Enhancing Availability in Distributed Real–Time Databases

Kwei–Jay Lin and Ming–Ju Lin

Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, Illinois 61801
(217) 333–1424
{*klin, mlin*}@*p.cs.uiuc.edu*

Abstract

One of the issues in distributed databases is to maintain the data consistency when a database is replicated for higher availability. In a real–time database system, availability may be more important than consistency since a result must be produced before a deadline. We propose techniques to increase the availability in a partitioned real–time database. We also suggest that a transaction may execute even when the most up–to–date information is not available or when a serializable execution cannot be guaranteed. As long as data integrity is maintained, serializable execution may not be necessary.

1. Introduction

In many distributed systems, communication networks are vulnerable. When a partition occurs due to network failures in a replicated distributed database, the main concern is to maintain the data integrity. Replicated copies in two different partitions should not be accessed independently without any restriction. Such accesses (including reads and writes) are correct only if they preserve the *one–copy consistency*. Protocols have been proposed to guarantee the one–copy consistency [Bernstein and Goodman 83]. It has also been shown that if concurrent transactions in different partitions do not have write–write, write–read or read–write conflicts, the data integrity will not be violated (assuming each transaction, when executed alone, maintains the integrity.) In other words, concurrent transaction executions are correct if they are *serializable*.

In a *hard* real–time system, however, a transaction must be completed before its deadline or the execution is considered a failure. In such circumstances, we not only want to attain database consistency, but also require a database to provide high availability. In fact, database availability may be more important than its consistency for some applications. This is because data inconsistency may be removed upon detection but lost time can never be recouped. For example, suppose part of an automated factory is partitioned from the rest of the factory so that a majority of the copies of a data item are not available. Using most (pessimistic) replication control techniques [Davidson et al 85], the execution in this part of the factory may be forced to stop since it cannot access the data. However, it may be desirable or even necessary to continue the operation in the partition regardless of the inaccessible data. The cost of providing no service at all may be much higher than that of a degraded service.

This work was partially supported by a contract from the ONR (N00014–87–K–0827).

In this paper, two quorum protocols are proposed to increase the database availability when a network is partitioned. Our protocols were motivated by the fact that most real–time database systems are static and well–defined. Furthermore, we suggest that sometimes the data consistency may be temporarily neglected if it can be reestablished later. The rest of the paper is organized as follows. In Section 2, we discuss major design problems with database systems which must provide real–time responses. Related work in handling network partitioning in non–real–time setting is presented in Section 3. Section 4 presents the definition of quorums and the algorithm to use them. We conclude the paper in Section 5.

2. Real–Time Databases

Databases for real–time applications have many important issues which cannot be handled by algorithms for non–real–time databases. One of the main differences between the two is the close relationship between a real–time database and its service environment. In a real–time system, one cannot consider only the consistency issue for the data values within the database, but one must also make sure the values are consistent with the corresponding values in the real world. Another issue is that, when using a real–time data, we cannot simply assume that the most up–to–date value is the most desirable one. We should use a value that is consistent (in time) with the rest of the data used. In this section, we discuss such features in real–time databases.

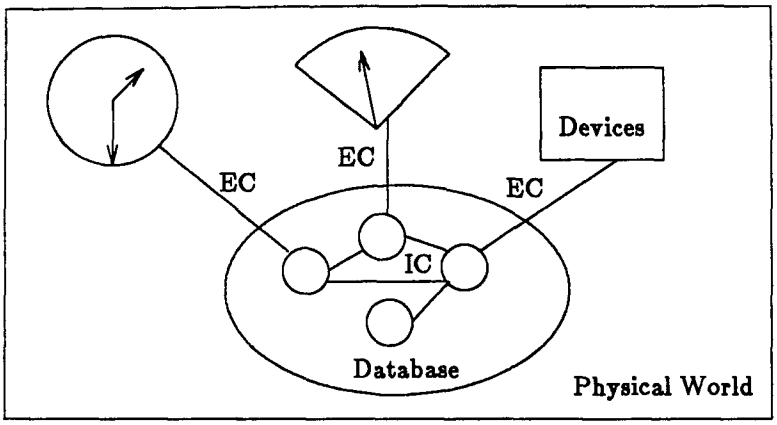
2.1. External Consistency

In many systems, a database is created and maintained to provide a basis on which transactions are to take place. This basis is called the *logical world.* The logical world is designed to embody all relevant real–life objects and variables. After the database is in operation, however, the logical world may not have any direct interaction with the real world. All updates to the database are executed in form of transactions. Consistency within this logical world, which will be referred to as *internal consistency,* is traditionally maintained by enforcing the serialization [Bernstern et al. 87] of all transactions and is usually considered as the most important correctness criterion.

