaneously. When the sites from different partition merge, the algorithm detects and resolves both read-write and write-write conflicts without taking into account the semantics of the transactions. Once the conflicts have been detected, some reconciliation steps for the resolution of conflicts have also been proposed. Our algorithm will be useful in real-time systems where timeliness of operations is more important than response time (delayed commit).

Keywords: Replication, Network Partitions, Conflicts, Timestamp, Reconciliation.

1. Introduction

Replication of database files is a key factor to improve availability in distributed systems. Replicated files at multiple sites permit accesses in the presence of some site failures or network partitions. This improves availability in distributed environment. However, when file replication is there, replicated copies must behave like a single copy. That is, all the copies of the same logical file must make available the same current value. This value should be the logical value in terms of the transactions executed on different copies of the same file.

To our knowledge, several proposed methods [1,2,3,4,5,6,7,12,13,14,15] enforce consistency of the database by permitting database files to be accessed only in one partition in case of network or site failures. Many of these methods put restrictions on the execution of different transactions without guaranteeing that the data files can be accessed in atleast one partition. Many of these algorithms handle only simple partitions (i.e., no multiple partitions). Most of these algorithms do not permit transactions to be backed out once they have been committed. Therefore, these protocols do not allow execution of conflicting transactions [1]. Thus, they guarantee the consistency of the database across the partitions by severely limiting availability.

In many real-time situations, it is desirable to keep the system functioning in the presence of some site failures or network partitions to increase availability. The operations may be allowed to execute independently in different partitions. However, the system will delay actual commit (i.e., the transfer of data to stable storage) until recovery is completed. This is, because there is a possibility of backing out some of the committed transactions. Thus, the processing of transactions in each partition will be consistent. However, global inconsistencies across the partitions may occur.

When the system is partitioned, each partition maintains the consistent data but cannot make sure that its actions do not conflict with the actions in the other partitions. In such cases, the conflicts are to be detected whenever any two partitions or some sites from different partitions merge. There are mainly two types of conflicting operations namely read-write and write-write [1] depending upon the order of executions of read and write operations. These conflicting operations are important as their order of executions affects the final database state. When sites from different partitions merge, read-write and write-write conflicts among the copies of the database files are to be detected and resolved. This will re-establish the consistency among the copies of the database files at all the sites within the new partition.

Parker et al. [8] have proposed the detection of only write-write conflicts for a single file using version vectors. However, resolving inconsistency is not straight forward and is essentially left to the user. This scheme has also been extended to the transactions which access more than one file [9]. However, it does not detect all inconsistencies and in fact, detect some false inconsistencies [10]. We think that if the scheme given in [8] can be extended using timestamps to deal with both read and write conflicts then it will be very useful in increasing availability in real-time distributed systems.

In [10], a precedence graph technique for both read-write and write-write conflict detection is proposed for replicated data in distributed systems. Conflicts are detected from the cycles in the global transaction graph caused by the independent running transactions in different partitions. The committed transactions in each partition form the local transaction graph. At the time of reconnection, a global transaction graph is formed. The cycles from the global transaction graph are detected and resolved by a transaction back out strategy to make the database

consistent. In the situations when millions of transactions access the single file each second, the algorithm has to keep track of all the committed transactions and their commit orders. Furthermore, it has to detect all the cycles among the committed transactions in the global transaction graph. Also, to bring back the value of a file to a consistent state, conflicts have to be resolved. Hence, the algorithm has high cost associated with it. Therefore, it may be worthwhile to detect and resolve conflicts without keeping track of transactions, or operations executed under the transactions.

One might think that with a simple timestamp scheme using synchronized clocks [11] in each partition, it would be possible to detect write-write and read-write conflicts among the copies of a single file. However, this is not possible as the operations (read and write) execute independently in each partition. Therefore, the conflicts may or may not occur even if reading or writing time of a file in one partition is less than the writing time of the same file in other partition. This is because these timestamps are independent of each other and belong to two different partitions. Hence, the detection of conflicts using simple timestamp scheme may not possible. This has also been stated in [8].

