Replicated Data Management in Distributed Database Systems

Sang Hyuk Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

Replication is the key factor in improving the availability of data in distributed systems. Replicated data is stored at multiple sites so that it can be accessed by the user even when some of the copies are not available due to site failures. A major restriction to using replication is that replicated copies must behave like a single copy, i.e., mutual consistency as well as internal consistency must be preserved. Synchronization techniques for replicated data in distributed database systems have been studied in order to increase the degree of concurrency and to reduce the possibility of transaction rollback. In this paper, we classify different synchronization methods by underlying mechanisms and the type of information they use in ordering the operations of transactions, and survey some of the replication management methods appeared in the literature.

1. Introduction

A distributed system consists of multiple computer systems (called *sites)* that are connected via a communication network. Distributed systems have been claimed to be inherently more reliable than systems built around a single central processor, because the propagation of low-level hardware and software errors is restricted by physical separation of processes and resources. In a single processor system, the reliability of the system is closely related to the reliability of the processor. However, when one site fails in distributed systems, it might be possible to finish the computing tasks in progress by using services of another site.

In a distributed system data can be stored at several sites. The proliferation of workstations and personal computers makes replication attractive for several reasons. For example, consider a transaction, being processed at one site, requires access to data stored at a different site. Both sites must be operational for the transaction to be completed in this situation. One way of improving the availability of data in a system with unreliable sites for such a transaction is to replicate the data and store it at multiple sites. A single site failure does not make replicated data inaccessible; the system can access the data in the presence of failures, even though some of the redundant copies are not available. In addition to improved availability, replication also enhances performance by placing the data closer to the process that requires it. For example, queries initiated at sites where the data are stored can be processed locally without incurring communication delays, and the workload of queries can be distributed to several sites where the subtasks of a query can be processed concurrently. Two major technological developments have made the implementation of replication techniques cost-effective: inexpensive processors and memory, making it cost-effective to develop large networks; and new communication technology, making it feasible to implement distributed algorithms with substantial communication requirements. However, the benefits of data replication must be balanced against the additional cost and complexities introduced for the synchronization of replicated data.

Assuming that the programs of individual transactions running in different sites are correct, the problem of maintaining a consistent database becomes a problem of synchronization and recovery. A major restriction in using replication is that replicated copies must behave like a single copy, i.e., *mutual consistency* of a replicated data must be preserved. By mutual consistency, we mean that all copies converge to the same value and would be identical if all update activities cease. The inherent communication delay between sites that store and maintain copies of a replicated data makes it impossible to ensure that all copies are identical at all times when updates are processed in the

This work was supported in part by the Office of Naval Research under contract number N00014-86-K-0245, by the Department of Energy under contract number KC-07-01-02, and by the Federal Systems Division of IBM Corporation under University Agreement WF-159679.

system. The principal goal of a synchronization mechanism for replicated data is to guarantee that all updates are applied to each copy in a way that assures the mutual consistency. Much research has been done to develop techniques for storing and retrieving data reliably through replication [BER84, BHA86, CHU85, GIF79, HER86, MIN82, PAR86, SON86, SON87].

Mutual consistency is not the only constraint a distributed system must satisfy. In a system where several users access and update data concurrently, operations from different transactions may need to be interleaved and allowed to operate concurrently on data for better system throughput. An interleaved execution of read and write operations of transactions may produce incorrect results. Concurrency control is the activity of coordinating concurrent accesses to the database in order to provide the same effect as if each request is executed in a serial fashion. The task of concurrency control in a distributed system is more complicated than that in a centralized system mainly because the information used to make scheduling decisions is itself distributed, and it must be managed properly to make correct decisions. Unless a correct concurrency control mechanism is used to restrict the methods of interleaving the operations from different transactions, update may be lost and incorrect retrieval would occur.