In a real–time system, the role of the database is no longer the center of the universe but only one of the many devices in the system. The ultimate goal of the whole system is to maintain the correct operations of all devices as defined by the mission of the system. In this type of system it is critical that the database mirror the *physical world.* The database is dependent upon frequent interactions with the physical world to obtain accurate and timely data (Figure 1). A correct system state must ensure the consistency between the physical world and the database, which will be referred to as *external consistency.* Although there may be internal inconsistency within the database, resolving inconsistencies with the physical world may take precedence.

Traditionally, resolving internal inconsistencies involves undoing or rolling back transactions in conflict. However, it makes no sense to roll back a real–time database to an earlier state which no longer reflects the physical world. For example, consider a system which is responsible for monitoring and controlling the activities of a robot. Suppose the robot performs an operation which breaks a physical object into pieces. Once the event takes place, there is no way to cancel it. Even if the database then finds the operation violates some internal consistency, it is useless to "undo" the operation internally. Instead, the database can only regain its internal consistency by making all its data reflect the external reality.

To maintain external consistency, transactions reflecting irreversible external events must not be blocked or undone. When an update transaction is initiated to record what has happened



IC: Internal Consistency
EC: External Consistency

Figure 1. Real–time database consistency

in the physical world, even if the data is being used by a transaction in progress, the update may have to take precedence. Any conflicting transaction should either be aborted or terminated prematurely.

2.2. Handling Time

Most transactions in a *hard* real–time database have a deadline. The deadlines are not so critical to transactions in a *soft* real–time system but still should be met as much as possible. To meet all deadlines in a system requires powerful scheduling algorithms and careful system implementations. Much research has been done in this area and much more work is still required. We will not discuss the topic since it is beyond the scope of this paper.

Besides transaction deadlines, a real–time database often contains data items whose values change frequently with time. A value is current only for a certain amount of time. Many transactions can use a value only when it is still current. However, transactions may read different data items at different points of time. It is important for such transactions to use values which are *temporal consistent.* We say that the values of different data are temporal consistent if they exist "approximately" at the same time. For example, two sensors may read the x–position and y–position of an object independently. When updating the database, the two values may not be entered at exactly the same time. A transaction which is reading these two values must make sure that they are the x– and y– positions for the object at the same point of time.

The construct of atomic action sometimes can be used to ensure that all data entered in the database are temporal consistent or they are not entered at all. A transaction can also specify an atomic action to prevent temporal consistent data from being changed between reads. However, sometimes it may be impossible to do so. If updates come from separate physical devices which have different update frequencies, they may be difficult to be included in one atomic action. Moreover, many atomic action implementations prevent data in use from being revealed to other actions until the action commits. This may not be desirable when many actions have deadlines to meet.

We suggest that real–time databases should keep multiple timestamped data versions to provide some history of data values. It is then up to the users to decide which version is good for them. In other words, data consistency can be decided at the time of reads rather than the time of writes. Such "lazy evaluation" is advantageous since not all transactions may need the same consistency. As a side benefit of this, the database can employ non–blocking concurrency control protocols. If a data being used is still available to the transactions in progress, a writer may produce a new version at any time as long as there is no read–write conflict.

2.3. Process Population

Real–time systems usually consist of a *fixed number* of long–life (logical or physical) processes. Processes may change their locations, but not their duties. Transactions are initiated by processes to perform specific jobs periodically or sporadically. Some transactions are less frequently invoked. But when they are, they must be executed immediately. For example, error recovery is started only if there are abnormal system events. We sometimes use the terms "process" and "transaction" interchangeably.

Since the number of processes in a real–time database is predetermined, we may identify all reader and writer processes for a data item. Given a temperature record in a database, we know that only one or a few sensor processes may update the value. In addition, we can also find out what processes may read the temperature from the database. All these information can be used to improve the data availability.

One way to utilize the knowledge is to replicate a data item according to its reader/writer distribution. Processes with high access frequencies are provided with their local copies to speed up read operations. Another way to improve the database performance is to assign access privileges according to the access patterns. We will show how this can be done in Section 4.