In an earlier attempt [8], the following strategy has been mentioned (no algorithm was given) for the detection of only write-write conflicts using timestamps for a single file. Whenever a file is modified, one marks it with the two update times namely the previous and the last. When two partitions merge, a check is made to find whether no update in the file has occurred or one copy of the file differs from the other by a single update. In such cases no conflict occurs, but in many complex situations, the approach fails [8]. For example, suppose $\{wT_9, wT_{11}\}$ and $\{wT_{10}, wT_{12}\}$ are two write timestamp elements associated with the copies of the same file in the two partitions say A and B, respectively. Each timestamp element represents the previous and the last write timestamps of the same file in the corresponding partition. When these two partitions merge, we compare the write timestamp elements of the partitions A and B to detect the possible conflicts. Observe that write timestamps wT_9 and wT_{11} of partition A are less than wT_{10} and wT_{12} of partition B, respectively. However, this does not detect whether a conflict is there or not for the following reasons. If these write timestamps correspond to independent updates in two different partitions then there will be a write-write conflict. Consider another situation where one of the write timestamp elements actually belongs to one of the previous partitions when the sites of the partitions A and B were together in one partition. Furthermore, suppose no further updates have taken place in the partition A since then. In this case, there will be no conflict. Therefore, the above scheme fails to detect whether conflict is there or not.

The timestamp-based approach given in this paper permits the operations to execute independently in various partitions and thus, allows possible inconsistencies to occur at the cost of more availability. We think that timeliness of operations in a real-time distributed system is more important than response time (real commit). That is, commits can be delayed until all the partitions finally merge into one partition but operations should be allowed to occur in real-time distributed systems. Our algorithm detects read-write and write-write conflicts when any two partitions or sites from different partitions merge. Our approach here uses read and write timestamps to detect and reconcile both read-write and write-write conflicts for a single file. Once inconsistencies have been detected, we provide some reconciliation steps to resolve conflicts. Our technique for resolving conflicts do not take into account the semantics of the operations that manipulated the file, and the semantics of the data being stored. Hence, our

scheme does not provide transaction oriented database recovery. Our scheme also assumes that all the transactions complete in their respective partitions before a partition occurs. That is, there is no active transaction at the time of a partition. However, as mentioned, no transactions can commit until all partitions finally merge into one partition. We, also, assume "read-one and write-all" approach within a partition. Also, within a partition, no conflicts are allowed. Our algorithm also handles multiple partitions.

2. Definitions

In this section, we formalize some definitions, to be used in the paper, as follows:

Definition 1: A network partition is said to occur when there are disjoint groups of sites such that no communication is possible between the groups. Each of the disjoint groups is called a partition that shares a common synchronized view of some set of files.

Definition 2: A W-timestamp vector for a file f is defined as a sequence of n timestamp elements where n is the number of sites in the system. Each timestamp element can be at the most two tuple where the first entry is the first update time and second entry is the last update time at that site. After a network partition occurs, a new W-timestamp vector is formed corresponding to the updates in the new partition. When an update occurs, only the timestamp elements corresponding to the sites present in that partition is updated and the others remain same.

For example, suppose s_1 and s_2 are two sites in the system. Let wT_i and wT_j be the initial and final update times at these sites respectively, for a file f, before a network partition occurs. Then the W-timestamp vector, when both the sites are in one partition, will be $<\{ wT_i, wT_j \}, \{wT_i, wT_j \}>$. After a partition, suppose sites' s_1 and s_2 go to two different partitions say A and B, respectively. If the first update occurs at site s_2 in the partition B at time wT_k then the new W-timestamp vector of partition B (and hence of site s_2) will be $<\{ wT_i, wT_j \}, wT_k>$. That is, only the timestamp element of the site present in the partition is updated and the other remains same.

Definition 3: A W-timestamp vector T_0 is said to dominate an another vector T_1 if the following holds.

1. T_0 and T_1 are the W-timestamp vectors associated with the copies of the same file in the two partitions, and 2. $Max \{ wT_i, wT_j \}_k \in T_0 \geq Max \{ wT_l, wT_m \}_k \in T_1$ for each k = 1,2,...,n where n is the number of sites in the system and $\{ wT_i, wT_j \}$ and $\{ wT_l, wT_m \}$ are the two timestamp elements of the W-timestamp vectors T_0 and T_1, respectively. Also, wT_i and wT_l are the initial and wT_j and wT_m are the last update times in their respective partitions. Intuitively, if T_0 dominates T_1, the copy of the file with vector T_0 has seen a superset of updates seen by the copy with vector T_1.

Definition 4: Two operations belonging to different transactions are said to be in conflict if they access the same data item simultaneously and one of the two operations is a write operation. In case both the operations are write, it is called a write-write conflict. If one of the two operations is a read then it is called a read-write conflict.

Definition 5: Two W-timestamp vectors are said to be in a write-write conflict if no one dominates the other. That is, the conditions given in Definition 3 are not satisfied.