2. Classification of Synchronization Mechanisms

2.1. Pessimistic versus Optimistic

Synchronization mechanisms can be classified by several ways: pessimistic or optimistic, and syntactic or semantic. The first classification concerns the ordering of transactions, while the second classification concerns the type of information used in determining the correctness. Two synchronization mechanisms, two-phase locking and time-stamp ordering, can be considered pessimistic in their outlook. To avoid inconsistency, they synchronize operations of transactions first, and then perform the tasks of the transactions. They are called pessimistic because they operate under the pessimistic assumptions that the transaction conflicts will be frequent, and if an inconsistency can occur, it will occur. The mechanisms based on this approach avoid the overhead of undoing and redoing transactions that may be necessary if nonserializable executions were allowed. However, pessimistic mechanisms prevent several serializable executions of transactions in addition to all nonserializable executions because it is not possible to predict conflicts in advance. This reduces the degree of concurrency of the system, and lengthens the response time of transactions. Other drawbacks of the synchronization methods based on pessimistic approach include the deadlock problem in locking and unnecessary rollbacks in time-stamp ordering.

In optimistic approaches, the task of the transaction is performed temporarily, and then validated to see if a conflict has occurred. In this approach, the assumption is made that concurrent transactions will rarely conflict, and therefore synchronization at each transaction step is wasteful. Instead, transactions are given unrestricted read access to the database. However, writes are severely restricted. It is required that any transaction must go through two or three phases: a *read phase*, a *validation phase*, and a *write phase* (only for update transactions). During the read phase, transactions write only in local storage, and before those changes can be finalized in the database, a check is made during the validation phase. If a conflict has occurred, the transaction is restarted. If not, updates made by the transaction are installed in the database during the write phase. One drawback of the optimistic approach is that transactions may see an inconsistent database, which is not desirable in many applications. Another drawback is that performance degrades in case of frequent conflicts, since substantial computing and communication resources are wasted if the transactions are to be aborted in the validation phase. Furthermore, the system may become unstable because some transactions are caught in the loop, execute - conflict - abort - execute. Fortunately in many practical cases, conflict probability is very low, making an optimistic approach feasible. If the probability of conflicts is high, or the specification of the system requires that no transaction is allowed to see an inconsistent database state, pessimistic approach is appropriate to use.

An interesting difference between a pessimistic approach and an optimistic approach is that in a pessimistic approach, the serialization order of transactions is determined by some actions performed before the transactions, while in an optimistic approach, transactions are executed freely with the serialization order to be decided at the end. For example, in the two-phase locking mechanism, the order in which locks are requested or set determines the serialization order. In many time-stamp ordering mechanisms, it is decided by the time of the transaction arrival.

2.2. Syntactic versus Semantic

One-copy serializability has been used as the correctness criterion for transaction executions in distributed database systems with replicated data. It is a generalization of the concept of the *serializability* for non-replicated database systems. Syntactic approaches use one-copy serializability as the sole correctness criterion and check serializability by examining read-sets and write-sets of the executed transactions. Neither the semantics of the transactions nor the semantics of the data objects are used in ascertaining correctness. In semantic approaches, such information of transactions or data objects are used in checking the serializability, or in defining a different criterion of correctness for a given database. Although semantic approaches may result in higher degree of concurrency and system throughput, they might be less general than syntactic approaches, since they require more specific and application-dependent information of the system. For example, in log transformation technique [BLA85], transactions should be predefined, and semantic properties of pairs of transactions such as commutativity ($T_iT_j = T_jT_i$), should be declared in advance to avoid unnecessary roll-back and reexecution of transactions. If these properties are not predefined, this technique cannot be used in general.

3. Replication Management

Techniques for replicated data management can be characterized in terms of the control disciplines utilized by them: centralized or distributed. In centralized control techniques all requests on the replicated data objects pass through a single central control point, at which the requests are validated first, and then distributed to all the sites that maintain replicated copies. In distributed control approach, the responsibility for validating requests and actual execution of the requests on the replicated copies is distributed among the collection of sites. Centralized control techniques seem attractive because it is relatively easy to detect and resolve conflicts between requests. The primary disadvantage of such techniques is the vulnerability of the system; database activity must be suspended whenever the central control point is inaccessible due to either the failure of the site where the control point resides or network partitioning. Distributed control techniques achieve higher reliability than centralized control techniques in this regard, in principle at least. The most crucial problem in distributed control techniques is that it is nontrivial to design a correct distributed mechanism which preserves the database consistency. In the following, we review some of the replication control mechanisms that appeared in the literature.