3. Related Work

Many concurrency control protocols have been proposed to maintain consistent replicated data in distributed databases. With different emphases on trade–offs between consistency and availability, these protocols can be classified as pessimistic and optimistic. Examples of pessimistic strategies are primary copy [Alsberg and Day 76], tokens [Minoura and Wiederhold 82], and voting [Gifford 79]. The primary copy method designates one copy of an item as the primary copy which holds the current value. The tokens scheme associates a token with each item. Only the site holding the token is permitted to access the item. In the voting approach, each copy of the data is assigned a number of votes. A transaction can access the data if it collects a majority of the votes. Since pessimistic strategies always make worst–case assumptions, any operation that could lead to inconsistency is avoided. Hence the consistency is maintained at the expense of availability.

Optimistic approaches, on the contrary, do not block an access when it is requested. Database inconsistencies are resolved only after they are discovered. The probability that an inconsistency will occur is assumed to be small. Examples of theses schemes are version vectors [Parker et al. 83], the optimistic protocol [Davidson 82], and data–patch [Garcia et al. 83]. In the version vectors method, each copy of an item is associated with a vector which records the number of updates for the copy. Conflicts are detected by comparing the version vectors for the same item from different partitions. The optimistic protocol constructs a precedence graph of data access to detect conflicts. Data–patch is a method that allows the database administrator (DBA) to specify rules for integrating divergent databases.

Conventionally, serializability is the measure used to guarantee the correctness of concurrent transaction executions. Garcia and Wiederhold [Garcia and Wiederhold 82] proposed two different consistency levels : strong consistency and weak consistency. If a transaction requires that its execution be serializable with any other transaction, it is processed with strong consistency. For read–only transactions, their executions do not have to be serializable with all update transactions; they are said to require only weak consistency. With two levels of consistency, better throughput can be accomplished in a partitioned database.

4. Semantic Quorum Approach

When a partition occurs, we assume it is "clean". If the two sites are in the same partition then they can communicate with each other; otherwise they cannot. We also assume that site failures are distinguishable from network failures and each site detects a network failure instantaneously. After a partition has occurred, each site sends out reconfiguration messages to look for sites which are still connected. We assume no malicious site exists to attack the reconfiguration protocol.

To achieve maximum availability, three kinds of quorums are defined for each data item: *data*, *user* and *integrity*. The data quorum is used by replicated data copies to ensure that operations on the data are serializable. We use read and write quorums to improve the performance as in [Gifford 79]. We introduce two new quorums: the user quorum for processes which share the data, and the integrity quorum for data involved in an integrity constraint. In this section, we discuss how these quorums are defined and the algorithm for using them.

4.1. User Quorum

Conventional distributed databases have no knowledge of when and where a transaction may be started. During a network partition, the only information a database may use is the number of data copies. Any transaction which can collect a quorum of copies is allowed to use the data. In real–time systems, this may not be a good idea since the purpose of data replication may be simply for convenience. The partition with a majority of the data copies may not be the partition which needs the data most frequently or most importantly. For example, the site which is connected to an external device from which new values are produced may not be in a partition with a majority copies. To maintain the external consistency, we should allow the site to at least record all versions produced even though most other (reader) transactions cannot access the values immediately.

We propose the user quorum protocol so that a partition which has a majority users for a data item is granted the access privilege to the data. In other words, a partition with only a minority or no writers should make the data to be read–only regardless of how many data copies it may have. The reason for this is quite simple: data is passive but transactions are active. To achieve better performance and availability, active processes should be continued as much as possible. It is useless to grant a privilege to the transactions which do not need it.

The votes in the user quorum for a data item are assigned to the processes who may update the item. The votes are collected only when a partition occurs. When a database is perfectly connected only the data quorum is used for concurrency control. When a process detects a partition, the time is recorded and the system is switched to the "partitioned" mode. The total number of writer processes in the partition is compared with the user quorum. If the number of writer processes is greater than the old user quorum, the data quorum is updated to accommodate writes in the partition. Otherwise, the data quorum is changed to disallow any update.

Each partition thus has the privilege to utilize data according to the actual need of the processes in the partition. Furthermore, with the help of dynamic quorum assignment, no two processes may update an item inconsistently either in the same or different partitions. As in many data quorum protocols, no majority processes may exist in any partition. To achieve even higher availability, we propose a more optimistic approach in Section 4.3.