Definition 6: An update partition row-vector for a copy of the file f is an ordered tuple of values. Each value corresponds to a site present in the partition. Initially, the value corresponding to a site is set to 1. Whenever an update occurs in a new partition, the values corresponding to the sites present in the partition remains 1 and the others are changed to 0. Moreover, if a site was absent in the last partition but appears into the new partition, its value is set to 1^*. This reflects that this site is new in this partition. This also says that the site's data value has been

made consistent with respect to the other sites present in its new partition. The next update in this partition will change 1^* to 1.

Definition 7: A partition graph PG(f) for any file f is a directed graph where the source node (and sink if it exists) is labelled with the names of all the sites in the network having a copy of the file f and all the other nodes are labelled with a subset of this set of names. Each node can only be labelled with the names of the sites appearing in its ancestor nodes in the graph; conversely every site name on a node must appear on exactly one node of its descendants.

Example 1: Consider a partition graph PG(f) with three sites A,B,C where each site has a copy of the file f as shown in Fig.1. Initially, sites A, B, C were in the same partition, and after multiple partitions, sites isolate themselves in different partitions. In the last merge, all the three sites again join the same partition.

2.1. When to Detect Conflicts

Let N be a node in the partition graph PG(f) for a file f. The read-write and write-write conflicts are to be detected at the node N if node N has two distinct fathers N_1 and N_2 such that the following conditions hold:

1. Some writes or reads or both of f has taken place at N_1 or N_2 or at both, or a conflict is previously detected at one or both nodes and may be some more reads or writes or both have occurred at one or both nodes.
2. There is no ancestor node of N having two identical fathers N_1 and N_2.

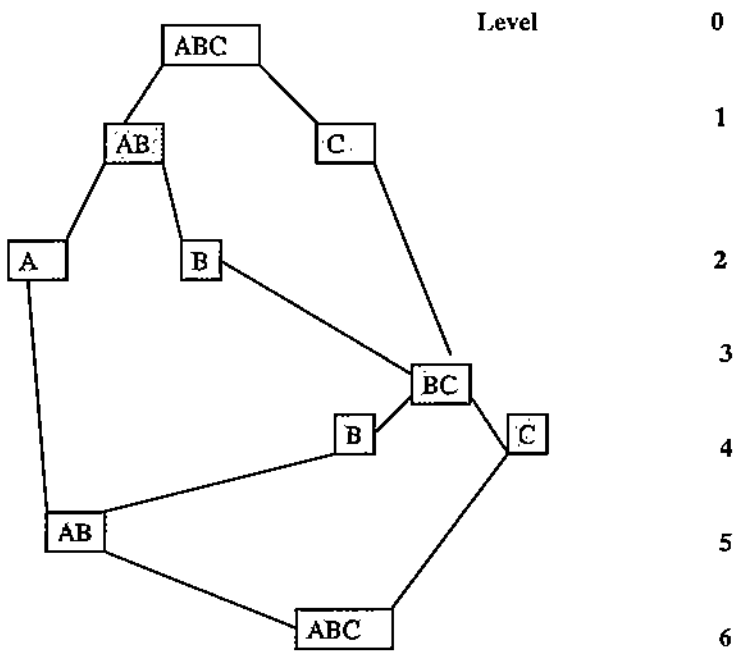

Figure 1. Partition graph

3. How to Keep and Update Timestamps and Row-vectors

For the detection and resolution of write-write and read-write conflicts, the algorithm needs only the first and the last write timestamps in each new partition. However, the algorithm needs all the timestamps corresponding to the read operations performed at a site. Therefore, the algorithm stores one read timestamp per read operation at the respective sites. The write timestamps are kept as a vector, called W-timestamp vector. The algorithm stores a list of write timestamp vectors at each site. Initially the W-timestamp vector consists of the first and the last write

timestamps corresponding to all the sites present in the system. If a write operation occurs after a network partition then the write timestamp entries at all the sites present in the new partition is updated. This is because we use write-all approach within a partition. This gives a new W-timestamp vector. A site will have one W-timestamp vector corresponding to each partition the site has travelled provided that the value of the file is updated at that site in each of those partitions. That is, each partition corresponds to one new W-timestamp vector in case there is an update in that partition. If there is no update in a new partition then this partition will not have any new W-timestamp vector. The last updated value of the file in each partition is attached with the W-timestamp vector of that partition. These timestamps are kept at each site until all sites merge into one partition. However, some of them will be discarded while resolving conflicts.

