SYNCHRONIZATION OF REPLICATED DATA IN DISTRIBUTED SYSTEMS

Sang Hyuk Son

Computer Science Report No. TR-86-22
October 17, 1986

This work was partially supported by the Office of Naval Research under contract no. N00014-86-K-0245 to the Department of Computer Science, University of Virginia, Charlottesville, VA.

ABSTRACT

Replication is the key factor in improving the availability of data in distributed systems. Replicated data is stored redundantly at multiple sites so that it can be used by the user even when some of the copies are not available due to site failures. A major restriction in 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 based on the maintenance of multiple versions of data objects have been studied by many researchers in order to increase the degree of concurrency and to reduce the possibility of transaction rollback. Improved availability and increased degree of concurrency would result if multiversion concurrency control techniques can be used with replication control methods in distributed systems. This paper surveys synchronization methods for multiversion distributed systems with replicated data objects that have been appeared in the literature. Different synchronization methods are grouped by underlying mechanisms they use in ordering the operations, and their advantages and disadvantages are discussed. A theory that is used in analyzing the correctness of synchronization methods is reviewed, and a proving method based on the serializability theory is demonstrated using a simple synchronization scheme.

1. Introduction

A distributed system consists of multiple autonomous computer systems (called *sites*) that are connected via a communication network. One of the advantages of distributed systems over centralized systems is that replicated copies of critical data can be stored redundantly at multiple sites. Two major technological developments have made the implementation of replication techniques cost-effective: inexpensive processors and memory, making it possible to develop large networks, and new communication technology, making it feasible to implement distributed algorithms with substantial communication requirements. The main goal of having replicated data is to enhance the availability of data. By storing data at multiple sites, 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 increases the reliability of data by reconstructing accidently destroyed copy from other copies. Replication can enhance performance by allowing queries initiated at sites where the data are stored to be processed locally without incurring communication delays, and by distributing the workload of queries to several sites where the subtasks of a query can be processed concurrently. These benefits of replicated data must be balanced against the additional cost and complexities introduced for replication control.

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 system. The principal goal of a replication control mechanism is to guarantee that all updates are applied to copies of replicated data in a way that assures the mutual consistency. Considerable research effort has been focused in recent years in developing techniques for storing and retrieving data reliably through replication, and a number of replication control schemes have been proposed. They include [BER84, CHU85, EAG83, HER86, JOS86, MIN82, NOE85, SON86].

We assume that the system provides the ability to make the execution of user requests *atomic*. By atomic, we mean that the execution of requests are performed in an all-or-nothing fashion: it either succeeds completely (commits), or it has no effect (aborts). Atomic requests are called *transactions* [GRA81]. A transaction issues read and write operations.

Mutual consistency is not the only constraint a distributed system must satisfy. In a system where several users concurrently access and update data, operations from different transactions may need to be interleaved and allowed to operate concurrently on data for better throughput of the system. In such a case, 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 effect that 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 exercised to restrict the methods of interleaving the operations from different transactions, anomalous situations such as lost update, incorrect retrieval, and inconsistent update would occur[BER81].

Concurrency control mechanisms can be categorized by several ways: pessimistic or optimistic[KUN81], two-phase locking or time-stamp ordering[BER81], single version or multiple versions[CHA85]. Pessimistic mechanisms synchronize operations of transactions first, and then perform the tasks of the transactions. They are called pessimistic because they assume that the transaction conflicts will be frequent. 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.

In optimistic approach, the task of the transaction is performed temporarily, and then go through the validation phase to see if a conflict has occurred[KUN81]. 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 due to conflicts. Furthermore, the system may become unstable because some transactions are caught in the loop, execute - conflict - abort - execute. 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 the pessimistic approach, the serialization order of transactions is determined by some actions performed in the beginning of the transactions, while in an optimistic approach, transactions are executed freely with the serialization order to be decided at the end. For example, in 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.