Two most commonly used techniques for managing replicated data in distributed database systems are voting and special copy. In voting-based schemes, each copy has some number of votes, and a predetermined number of vote is necessary to perform a desired operation. In the schemes using the notion of special copy, availability of a special copy (or any copy) enables an operation on the data object.

3.1. Quorum Consensus Approach

Majority consensus

The first voting approach proposed was the majority consensus algorithm[THO79]. It assumes that the database is fully replicated at all sites. Each copy of the data carries the time-stamp of the last transaction which updated it. A read and a write are two basic operations allowed on data objects. A read returns the value and the time-stamp associated with the queried data object. The update procedure is decomposed into several steps for determining the new values to be written, and then applying updates which successfully collected majority of votes during the synchronization step. A transaction involving update operations cannot be committed unless a majority of sites consent, while queries can be processed locally. Time-stamp ordering technique is used both in the voting procedure and in the application of the accepted updates. In the synchronization step, a site compares the time-stamp of each data object with the corresponding time-stamp of its local copy. If the update request is based on an obsolete value, it is rejected. Otherwise, based on the priority of the request with other conflicting update requests, it gets either an OK vote or a PASS vote. The database consistency is maintained because the consenting majority subsets for any two accepted update requests must have at least one site in common, and the voting rule prevents a site from voting two OK's to two conflicting update requests. Mutual consistency is assured by the update application rule which guarantees that recent updates supersede older updates, and a reliable message system which guarantees message delivery even if the receiving site is not accessible.

Algorithms based on majority consensus attain a high level of robustness, in that only a majority of the sites need to be operational for transactions with update operations to complete. However, it may happen that a query sees an obsolete state of the database. This may be undesirable for some applications. If the algorithm is modified in order to guarantee consistent query execution, component failures would significantly reduce the extent to which database activity can continue.

Weighted voting

A weighted voting scheme was proposed by Gifford in [GIF79] as a generalization of the majority consensus algorithm of [THO79]. Each copy of a replicated data object is assigned some number of votes that can be used in gathering a quorum in order to execute an operation on the data. A transaction must collect a read quorum of r votes to read a data object, and a write quorum of w votes to write a data object. Quorums for each data object must satisfy the following two conditions:
(1) $r + w$ exceeds the total number of votes assigned to the data object.
(2) w exceeds the half of the total number of votes.

A transaction that writes to a data object reads the version numbers from a write quorum, generates a version number higher than any it has observed, and stores the new version at a possibly different write quorum. The first condition ensures that there exists a nonnull intersection between every read quorum and every write quorum, and hence guarantees that at least one copy that is most up-to-date is included in a read quorum. Here, version numbers are used instead of time-stamps to identify the most recent copy. To ensure that each new version number generated in such a method is greater than its predecessor, two write quorums (one used for finding version numbers and the other for writing the new version) must intersect. The second condition satisfies this constraint. However, this constraint reduces the number of distinct quorum assignments because each write operation requires majority.

Varying the weight of a vote can be used to reflect the needed accessibility level of a replicated copy. For example, weight of a vote can reflect the frequency of the use of each replicated copy. Suppose that there are four sites in the system, and the user submits the transaction to the site 1 and 2 most often, sometimes to site 3, but never uses site 4. Assigning $r = w = 3$ and the replicated copy at sites 1 and 2 a vote of 2, site 3 a vote of 1, and site 4 a vote of 0 would reflect this usage pattern of the user. Note that by changing r and w, the read accessibility and the write accessibility of the data object can be adjusted; read accessibility can be given a higher priority by making r smaller. However, reading a data object is still expensive in this scheme, since a read quorum of copies must be read in this scheme, whereas reading a single copy suffices in the voting scheme of [THO79].

Garcia-Molina and Barbara [BAR85, GAR85] have proposed a different interpretation of voting based on set theory, and have studied fault-tolerance characteristics such as the vulnerability provided by the quorum consensus approach.