Our algorithm also associate a row-vector with each W-timestamp vector and read timestamp. The row-vector gives the information about the sites present in the partition at the time of read or write operations. As explained before, the first update in a new partition changes the entries in the row-vector to 1 for all the sites present in the partition and others to 0. The subsequent updates in the same partition will not affect the row-vector entries. In a new partition, if there is no new update, a read operation will return the value that will be the last updated value at that site in one of the previous partitions. Therefore, the row-vector attached with the read timestamp will be the row-vector attached with the W-timestamp vector of the corresponding old partition. On the other hand, if there is an update in the new partition, a read operation will return the new value. In this case, the row-vector associated with the read timestamp will be the row-vector attached with the W-timestamp vector of the new partition.

3.1. How to form a New Partition and Store First and Last Write Timestamps

When a write operation wants to update a file, it first checks the row-vector associated with the W-timestamp vector at its site. It then updates the copies of the file at all the sites (write-all approach) having entries as 1 and 1^* in the row-vector. In case the write operation is not able to update all the copies of the file at all the sites having entries as 1 and 1^* in the row-vector, it forms a new partition. To accomplish this, the home site broadcasts to all the sites it can communicate to join the new partition. Once it receives the response from a number of sites, it decides about its new partition. It then updates the copies of the file as well as the row-vectors at all the sites in its new partition. However, a read transaction will not be able to find out if there is a new partition as it reads the value only at its home site. Therefore, it may return an old value. However, it will be detected as it will generate a read-write conflict with respect to updates in other partitions.

Initially each site in the system is also associated with a flag bit 0. When a write operation performs the first update on the file (before this the file has the initial value), the flag bit is changed to 1 and the time of this write operation is stored. The 1 value of the flag bit means that the first update in the initial partition has occurred. Now onwards the write time of the next write will be stored but this will be updated for subsequent writes. When a write operation's home site forms a new partition, it will be the first operation that will update the file in the new partition. Therefore, its time will be stored as the first update time. For the next write operation within the same partition, its write time will be stored but will be updated every time for subsequent writes within the partition. This will determine the first and the last update time in each new partition.

4. Detection of W-W Conflicts

When two sites from two partitions merge to form a new partition, the algorithm compares the last W-timestamp vectors of the two merging partitions. Note that two W-timestamp vectors are said to be in write-write conflict if neither dominates the other (see Definition 3).

Example 2 : Consider a system consisting of four sites a, b, c, d. To detect write-write conflicts when two partitions merge, we compare the last W-timestamp vectors of these two partitions. Two cases can arise; either one of them dominates the other (see Definition 3) or they conflict (see Definition 4). For example, the W-timestamp vector $<wT_1$,{ wT_2 , wT_5 }, wT_4 , $wT_1 >$ dominates $<wT_1$, wT_4, wT_3, $wT_1 >$ but W-timestamp vectors $<wT_1$, wT_4 , wT_3 ,$wT_1 >$ and $<wT_1$,{ wT_2 , wT_3 }, wT_4 , $wT_1 >$ are in conflict whereas $<wT_1$, wT_4 , { wT_2 , wT_3 } , $wT_2 >$, $<wT_1$, wT_2 , wT_3 , $wT_4 >$ and $<wT_1$, wT_5 , {wT_6 , wT_7 }, $wT_7 >$ do not conflict (considering two at a time since detection of a conflict is assumed to be a binary operation in this paper) since the last one dominates the other two. For a more detailed example, see Appendix.

4.1. Resolution of W-W Conflicts

Once a W-W conflict for a file f between the last W-timestamp vectors of the file at the two merging sites, say s_1 and s_2 , has been detected, the next task is to resolve this conflict. To resolve the W-W conflict, the algorithm compares the last W-timestamp vector of the file f at site s_1 with the previous W-timestamp vectors of the same file stored at site s_2. This is under the assumption that site s_1 has seen more updates than site s_2. However, if the updates occur at site s_2 are not to be discarded for any reasons (e.g., critical updates) then the algorithm compares the last W-timestamp vector of the file at the site s_2 with the previous W-timestamp vectors of the site s_1. By comparing in this fashion, the algorithm always finds that at some point, the last W-timestamp vector at site s_2 dominates one of the W-timestamp vectors at site s_1. In other words, at this point of time, there was no conflict between the sites s_2 and s_1. Therefore, the algorithm will discard all the W-timestamp vectors at sites s_1 which are in conflict with the last W-timestamp vector at site s_2. It will also discard all the read timestamps at site s_1 after the last discarded W-timestamp vector. This is because these reads will be in conflict with the writes performed at site s_2. Therefore, it is desirable to detect write-write conflicts (if any) before read-write conflicts. This will reduce the number of comparisons required to detect read-write conflicts later.