A number of concurrency control schemes proposed are based on the maintenance of multiple versions of data objects[BAY80, BER83, CHA85, REE78, SON85, STE81]. The principal objective of using multiple versions is to increase the degree of concurrency and to reduce the possibility of transaction rollback by providing transactions with a succession of views of data objects. One of the reasons for rejecting a transaction is that its operations cannot be serviced by the system. For example, a read operation has to be rejected if the value of data object it was supposed to read has already been overwritten by some other transaction. Such rejections can be avoided by keeping old versions of each data object so that an appropriate old value can be given to a tardy read operation. In a system with multiple versions of data, each write operation on a data object produces a new version of it instead of overwriting it. Hence, for each read operation, the system selects an

appropriate version to read, enjoying the flexibility in controlling the order of read and write operations.

We survey here the methods and techniques for achieving global consistency in distributed systems with replicated data objects, discussing the advantages and disadvantages of each method. The paper is organized as follows. Section 2 presents a model of computation used in the paper. Section 3 describes different replication control methods. Section 4 presents concurrency control techniques using multiple versions of data. Section 5 presents a multiversion synchronization scheme for replicated data objects. Section 6 reviews serializability theory that provides necessary conditions for the correctness of concurrency control algorithms. Section 7 concludes the paper.

2. Model of Computation

To understand different replication and concurrency control methods, we need to understand first the organization of distributed systems and how the methods fit into an overall transaction processing in distributed systems. In this section we present a simplified model of a distributed system, and describe how the system processes transactions.

2.1. Distributed System Environment

A distributed system is a collection of sites, each of which maintains a local database system. Each site is able to process *local transactions*, those transactions that access data only in that single site. In addition, a site may participate in the execution of *global transactions*, those transactions that access data in several sites. The execution of a global transaction requires communication among participating sites. Each site runs processes called *transaction managers* which supervise interactions between users and the system, and *data managers* which manages the local database. Since a transaction must be executed atomically in any circumstances, one of the most important functions of the transaction manager of a distributed system is to ensure that the execution of global transactions preserves atomicity.

The smallest unit of data accessible to the user is called *data object*. Data objects do not correspond directly to real database items; they are an abstraction. In a particular system, the data objects might be files, pages, records, etc. In distributed systems with replicated data objects, a logical data object is represented by a set of one or more replicated physical data objects. Two types of *logical operations* that can be performed on a logical data object are read and write. A logical operation requested at one site is implemented by executing *physical operations* on one or more copies of physical data objects in question.

Users of a distributed system see only the logical database, a collection of logical data objects. They expect the system to behave as if it executes transactions one at a time to the logical one-copy database, even though the actual database system executes many transactions at a time using several replicated physical data objects. Even though physical data objects can be read or written independently, the system must provide the user a view of a single-copy of each logical data object.

2.2. Transactions

Users interact with the system by submitting transactions. Each transaction represents a complete and correct computation, i.e., if a transaction is executed alone on an initially consistent database, it would terminate in a finite time and produce correct results, leaving the database consistent. A transaction consists of different types of operations such as read, write, and local computations. Read and write operations are used to access data objects, and local computations are used to determine the value of the data object for a write operation. Algorithms for replication and concurrency control pay no attention to the local computations; they make scheduling decisions on the basis of the data objects a transaction reads and writes. The *read set* of a transaction T is defined as the set of data objects that T reads. Similarly, the set of data objects that T writes is called the *write set* of T.

The transaction managers that have been involved in the execution of a transaction are called the *participants* of the transaction. The coordinator is one of the participants which initiates and terminates the transaction by controlling all other participants.

A *private workspace* which functions as a temporary buffer for values read from and written into the database is provided for each transaction. When a transaction makes a read access, it first checks the workspace and reads from the workspace if the data is there. When a transaction makes a write access, it writes only into the workspace, not storing the new value in the database at this time. The output of the transaction is also kept in the workspace until the transaction is ready to commit. The use of the private workspace allows aborting the transaction at any time before the initiation of committing the transaction, by simply discarding its workspace without any undoing operations.

When a transaction commits, updates in the workspace must be written into the database. All participants must commit unanimously, implying that the updates performed by the transaction are made visible to other transactions in an "all or none" fashion. One of the most well-known implementation of atomic commitment is a procedure called *two-phase commit* [GRA79], which works as the following:

In the first phase the coordinator sends "start transaction" messages to all the participants. Each participant individually votes either to commit (YES) or abort (NO) the transaction according to the result of the subtransaction it has executed. A transaction is said *precommitted* at a participating site when it has completed the execution of the transaction at the site and has sent a YES vote to the coordinator. If a failure occurs during the first phase, consistency of the database is not violated, since none of the transaction's updates have yet been applied to the actual database.