Abstract data types

Weighted voting has been extended to replicated abstract data types, exploiting the semantic information of data types such as replicated queues and directories [HER86]. Each data object is viewed as an instance of an abstract data type, and for each abstract data type, a set of operations supported by the data object of that type is defined. For example, two operations are defined for a FIFO (first-in, first-out) queue: Enqueue (append new element at the end of the queue) and Dequeue (remove an element from the head of the queue and return its value). Data objects of type FIFO queue can only be accessed through these two operations.

A quorum is divided into two parts: an *initial quorum* and a *final quorum*. Conceptually, they are similar to a read quorum and a write quorum for simple data objects that support only read and write operations. An operation is executed by reading from an initial quorum of the replicated data object, performing a local computation to determine the value, and writing the determined value at a final quorum of the data object. Either the initial or final quorum may be empty. For example, a final quorum of read operation in Gifford's method is empty.

One of the ideas used in [HER86] is that quorums associated with each operation defined by an abstract data type can be different for different operations. For example, in a FIFO queue type data object, each Dequeue and Enqueue operation influences the value returned by a later Dequeue operation. Therefore the final quorums of both operation must have a non-null intersection with the initial quorum of Dequeue operation. On the other hand, neither Enqueue nor Dequeue operations influence later Enqueue operation. Hence the initial quorum of Enqueue operation needs not intersect with the final quorum of either operations.

One interesting change in [HER86] is that logs and time-stamps are used instead of versions and version numbers. A logical time-stamp is associated with each *event,* which is a pair of operation and its response. One advantage of using logical time-stamp is that write operations require half as many messages, since there is no need to ascertain the current version number. The state of a replicated data object is not represented by a collection of copies; instead each site maintains a partial log of the operations that have been applied to the data object. A *log* (or *history)* is a sequence of entries, each consisting of a logical time-stamp and an event. The log entries are partially replicated among a set of replicated copies. For example, the history for some queue Q might contain the following operations:

Q: {(Enqueue, 2, 30), (Enqueue, 4, 32), (Dequeue, 36), (Enqueue, 1, 37)}

This sequence of operations results in a queue containing two elements with values 4 and 1. Each copy of a data object stores a subsequence of the object's complete history. A subsequence of the history stored at each copy might be incomplete owing to failures preventing the receipt of certain operations at other copy sites, or because of communication delays. However, it is straightforward to merge subhistories of several copies into a single, more complete subhistory. The log-based approach places fewer constraints on quorum intersection than the version-based approach, and therefore a wider range of quorum choices and availability trade-offs are provided. This additional flexibility arises because logs permit a finer grain of control over versions. A version of a data object is either current or obsolete. If current, then the version contains the entire state of the data object; if obsolete, then it has no information about the current state of the data object. Any operation that modifies the data object's state must modify the current version, because versions that have diverged cannot be reconciled. By contrast, the system may be able to merge divergent logs. These mechanisms along with type-specific properties of abstract data types provide effective replication control method by reducing the quorum intersection constraints and increasing the range of realizable availability properties.

Missing update

Eager and Sevcik [EAG83] proposed a replication technique that does not require the intersection of all read and write quorums. This technique is based on the observation that requiring a read quorum for data objects in the read-set of a transaction significantly degrades performance, and it is not necessary when there are no failures. It is necessary to ensure correctness only when there are failures. In this technique, a transaction executes differently, depending on whether sites are up or down. In normal mode when all sites are up, a transaction reads from any copy and writes to all copies. In partitioned mode when any site is down, transactions use quorum consensus to read and write a majority of copies. Consistency is preserved by ensuring that transactions executed under partitioned-mode are serialized after transactions executed under normal-mode. When a transaction cannot update a copy because of failure, the corresponding update is said to be *missing*. A transaction becomes aware of the missing update when its precommit is unsuccessful. It is required that the transaction that awares missing update must pass its missing update information to any other transactions which may depend on it. All the transactions that receive missing update information are under a similar requirement to transfer to those which depend on them. The result of this transfer of information is that a transaction T_i which depends on T_1, T_2, ..., T_n becomes aware of all unapplied missing updates of which T_1, ..., T_n are aware. Once a missing update has been applied, the pertinent missing update information will be discarded.