After the site s_2 has joined the new partition, the last update time in the write timestamp element of site s_2 in the W-timestamp vector will be set to the maximum of the timestamp element of any site in the new partition. It is also marked with a *. Also, the last write timestamp of site s_2 in the old partition will become the first update time in the new timestamp element. For example, if {wT_m ,wT_n } is the timestamp element of any site in the new partition and the last write timestamp of the site s_2 is wT_i then the write timestamp element of the new joining site s_2 is kept as {wT_i ,wT_n^* }. We later see that the write timestamp wT_i is used for the detection of read-write conflicts. The timestamp entry wT_n^* denotes that the site s_2 has joined the new partition but no new updates have taken place at site s_2 in the new partition. It also informs that the value of the file at site s_2 is made consistent with the help of the value of a copy of the file at the site s_1 as exists at time wT_n. The entry in the row-vector corresponding to the site s_2 is also changed to 1^*. The entry marked with 1^* informs that this site is new in this partition. The next update at site s_2 in the new partition will change 1^* to 1. For examples, see Level 3 and Level 5 in Appendix.

4.2. Complexity of the Algorithm for the Detection and Resolution of W-W Conflicts

To detect a W-W conflict, the algorithm compares the two write timestamp vectors as explained above. The complexity of this comparison depends on the number of write timestamp elements to be compared which in turn depends on the number of sites in the system. That is, the number of the corresponding timestamp elements to be compared at the two merging sites will be same as the number of sites in the system. Therefore, the complexity for this part of the algorithm will be O(n) where n is the number of sites in the system. The complexity of the resolution part of the algorithm will be O(nm) (in worst case) where m is the number of write timestamp vectors with whom a write timestamp vector of the dominating site is to be compared and n is the number of sites in the system.

5. Detection of R-W Conflicts

The algorithm detects R-W conflicts only after the detection of W-W conflicts. Suppose the last W-timestamp vector at site s_1 dominates the last undiscarded W-timestamp vector at site s_2. In this case, the R-W conflicts are detected between the read timestamps stored at site s_2 and the write timestamp elements from the W-timestamp vectors stored at site s_1. First, the algorithm compares the latest read timestamp available at site s_2 with the write timestamp element of the file from the last W-timestamp vector at site s_1. Suppose the latest read timestamp of the file at site s_2 is less than the corresponding write timestamp element of the file at site s_1. In this case, the algorithm keeps comparing the read timestamp with the write timestamp elements from the previous W-timestamp vectors at site s_1 until one of the conditions given below is satisfied. Note that for the purpose of comparison, a read timestamp is always compared with the write timestamp associated with WT_n^*, and in the row-vector, the corresponding 1^* entry is treated as 0 since WT_n^* is not a real update. For a complete working example, see Appendix.

5.1. Conditions for the Detection of R-W Conflicts

Condition 1 : **If** $\text{Min}\{wT_m, wT_n\}_{s_1} < [rT_k]_{s_2} < \text{Max}\{wT_m, wT_n\}_{s_2}$

and the row-vectors attached with read and write timestamp elements differ

then R-W conflict

else no R-W conflict

Condition 2 : **If** $[rT_k]_{s_2} = \text{Max}\{wT_m, wT_n\}_{s_1}$ or $[rT_k]_{s_2} = \text{Min}\{wT_m, wT_n\}_{s_1}$

then R-W conflict

Condition 3 : **If** $[rT_k]_{s_2} > \text{Max}\{wT_m, wT_n\}_{s_1}$ and the row-vectors attached with read and write timestamp elements differ

then R-W conflict

else no R-W conflict

Note: $[rT_k]_{s_2}$ indicates the k_{th} reading time at site s_2 .

$\{wT_m, wT_n\}_{s_1}$ indicates that wT_m is the initial and wT_n is the final update times at site s_1 in a partition.

$\text{Min}\{wT_m, wT_n\}_{s_1} = \{wT_m\}_{s_1}$ and $\text{Max}\{wT_m, wT_n\}_{s_1} = \{wT_n\}_{s_1}$.

5.2. Correctness of the Conditions

We now argue the correctness of the conditions given above as follows.

Correctness of Condition 1 : In general, suppose that a read timestamp associated with a file at one site falls between the first and the last update timestamp associated with the same file at the other site. In this case, the conflict is there or not depends upon the following. If the row-vectors attached with read and write timestamps differ (i.e., the reading of a file in one partition is independent of the updates in the other partition) then a read-write conflict will occur. If the row-vectors are same then it implies that read returned the last write value of the file and since then there is no new update operation at the other site. Hence, there will be no R-W conflict.