User quorum is different from the weighted voting protocol for replicated data [Gifford 79]. User quorum is defined for active users. The protocol can be used in the dynamic environment where processes migrate from one site to another. Weighted voting protocol is more static since vote reassignment for a data quorum is not easy to do.

4.2. Integrity Quorum

In real–time systems, it may not be possible to serialize all concurrent operations. This is especially true if there are long–life processes. Suppose there are two concurrent processes in a robot system, the arm and the eye. The eye process enters the positions of the arm and a moving object into the database periodically, while the arm process calculates the movement for the arm to grab the object. The arm needs to read the positions written by the eye. On the other hand, the arm changes its position which will be read by the eye. The two processes are not serializable. Some other criteria for the system correctness may be necessary.

The correctness of a database actually is decided by its integrity constraints. As long as these constraints are maintained, any means to achieve high availability is acceptable. However, as was pointed out in [Davidson et al. 85], the total number of constraints involved in a database usually is so large that it is impractical to check all constraints all the time. This is the main reason why most database systems instead use the serializability as the correctness measure of concurrent transactions. But since real–time systems usually have a fixed number of processes and the databases are statically structured, it may be feasible to specify a small set of integrity constraints which are the most critical to the system's correctness. Whenever a data item in a constraint is to be updated, the transaction must make sure that no other transactions will be updating the data in the constraint in the same time to violate the constraint.

When a real–time database is partitioned, if all data items involved in a constraint are writable in one partition, they can be updated in that partition as normal. If only part of the data in a constraint are writable in a partition, those data unwritable might be updated in other partitions inconsistently. One pessimistic solution is to disallow any data update in any partition. A better solution is to allow processes in only one of the partitions to update the data and to maintain the constraint. We choose the partition with a majority writable items to have the privilege. For example, suppose that a banking database has the following constraints:

$$checking + saving + investment \geq 0$$

The investment account is kept by an investment branch of the bank, while the checking and saving are kept by a local branch. When the local branch is partitioned from the investment branch, an independent withdrawal in each branch may cause the total to be less than zero. Instead of stopping all branches to maintain the constraint, at least one of the two branches can still proceed. In our protocol, the partition which has a majority data items in the constraint is the choice since it usually allows more transactions to proceed. In this example, the local branch is allowed to update the checking and saving accounts. The investment branch cannot update any data in this constraint or the database integrity may be violated.

4.3. Consistency Levels

When a database is partitioned, we assume that everything else except the network communication is still working. Therefore, the data version at any site is not corrupted and it reflects a valid or used–to–be valid value. Moreover, any data copy is accessible by the processes in the same partition; so it is still "available" in some sense. For example, read–only transactions may find it perfectly useful as long as it is the result of some serializable updates.

Using the user and integrity quorums, only part of the data items may be writable in each partition. If all data to be used by a transaction are writable in the partition, the transaction may proceed as usual. A new set of quorums may be assigned but the transaction is not interrupted. This is ideal since the partition does not affect the transaction. If not all data required by a transaction are writable in a partition, using the integrity quorum, the transaction may conclude that those read–only items will not be updated in other partitions. If so, the transaction may still proceed without interruption. In both cases, the consistency of the database is not compromised.

In case that a transaction must finish before a deadline but some data are not available, the transaction may want to proceed with possibly out–of–date data values. For certain transactions, as long as the data used are temporal consistent, they may still be acceptable. For many read–only transactions, this temporal consistency is all that is required.

In the most extreme case when a result is needed but neither an up–to–date nor a temporal consistent data set is available, a transaction still may proceed. Such results are usually needed to provide temporary response or to prevent some external situations from getting worse. The results may be corrected after a more consistent data is available. This is an optimistic approach which should be used only if compensating or undo actions are possible.

4.4. The Algorithm

For every data item in the database, a quorum vector (t, c, rq, wq, uq) is assigned, where t is the last time when the data was updated, c is the number of votes available, rq, wq, uq are the data read, data write, and user quorums respectively. When a system is in its normal operation, we use data quorums for concurrency control. We allow a data copy to have more only one votes. For each item, any transaction must have at least rq votes to read it, and wq votes to write it. Data quorums are assigned according to the following constraints:

(1) $rq + wq > c$;

(2) $2wq > c$.

The first constraint is to avoid read/write conflicts and the second write/write conflicts.