One shortcoming of this method is that once a transactions has entered partitioned mode, it may be difficult to restore the system back to normal mode. Once missing update information has been created, transactions that receive such information will be restarted in partitioned mode, causing missing update information to propagate throughout the system. Eventually, the likelihood that a transaction will be able to execute to completion without awaring any missing update may become quite small. Once the partition is rejoined, all the missing update must be carried out, and the missing update information must be located and discarded. A special bookkeeping is necessary to perform this rejoining process correctly. The missing update information of a transaction cannot be discarded until all the missing updates associated with that transaction and with all previously committed transactions have been carried out. The difficulty in restoring the system back to normal mode stems from the fact that it is not a local process; since missing update information created by a transaction can be propagated to an arbitrary extent, the number of sites whose cooperation is necessary cannot be bounded in advance.

3.2. Special Copy Approach

Primary copy

In the *primary copy* method[ALS76], each data object is associated with a known primary site, also called as *master site*, to which all updates in the system for that data object are first directed. Distributed INGRES [STO79] follows this approach. Different data objects may have different primary sites. Basically, updates can be executed only if the primary copy of a data object is available. Updates are sent to non-primary copies on the commitment of the update transaction. The main advantage of this method is its simplicity. Its main drawback is its vulnerability to failures of primary copy sites and to network partitioning.

The basic scheme of primary copy approach can be made more reliable by making all non-primary copies as *backups*. A current value of the data object can be found at primary copy site and all backup copy sites as well. There is a predefined linear ordering of all copies, and any site can access the list of up-sites and find the particular copy which is the lowest in the ordering among those at sites in the up-sites list. This is the current primary copy of the data object. A read operation can be performed by reading the value of the current primary copy; a write operation is performed by writing to the current primary copy and any available backups. However, it should be noted that this approach works well only when site failures are clearly distinguishable from network failures; if it is uncertain whether the primary failed or the network failed, the assumption must be that the network failed and no new primary should be used.

When the network is partitioned, sites in different partitions would disagree on the identity of the primary copy. Hence, it is not enough to read from a copy that considers itself to be the primary. An inconsistency would result if more than one partition was allowed to update a given data object independently. One way to avoid this situation is to allow a primary copy to exist only if a majority of copies exists at sites in the up-sites list. Otherwise, the data object in question is considered to be not available. However, a problem arises in the case where there are exactly two copies of the data object. In this situation, any network partition will make both copies inaccessible. The *witness* scheme proposed in [PAR86] can be used in such a case. A witness is a virtual copy that is used for determining a majority, with no data attached to it. Multiple witnesses can also be used in voting-based replication methods in which they can vote like conventional copies, testifying about the current state of the replicated data object. Although all such techniques can make the primary copy approach more robust, they may bring the complexity to the system, spoiling the simplicity of the original approach.

Available copy

The *available copy* scheme[BER84] is a descendent of the primary copy algorithm. In this scheme, the system is dynamically reconfigured by removing failed sites and integrating recovered sites with the operational sites. There is no primary copy of a data objects; all copies are treated equally. It is based on *read-one/write-all* strategy, in which transactions may read from any copy, and must write to all available copies. The scheme maintains a *directory* which lists the copies of the data object that are available to use. It is assumed that if a transaction operates on a data object, a copy of the directory is available at the site running the transaction. When a transaction reads a data object, it consults the directory and reads from some copy listed there. When a transaction writes a data object, it consults the directory and writes all copies. The scheme executes special *status transactions* to keep directories up-to-date as sites fail and recover. One important feature of this scheme is the improved availability of data objects; a transaction can operate on a data object so long as one or more copies are available. One of its drawbacks is that only a limited class of failures can be handled; the scheme does not handle arbitrary failures and network partitions. A similar replication control scheme proposed in [BHA86] uses session numbers instead of directories.