Correctness of Condition 2 : Suppose a read timestamp at a site coincides with either the first or the last update timestamp at some other site. In this case, the row-vectors attached with the read and write timestamps will always differ. This is because same read and write timestamps implies that a read and a write operations are performed at the same time in two partitions. In the same partition, this is not possible as no conflicts are allowed within the same partition. Hence, there will be a read-write conflict.

Correctness of Condition 3 : Suppose a read timestamp of a file at a site in one partition is greater than the last write timestamp of the same file at some other site in a different partition. In this case, if the corresponding row-vectors differ then it will generate a read-write conflict. This is because different row-vectors implies that a read in one partition is independent of the last write of the same file in some other partition. If the row-vectors are same then there will be no conflict since same row-vectors imply that the read is consistent with the last write at the other partition. In other words, same row-vectors imply that both the sites have seen consistent updates. Therefore, read will return the correct value.

5.3. Complexity of the Algorithm for Detection of Read-Write Conflict

For the detection of read-write conflict, we compare each read timestamp at site s_2 with the write timestamp element at site s_1. That is, in worst case, we might have to compare one read timestamp with at the most m different timestamp elements. We also need some constant number of comparisons for the conditions given above. This includes some constant number of comparisons to compare the two row-vectors. The row-vector elements of the corresponding sites to be compared depends on n where n is the number of sites. Therefore, we assume that the number of comparisons needed in the conditions given above to be equal to some constant k. Therefore, in case we have to compare p read timestamps, the complexity of the algorithm in worst case will be O(pm+k). However, practically, the complexity of the algorithm will be much less than its worst case complexity due to the following reasons. First, many reads are discarded during the resolution of the write-write conflicts. Also, if a latest read timestamp is not in read-write conflict, earlier reads will not be in conflict. Therefore, we need not compare rest of the read timestamps with the write timestamp elements. Furthermore, if we keep track of last compared write timestamp element, the next read timestamp can be compared with this write timestamp element and then with its next preceding write timestamp element and so on.

5.4. Resolution of R-W Conflicts

Suppose a R-W conflict is detected between the two merging sites. In this case, the algorithm simply discards the read timestamp in conflict with the write timestamp element from the W-timestamp vector at the other site.

Similarly, the other read timestamps are also compared, and are discarded if they are in conflict with the write timestamp elements stored at other site.

Note that some R-W conflicts are detected and resolved automatically during the resolution of W-W conflicts.

Suppose a read timestamp at site s_2 is found to be not in conflict with the write timestamp element at site s_1. In this case, the remaining read timestamps at site s_2 will not be in R-W conflicts with the write timestamps at site s_1. Therefore, there is no need to compare the earlier read timestamps at site s_2 with the write timestamp elements at site s_1.

Suppose the last W-timestamp vector at s_1 dominates the last undiscarded W-timestamp vector at site s_2. In this case, there will not be any R-W conflicts between all the reads performed at site s_1 and all the updates performed at site s_2 before the last undiscarded W-timestamp vector.

6. Conclusion

In this paper, we have presented an efficient and useful technique for detecting and resolving read-write and write-write conflicts in real-time distributed systems based on timestamp approach. Here, an inconsistency has been assumed due to multiple users modifying different copies of the same file without mutually excluding one another. This situation will occur, for example, when network failures isolate these users in different partitions. Our scheme uses only some of read and write timestamps to detect and resolve conflicts. Our future work will be to test this scheme in a real world environment. In that case, we can also discuss about the space requirements for storing the read , write timestamps as well as row-vectors. Also, we intend to extend this scheme for more than one file.

References

[1] Bernstein, P., Hadzilacos, V., and Goodman, N., Concurrency Control and Recovery in Database systems, Reading, MA : Addision-Wesley,1987.

[2] El Abbadi, A., and Tough, S., Availability in Partitioned Replicated Databases, ACM Transactions on Database Systems, Vol. 4, No.2, pp.264-290, June,1989.

[3] Ellis C.A., A Robust Algorithm for Updating Duplicate Databases, In Proceedings of 2nd Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 1146-1158, 1977.

[4] Jajodia, S., and. Mutchler, Dynamic Voting Algorithm for Maintaining the Consistency of a Replicated Database, ACM Transaction on Database Systems, Vol.15, No.2, pp. 230-280, 1990.

[5] Stonebraker M., Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES, IEEE Transactions on Software Engineering, Vol. SE-5, pp. 188-194, May, 1979.

[6] Thomas R.F., A Solution of the Concurrency Control Problem for Multiple Copy Databases, In Proceedings of Spring COMPCON Feb.28-Mar.3, 1978.