In the second phase the coordinator collects all the votes and makes a decision. If all votes were YES, the coordinator sends "commit" messages to the participants. If the coordinator had received one or more NO votes, it sends "abort" messages to the participants.

3. Replication Management

Replication management techniques 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 the failure of the site where the control point resides. Distributed control techniques achieve higher reliability than centralized control techniques in this regard, in principle at least. The most crucial problem in distributed control schemes is that it is nontrivial to design a correct mechanism which preserves the database consistency. In this section we review several techniques for replication management, central and distributed, appeared in the literature.

Two major techniques for managing replicated data objects in distributed systems are voting and special copy. In voting-based schemes, each copy has an equal number or different number of votes, and a predetermined number of vote is necessary to perform a desired operation. The database can either be fully replicated or partially replicated. In the schemes using the notion of special copy, availability of a special copy (or any copy) enables an operation on the data object. We first review the replication management techniques based on voting.

3.1. Quorum Consensus Approach

In the majority consensus approach[THO79], a transaction can commit only if a majority of sites consent. 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. The algorithm employs the time-stamp ordering technique both in the voting procedure and in the application of the accepted requests to the replicated copies. A transaction involving update operations cannot be committed unless a majority of sites consent, while queries can be processed locally. Since there must be at least one site that is a member of two different majority,

no two conflicting user requests can be executed and committed. Mutual consistency is assured by assuming a reliable message system which guarantees message delivery even if the receiving site is not accessible.

A scheme for concurrency control based on majority consensus has been also proposed in [BRE82]. The scheme requires the mutual consistency of only a majority of copies; a minority of copies may be obsolete. Operations on replicated data objects can be executed only when a transaction obtains necessary locks on a set of copies containing at least a majority. The most up-to-date copies are distinguished from outdated copies by the use of version numbers. Once the most up-to-date copy is determined, it is used to update all other copies in the majority set. Mutual consistency is not strictly imposed since copies not in the majority set may not be updated.

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, as noted by Thomas, it may happen that a query sees an inconsistent 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.

A weighted voting scheme is described in [GIF79]. As in [BRE82], version numbers are used instead of time-stamps. Each copy is assigned a fixed number of votes that can be used in gathering a quorum in order to execute an operation on the data. 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. 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. This constraint reduces the number of distinct quorum assignments because each write operation requires majority. Since quorums are determined such that a read quorum and a write quorum must intersect,

at least one copy that is most up-to-date is guaranteed to be included in a read quorum.

Weighted voting has been extended to replicated abstract data types, including replicated queues and directories[HER86]. A quorum is divided into two parts: an *initial quorum* and a *final quorum*. 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 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* 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. It has been shown in [HER86] that 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.

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.

Eager and Sevcik [EAG83] proposed a replication technique that does not require the intersection of all read and write quorums. 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*. The 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_1 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

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. Update requests will be sent to non-primary copies either before or 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.

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 known linear ordering of all copies. 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 primary copy and any available backups. However, it should be noted that if 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 independently update a given data object. 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.

A complicated descendent of primary copy algorithms is the *available copy* scheme[BER84]. 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 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.

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*. 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]. 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. 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.

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 *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.

EWP is simple to implement, and has no deadlocks and transaction restarts. However, it is not a general purpose protocol since it only ensures a limited form of serializability. It may be suitable for distributed real-time control systems, e.g., a distributed processing system for radar tracking data updating. 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. Multiversion Concurrency Control

Several interesting concurrency control algorithms that exploit multiple versions of data object in distributed systems have been proposed, and some of them have been implemented[SVO80, CHA82, DUB82]. One of the reasons of studying multiversion control methods is that in many distributed systems, old versions of data objects are maintained to support transaction recovery and system recovery. Therefore, it is very natural to ask if concurrency can be improved by reading old versions. For example, if the transaction T_i has been given permission to write a data object and the transaction T_j asked to read the same data object, then it is possible to give T_j the old version instead of making T_j wait until T_i is finished. However, an appropriate control must be exercised in doing so, otherwise the

database consistency might be violated. In the example above, assume that T_i has written a new value for two data objects X and Y, and T_j has read the old value of X. T_j wants to read Y also. If T_j gets the new version of Y created by T_i, there is no serial execution of T_i and T_j having the same effect because in reading the old version of X, T_j sees the database in a state before the execution of T_i, and in reading the new version of Y, T_j sees the database in a state after the execution of T_i.