When a transaction is initiated, the process should specify the desired *consistency level*. Each transaction also has a set of integrity constraints predefined. The data involved in each constraint are checked to see if the partition has enough integrity votes for the transaction to execute. If not, the transaction is aborted right away. The transaction may specify three consistency levels:

(1) *strong consistency:* This requires the transaction execution to be serializable or consistent with any other update transactions. If all data items needed by a transaction are writable or if the transaction has the integrity quorum for all constraints it has, we may proceed the transaction as if there is no partition at all. Otherwise, the transaction is aborted. The strong consistency are usually used by transactions which either maintains internal consistency or produces a result which is irreversible.

(2) *weak consistency:* In this case all operations in the transaction are read only. No update to the database will be performed. The transaction may execute based on out–of–date data as long as they are temporal consistent.

(3) *optimistic consistency:* The transaction must produce a result regardless of the data consistency, or the transaction can be undone if an inconsistency is detected. In such cases, a transaction may proceed with data which are not totally current or temporal consistent. A transaction requires only the optimistic consistency may find all data perfectly available. If so, the transaction will not be using the optimistic consistency. If the transaction has to use the optimistic consistency, all updates performed must be recorded in the system log and examined when the partitions rejoin.

When a network partition occurs, the available data votes, c', and the user number, u, in each partition are collected to create a new quorum vector. The quorum vector is defined for strong consistency. Transactions want to execute using optimistic consistency do not have to follow the quorums. In order to reassign the quorums, the number of user processes in the partition is compared with the old uq. Two cases are possible:

(1) $u \geq uq/2$: this means that a majority of the user processes are in the partition. The new read and write quorums will be assigned based on the total number of votes available in the partition. A possible quorum vector is $(t,\ c',\ \lceil c'/2 \rceil,\ \lceil (c'+1)/2 \rceil,\ uq)$.

(2) $u < uq/2$: in this case, the quorum vector will be assigned as $(t,\ c',\ \infty,\ \infty,\ \infty)$. All quorums are assigned to ∞ because no modification is allowed. However, an implicit read/write quorum is used for optimistic transactions which must produce a result. The implicit quorum is $\lceil (c'+1)/2 \rceil$ since more than one optimistic transactions may exist in the partition. All reads and writes in the same partition should be serialized.

When the partitions are reconnected, all update histories in all partitions are examined. If no transaction has used the optimistic consistency, all data are merged according to the timestamp ordering. If some transaction did use the optimistic consistency, the database may have to undo or compensate some operations. The updates under the optimistic consistency must not be in conflict with either the updates using the strong consistency or other updates using the optimistic consistency. The detailed algorithm for merging histories will be presented in the future.

4.5. An Example

We use the banking database system example in Section 4.2 to show how the algorithm works. The banking system (Figure 2) has two branches and three accounts. There are three processes in the system: *teller*, *interest* and *planner*. The teller process updates the balances of the checking and the saving accounts whenever there is a withdrawal or deposit by the customer. The interest process enters the interests into the checking and the saving accounts. These two processes reside in the local branch. The planner process transfers money between the checking and the investment accounts. The planner is located in the investment branch. The checking and the saving replications in the local branch have two data votes each, while each copy in the investment branch has only one vote. The investment account exists only in the investment branch. The integrity constraint for the accounts is as in Section 4.2, i.e.

$$checking + saving + investment \geq 0$$

The quorums for both the checking and the saving accounts initially are $(t,\ 3,\ 2,\ 2,\ 2)$. The quorum for the investment account is $(t,\ 1,\ 1,\ 1,\ 1)$. Thus the teller can read the checking