[7] Tang, J., and Natarajan, A Static Pessimistic Scheme for Handling Replicated Databases, In Proceedings of ACM SIGMOD international Conference on Management of Data, 1989.

[8] Parker D.Scott, Gerald J., and Popek et al., Detection of Mutual Inconsistency in Distributed Systems, IEEE Transactions on Software Engineering, Vol. SE-9, No.3, May, 1983.

[9] Parker, D.S. and Ramos, R.A., A Distributed File System Architecture Supporting High Availability, In Proceedings of 6th Berkeley Workshop on Distributed Data Management and Computer Networks, pp.161-183,

1982.

[10] Davidson, S.B., Optimism and Consistency in Partitioned Distributed Database Systems, ACM Transactions on Database Systems, Vol. 9, No. 3, pp. 456-481, Sept., 1984.

[11] Lamport L., Time, Clocks, and the Ordering of Events in a Distributed System, Communication of ACM, Vol. 21, pp. 558-565, July, 1978.

[12] Nabil R. Adam, A New Dynamic Voting Algorithm for Distributed Database Systems, IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.3, pp.470-478, june,1994.

[13] D. Davcev, A Dynamic Voting Scheme in Distributed Systems, IEEE Transactions on Software Engineering, Vol.15, pp.93-97, Jan.89.

[14] B. Bhargava, Transactions Processing and Consistency Control in Distributed Systems, Journal of Management Information System, Vol.4, No.2, pp. 93-112, Fall, 1987.

[15] B. Bhargava and P.L. Ng., A Dynamic Majority Determination Algorithm for Reconfiguration of Network Partition, Information Science, Vol.4, pp.27-45, 1988.

APPENDIX : Example

Each node in the Fig. 1 corresponds to a partition for a file f during which the sites maintain an independent consistent view of the file f in their own partition. A conflict is detected when two partitions merge into one partition.

We have given below the six different levels through which the sites A, B, C travel and finally merge into one partition as shown in Fig.1. The W-timestamp vectors, and read timestamps, and their associated row-vectors are given in case of each partition.

Notations used :

1. A row-vector attached with a W-timestamp vector gives the information about the sites present when the updates start occurring in that partition.

2. A W-timestamp vector corresponds to a partition whereas a read timestamp corresponds to a site.

3. The row-vector $<111>_B$ attached with a read timestamp implies that reading is at site B. Also, the value read corresponds to the last update when all the three sites A,B,C were in the same partition. Similar notation has been used for other row-vectors of this type.

4. The read timestamp $<rT_4, rT_8><110>_B$ implies that first read at site B is at time rT_4 and the second read is at time rT_8. $<110>_B$ denotes that reads are with respect to the writes performed at the site B in the partition {AB}. Similar notation has been used elsewhere also.

Level 1 : Partitions of {ABC} are {AB} and {C}.

The W-timestamp vectors and read timestamps at partitions {AB} and {C} are as follows :

In partition {AB}

W-timestamp vectors	read timestamps
$<wT_1, wT_1, wT_1><111>$	$rT_2 <111>_A$
$<\{wT_3, wT_7\}, \{wT_3, wT_7\}, wT_1><110>$	$rT_9 <110>_B, <rT_4, rT_8><110>_A$

In partition {C}

$<wT_1, wT_1, wT_1><111>$	$rT_2 <111>_C, rT_4 <111>_C$

$<wT_1 , wT_1 , wT_5 > <001>$ $<rT_7 , rT_{10} ><001>_C$

Level 2 : Partitions of {AB} are {A} and {B}.

In partition {A}

W-timestamp vectors	read timestamps
$<wT_1 , wT_1 , wT_1 > <111>$	$rT_2 <111>_A$
$<\{wT_3 , wT_7 \},\{wT_3 , wT_7 \}, wT_1 > <110>$	$<rT_4 , rT_8 ><110>_A$
$<wT_{14} , \{wT_3 , wT_7 \}, wT_1 > <100>$	$<rT_{16} , rT_{19} ><100>_A$

In partition {B}

W-timestamp vectors	read timestamps
$<wT_1 , wT_1 , wT_1 > <111>$	
$<\{wT_3, wT_7 \},\{wT_3, wT_7 \}, wT_1 > <110>$	$rT_9 <110>_B$
$<\{wT_3, wT_7 \}, wT_{20} , wT_1 > <010>$	$rT_{22} <010>_B$

Level 3 : After merging of partitions {B} and {C} into {BC}. We assume that site B has seen more updates than C.