Multiversion concurrency control methods can be grouped by underlying mechanisms they use in ordering the operations of transactions: time-stamping and locking. In time-stamping, each transaction is assigned a unique time-stamp, and operations are processed in their time-stamp order. Read operations on data objects are translated to *version-read* operations by identifying an appropriate versions for them. In most schemes based on locking, each data object is associated with different kinds of locks, and transaction execution follows two-phase locking paradigm. We first review multiversion concurrency control methods based on time-stamping.

4.1. Time-stamp Ordering

A time-stamp is a number that is assigned to a transaction when initiated, and is kept by the transaction. There are two primary methods for generating unique time-stamps, one centralized and one distributed. In a centralized method, a global clock is maintained at a single site which distributes the time-stamp. The site can use a logical counter or its own local clock for this purpose. In a distributed method, each site generates a unique local time-stamp. A globally unique time-stamp can be obtained by concatenating the local time-stamp with the identifier of the site. In this method, a time-stamp consists of a pair (t,n) where t is the value of the local clock of the site, and n is the unique identifier of the site. In order to ensure that no local clock gets far ahead or behind another clock, local clocks are synchronized through message communication in the following way:

(1) Each site increments its local clock by one between any two successive events.

(2) Every message contains the current clock value of the sender site.

(3) On receiving a message with a clock value t which is greater than the current local clock value, the local clock is set to the value t+1.

A detailed discussion of time-stamp generation can be found in [LAM78, THO79].

The important properties of time-stamp are: no two transactions have the same time-stamp, and only a finite number of transactions can have a time-stamp less than that of a given transaction. For any two time-stamps TS1=(t_1, n_1) and TS2=(t_2, n_2), TS1 is smaller than TS2 if either ($t_1 < t_2$) or (t_1=t_2 and $n_1 < n_2$). If a transaction T_1 has a smaller time-stamp than T_2, we say that T_1 is the *older* transaction and T_2 the *younger*.

One of the earliest multiversion concurrency control scheme based on time-stamp ordering is Reed's scheme[REE78]. Besides time-stamping and multiversions, the scheme also uses the notions such as *tokens*, and *possibilities*. In order to achieve atomicity of transactions in spite of failures, each write operation creates a tentative version, called a *token*, in the appropriate sequence of versions of a data object. The set of tokens created by a transaction is called a *possibility*, since it represent a new state of the system whose existence is conditioned by the successful completion of the creating transaction. The state of a possibility is represented by a record called the *commit record*, which records the state of the possibility and the time of its time-out. When initially created, a commit record is in the "unknown" state. Committing the possibility is executed by changing the state atomically to the "committed" state. Aborting the possibility is accomplished by changing the state to "aborted". A time-out is used to avoid possible blockings of shared data or deadlocks by detecting that a transaction will not complete due to any errors or failures. If the state is still "unknown" after the time-out elapses, the state will be changed to "aborted".

Each version carries the time-stamp of the transaction that wrote it. Each read and write operation carries the time-stamp of the transaction that issued it, and read and write operations to shared data object behave such that a read operation returns the value of the data object written by the latest write operation of a transaction that precedes the read

operation. Operations are processed with no constraint on the real-time order, e.g., first-come-first-served. A read operation with time-stamp t on a data object may arrive at and be processed by the site maintaining the data object before a write operation whose time-stamp is less than t. If such a tardy write operation arrives at a site, it must not be executed, for that would cause the value previously seen by the read operation to be incorrect.

In order to keep track of the reads that have been executed, each version records the maximum time-stamp of the read operations that have accessed the version. A tardy write operation is detected by the fact that its time-stamp is in the interval between the time-stamp of the write operation creating a version and the maximum read time-stamp of that version. To execute a read operation, a proper version is first selected by comparing the time-stamp of the read operation to the time-stamps of versions. Then a check is performed to see if the version is committed, by testing the state of the possibility.