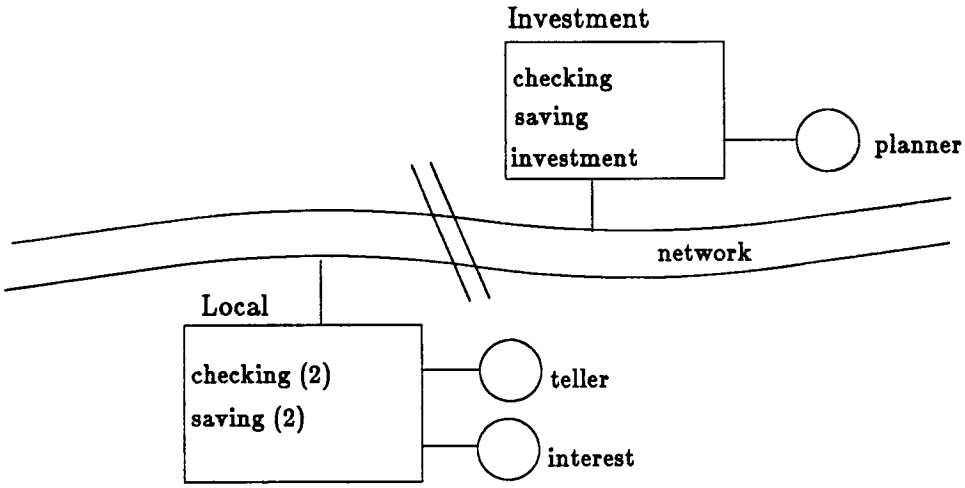


Figure 2. Banking system example

balance locally, while the planner must get the remote checking copy in the local branch for both read and write operations. When the two branches are partitioned, both will update their quorums independently. The quorum for the checking account in the local branch will be changed to $(t, 2, 2, 2, 2)$. The quorum for the same data in the investment branch will be changed to $(t, 1, \infty, \infty, \infty)$. This allows the checking account to be writable in the local branch but not in the investment branch.

Transactions which have real-time constraints may want to proceed regardless of the partition. For example, the planner process may need to transfer some money immediately due to market conditions. If so, the planner may go ahead and transfer the money but it also needs to enter the event in the system log. When the network recovers, the transfer transaction will be checked to see if the local branch has also updated the checking account. If the local branch did, the transfer transaction must be undone and then redone. If the local branch has never written into the checking during the partition, the result of the transfer is copied to the local branch as the current value for the checking account.

5. Conclusion

We have proposed the semantic quorum scheme which may enhance the availability of distributed real-time databases during network partitions. For replicated databases, we introduce the idea of *user quorum*. It is more productive to grant access privileges to a partition with a majority users rather than a partition with a majority data copies. In this way more processes, thus more work, can be performed. To allow non-serializable but correct executions, we suggest that transactions use *integrity constraints* and *integrity quorums* to prevent conflicting operations. Lastly, to facilitate the demand for producing a result before deadlines, we discuss various execution strategies for different consistency levels.

References

[Alsberg and Day 76]
Alsberg, P.A., and J.D. Day, "A principle for resilient sharing of distributed resources," *Proc. 2nd Int. Conf. Softw. Eng.,* Long Beach, CA, pp. 627–644, 1976.

[Bernstern and Goodman 83]
Bernstein, P.A., and N. Goodman, "Multiversion concurrency control – Theory and algorithms," *ACM Trans. Database Systems,* vol. 8, pp. 465–483, Dec. 1983.

[Bernstern et al. 87]
Bernstein, P.A., V. Hadzilacos, and N. Goodman, *Concurrent and Recovery in Database Systems,* Addison–Wesley Co., Massachusetts, 1987.

[Davidson 82]
Davidson, S.B., "An optimistic protocol for partitioned distributed database systems," Ph.D. thesis, Dept. EECS, Princeton Univ., NJ, 1982.

[Davidson et al. 85]
Davidson, S. B., H. Garcia–Molina, and D. Skeen, "Consistency in Partitioned Networks," *ACM Computing Surveys,* vol. 17, pp. 341 – 370, Sep. 1985.

[Garcia and Wiederhold 82]
Garcia–Molina, H., and G. Wiederhold, "Read–only transactions in a distributed database," *ACM Trans. Database Syst.,* vol. 7, pp. 186–213, Jun. 1982.

[Garcia et al. 83]
Garcia–Molina, H., et al., "Data–Patch: Integrating inconsistent copies of a database after a partition," *Proc. 3rd Symp. Reliability in Distributed Softw. and Database Systems,* pp. 38–48, Oct. 1983.

[Gifford 79]
Gifford, D.K., "Weighted voting for replicated data," *Proc. 7th Symp. Operating Systems Principles,* pp. 150–162, 1979.

[Minoura and Wiederhold 82]
Minoura, T., and G. Wiederhold, "Resilient extended true–copy token scheme for a distributed database system," *IEEE Trans. Softw. Eng.,* vol. SE–8, pp. 173–189, May 1982.

[Parker et al. 83]
Parker, D.S., et al., "Detection of mutual inconsistency in distributed systems," *IEEE Trans. Softw. Eng.,* vol. SE–9, pp. 240–247, May 1983.