True-copy token

In the *true-copy token* scheme[MIN82], a replicated data object is represented by a collection of copies. Copies that reflect the current state of the data object are called *true-copies*. Each data object has a token associated with it, permitting the bearer to access the data object. In the event of network partition, only the group containing the token will be able to access the data object. There are two kinds of true-copies: a unique *exclusive copy* used for both read and write operations, and multiple *shared copies* used only for reading. The set of true copies can be reconfigured to permit operations on a local copy of the data object: the exclusive true-copy can be reconfigured into multiple shared copies, and the set of shared copies can be reconfigured into a single exclusive copy. A true-copy is marked by a *true-copy token*, which also indicates whether the copy is shared or exclusive. Two types of locks, shared and exclusive, are used over the true-copies to realize consistent transaction processing. If all true-copies of a data object are lost, the data object cannot be accessed by transactions. A technique for regenerating true-copy tokens that have been destroyed by system failures is described in [MIN82]. However, it is difficult to detect reliably whether the token is lost or not, and to select an appropriate site for the regeneration of the token, avoiding concurrent regeneration. Furthermore, if site failures cannot be distinguished from network partitions, the failure of a site containing a true-copy token would limit reconfiguration. If an exclusive copy becomes unavailable, the data object can neither be read nor written; if a shared copy becomes unavailable, the data object can be read but cannot be reconfigured for writing. In this regard, the true-copy token method enhances performance of the system rather than availability of replicated data objects.

An interesting variation of token-based replication control scheme has been proposed in [SON86, SON87]. In this scheme, there is no distinction between exclusive true-copy token and shared true-copy token. A token designates a read-write copy, and there are predetermined number of tokens for each data object. Among multiple copies of a replicated data object, only token copies can be used to update the value of the data object. Non-token copies are updated during the commit procedure of the update transaction. It achieves higher availability of data objects than the true-copy token scheme in [MIN82] because a data object can be accessed and updated even if some token copies are unavailable.

Token-based methods can be viewed as a replication control approach that lies between the quorum consensus approach and the primary copy approach. It is different from the quorum consensus method in that it does not require a quorum to operate on a data object; availability of a single token copy is sufficient. It is different from the primary copy method in that the unavailability of a single token copy does not necessarily make the data object unavailable.

Exclusive-writer

The *exclusive-writer protocol* (EWP) and *exclusive-writer protocol with locking option* (EWL) proposed in [CHU85] are optimistic counterparts of the primary copy approach. In EWP, every data object has a special transaction referred to as its exclusive-writer (EW) which is solely responsible for resolving all conflicts involving the data object. A non-EW transaction sends to the EW of the data object an update-request message. It knows that its update request has been accepted when it receives an update message corresponding to its update request. Conflicts are detected by the use of the *update sequence numbers* (SN). An SN is attached to each copy of a data object, and update requests include the SN of the copy read by the requesting transaction. Updates are written in the order of their SN's. The EW accepts an update request only if the SN in the update request is identical to the SN of its copy; otherwise, a conflict occurred, and the transaction is discarded. If an EW accepts an update request, it increments the SN of its copy, and distribute the update with the new SN. EWP preserves the mutual consistency and the serializability of successful transactions whose updates are distributed.

Since EWP does not use locks, no deadlocks can occur due to shared data access. Update requests that are not accepted by the EW are simply discarded, and hence EWP has no transaction restarts. For many applications, discarding update requests is not acceptable. However, it may be suitable for distributed real-time control systems, e.g., a distributed processing system for radar tracking data updating, in which occasionally discarding an update request has little overall effect. EWL is the same as EWP with the exception that it ensures full serializability, and in case of a conflict, it can switch to the primary copy algorithm.

4. Concluding Remarks

Replication is the key factor in making distributed systems more reliable than single-processor systems. However, if replication is used without proper synchronization mechanisms, consistency of the system might be violated. In this regard, the copies of each data object must behave like a single copy from the standpoint of the database correctness.