The last W-timestamp vector at site B dominates the first W-timestamp vector at site C and therefore, we discard rest of the W-timestamp vectors and the read timestamps after the last discarded W-timestamp vector. The read timestamp $rT_2 <111>_A$ is not in conflict with any W-timestamp vectors (see section 4.1) and therefore, it will remain as valid read at site C whereas $r T_4 <111>_C$, $<r T_7 , r T_{10} ><001>_C$ are discarded due to conflict. Also, $<011^* >$ implies that site C has joined the partition and the value of the file is made consistent with respect to the value of the file at site at time wT_{20} (shown by wT_{20}^* at the corresponding position of the site C in the W-timestamp vector).

In partition {BC}

W-timestamp vectors	read timestamps
$<wT_1, wT_1, wT_1 > <111>$	$r T_2 <111>_C$
$<\{wT_3, wT_7 \},\{wT_3, wT_7 \}, wT_1 > <110>$	$r T_9 <110>_B$
$<\{wT_3, wT_7 \}, wT_{20} , \{wT_1, wT_{20}^*\} > <011^*>$	$rT_{22} <010>_B , rT_{26} <011^* >_C$
$<\{wT_3, wT_7 \}, wT_{29}, wT_{29} > <011>$	$rT_{32} <011>_C$

Level 4 : After partition of {BC} into {B} and {C}, the vectors are as follows :

In partition {B}

W-timestamp vectors	read timestamps
$<wT_1 , w T_1 , w T_1 > <111>$	
$<\{w T_3 , w T_7 \},\{w T_3 , w T_7 \}, w T_1 > <110>$	$rT_9 <110>_B$
$<\{wT_3, wT_7 \}, wT_{20} , \{wT_1, wT_{20}^*\} > <011^*>$	$rT_{22} <010>_B$
$<\{wT_3, wT_7 \}, wT_{29}, wT_{29} > <011>$	
$<\{wT_3, wT_7 \}, wT_{36}, wT_{29} > <010>$	$rT_{39} <010>_B$

In partition {C}

W-timestamp vectors	read timestamps
$<wT_1, wT_1, wT_1> <111>$	$rT_2 <111>_C$
$<\{wT_3, wT_7\}, \{wT_3, wT_7\}, wT_1> <110>$	
$<\{wT_3, wT_7\}, wT_{20}, \{wT_1, wT_{20}^*\}> <011^*>$	$rT_{26} <011^*>_C$
$<\{wT_3, wT_7\}, wT_{29}, wT_{29}> <011>$	$rT_{32} <011>_C$

Level 5 : After merge of partitions {A} and {B} into {AB} with the assumption that site A has seen more updates than site B. The last W-timestamp vector at site B dominates the second W-timestamp vector at site A (see Level 2) and therefore, we have discarded rest of the W-timestamp vectors along with corresponding bad reads and their read timestamps.

In partition {AB}

W-timestamp vectors	read timestamps
$<wT_1, wT_1, wT_1> <111>$	$rT_2 <111>_A$
$<\{wT_3, wT_7\}, \{wT_3, wT_7\}, wT_1> <110>$	$rT_9 <110>_B, <rT_4, rT_8> <110>_A$
$<\{wT_3, wT_7\}, wT_{20}, \{wT_1, wT_{20}^*\}> <011^*>$	$rT_{22} <010>_B$
$<\{wT_3, wT_7\}, wT_{29}, wT_{29}> <011>$	
$<\{wT_7, wT_{36}\}, wT_{36}, wT_{29}> <1^* 10>$	$rT_{39} <010>_B$

Level 6 : After merge of {AB} and {C} into {ABC} with the assumption that sites A and B dominates the site C.

In partition {ABC}

W-timestamp vectors	read timestamps
$<wT_1, wT_1, wT_1> <111>$	$rT_2 <111>_C, rT_2 <111>_A$
$<\{wT_3, wT_7\}, \{wT_3, wT_7\}, wT_1> <110>$	$rT_9 <110>_B, <rT_4, rT_8> <110>_A$
$<\{wT_3, wT_7\}, wT_{20}, \{wT_1, wT_{20}^*\}> <011^*>$	$rT_{22} <010>_B, rT_{26} <011^*>$
$<\{wT_3, wT_7\}, wT_{29}, wT_{29}> <011>$	$rT_{32} <011>_C$
$<\{wT_7, wT_{36}^*\}, wT_{36}, wT_{29}> <1^* 10>$	$rT_{39} <010>_B$
$<wT_{45}, wT_{45}, wT_{45}> <111>$	