In this paper we have surveyed some of the techniques proposed to achieve the consistency of a distributed database system with replicated data. We have classified different replication management techniques by the underlying mechanisms they use. Replication control techniques need to be considered from three different viewpoints: correctness, robustness, and performance. It may be rather straightforward to achieve only the requirement of the database consistency, by not allowing the access to the replicated data object when any copy of the data object is unavailable due to a failure. However, this approach is not acceptable for many applications of distributed database systems, in which most of the critical data are replicated to improve the system availability. Robustness of replication control techniques is very important, since one of the primary reasons of replication is to have the data available in failure situations. In many distributed systems, network partitioning should be considered in addition to site failures. For a survey on replication control in partitioned networks, interested readers are referred to [DAV85]. Performance characteristic must reflect the cost associated with the technique such as the complexity of the technique as well as the overhead incurred by the technique. Each technique has a unique characteristic, and hence it is the designer's responsibility to determine which technique would be the best choice for the system. While most of the work in the area of replication control in distributed database systems has been concentrated on the design and correctness proof of new algorithms, more research in the performance evaluation is needed. Although it would not be possible to adequately discuss or cite all important work in this area, the presentation given here should help interested readers to understand the issues related to replication control in a distributed environment, and some of the possible methods to handle those problems.

REFERENCES

ALS76	Alsberg, P.A., Day, J.D., A Principle for Resilient Sharing of Distributed Resources, Proc. Second International Conf. on Software Engineering, Oct. 1976, pp 562-570.
BAR85	Barbara, D., and Garcia-Molina, H., Evaluating Vote Assignments with A Probabilistic Metric, Digest of Papers FTCS-15: Fifteenth International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, June 1985, pp 72-77.
BER84	Bernstein, P., Goodman, N., An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases, ACM Trans. on Database Systems, Dec. 1984, pp 596-615.
BHA86	Bhargava, B., Ruan, Z., Site Recovery in Replicated Distributed Database Systems, Proc. 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 621-627..
BLA85	Blaustein, B. and Kaufman, C., Updating Replicated Data during Communications Failures, Proc. of 11th VLDB, 1985, pp 1-10.
CHU85	Chu, W. W. and Hellerstein, J., The Exclusive-Writer Approach to Updating Replicated Files in Distributed processing Systems, IEEE Trans. on Computers, June 1985, pp 489-500.
DAV85	Davidson, S., Garcia-Molina, H., and Skeen, D., Consistency in Partitioned Networks, ACM Computing Surveys, Vol. 17, No. 3, Sept. 1985, pp 341-370.
EAG83	Eager, D. and Sevcik, K., Achieving Robustness in Distributed Database Operations, ACM Trans. on Database Systems, September 1983, pp 354-381.
GAR85	Garcia-Molina, H. and Barbara, D., How to Assign Votes in a Distributed System, Journal of ACM, Oct. 1985, pp 841-860.
GIF79	Gifford, D., Weighted Voting for Replicated Data, Operating Systems Review 13, December 1979, pp 150-162.
HER86	Herlihy, M., A Quorum-Consensus Replication Method for Abstract Data Types, ACM Trans. on Computer Systems, February 1986, pp 32-53.
MIN82	Minoura, T. and Wiederhold, G., Resilient Extended True-Copy Token Scheme for a Distributed Database System, IEEE Trans. on Software Engineering, May 1982, pp 173-189.
PAR86	Paris, J., Voting with Witnesses: A Consistency Scheme for Replicated Files, Proc. 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 532-539.
SKE85	Skeen, D., Determining The Last Process to Fail, ACM Trans. on Computer Systems, Feb. 1985, pp 15-30.
SON86	Son, S. H., Agrawala, A. K., A Token-Based Resiliency Control Scheme in Replicated Database Systems, Proc. Fifth Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, January 1986, pp 199-206.
SON87	Son, S. H., On Multiversion Replication Control in Distributed Systems, Computer Systems Science and Engineering, Vol. 2, No. 2, April 1987, pp 76-84.
STO79	Stonebraker, M., Concurrency Control and Consistency of Multiple Copies in Distributed INGRES, IEEE Trans. on Software Engineering, May 1979, pp 188-194.
THO79	Thomas, R. H., A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases, ACM Trans. on Database Systems, June 1979, pp 180- 209.