One of the drawbacks of this scheme is that not only a write operation but also a read operation by a transaction causes the time-stamp associated with the data object to be updated, leaving a permanent trace in the system. This results in increased processing time and storage requirement. Another drawbacks are related to the need for modifying the version chain pointers when adding a token, and to the vulnerability of the commit record.

4.2. Locking

A multiversion concurrency control scheme based on locking has been proposed in [CHA85]. In this scheme, conflicts between read-only transactions and update transactions are eliminated by making use of the semantic knowledge that read-only transactions perform no updates, thereby reducing synchronization delays. It is necessary a transaction informs the system whether it is a read-only transaction or an update transaction when it begins. Update transactions synchronize by using two-phase locking. Read-only transactions set no locks; each read-only transaction sees a consistent snapshot of the database by selecting appropriate versions of data objects to read. This snapshot is defined by the *completed transaction list* (CTL). In essence, a read-only transaction sees the state of the database at the

time of its initiation, minus changes made by in-progress update transactions. In order to determine the snapshot that a read-only transaction should see, unique identifiers are assigned to update transactions, and each read-only transaction carries a copy of the CTL that is in effect at the time of its initiation.

The difficulty in determining a consistent snapshot comes from a possible unordered delivery of commit messages such that there could be a transaction dependency that will not be reflected in the site's CTL. This difficulty can be overcome by obtaining a global CTL, which is made up of a number of local CTL's, representing completion information for transactions that have been initiated from a given site. For an update transaction, each participant includes its copy of the CTL in the precommit message to the coordinator. The coordinator merges the received CTL's with its own, and tags the union-CTL onto the commit message. For a read-only transaction, the coordinator requests the CTL from each of the retrieval sites, computes the union-CTL, and retransmits it. Each retrieval site then performs multiversion reading using the received union-CTL. This propagation of transaction completion information makes the CTL's at individual sites consistent.

The scheme achieves only *weak consistency* which assures that each read-only transaction sees a consistent view of the database. It is because the scheme ignores possible read-write conflicts, and hence it cannot satisfy the conventional serializability criterion. Read-only transactions can be guaranteed *strong consistency*, which assures the serializability of all the transactions, if real-only transactions are run using two-phase locking.

The major benefit of this scheme is that read-only transactions are guaranteed to run to completion without blocking, which is particularly useful when there is a mix of short update transactions and retrievals that require sequential scanning of large portions of the database. In addition to the storage cost for multiple versions and I/O cost for its maintenance, lock table management and a possible long processing to chase through a chain of versions to access the appropriate version for a read-only transaction are two major costs of the scheme. Maintenance and propagation of transaction completion information is another

source of overhead of the scheme.

Bayer et al. [BAY80] and Stearns and Rosenkrantz [STE81] have presented concurrency control algorithms that are based on locking. These algorithms maintain at most two versions of a data object: one committed version and another updated but not yet committed version. When a transaction updates a data object, the new version is *uncertified*, in the sense that it must be certified later through the commitment of the transaction. The value of the committed version of a data object is identified as the *before-value*, and the value of the uncertified version is identified as the *after-value*.

The basic working principles of the algorithms are as follows. A read operation is performed either on the before-value or on the after-value, and a write operation writes the after-value. When a transaction finishes executing, the transaction manager tries to certify all the versions it wrote. For each data object it wrote, a *commit lock* is set. A commit lock is necessary to assure that there will be no more transactions that read the before-value. The locking attempt succeeds if no other transaction already has a commit lock. A transaction commits when the following three conditions hold:

(1) A commit lock is set on each data object it wrote.

(2) Each version it read is committed.

(3) All transactions that read the before-value of a data object it wrote have been committed.

The algorithm in [STE81] avoids deadlock by setting two different conditions on active transactions based on their time-stamps and conflict resolution actions. The algorithm in [BAY80] uses the dependence graph instead of time-stamp. It is used in translating read operations into appropriate version-read operations such that read requests can always immediately be granted without the problems of deadlock and inconsistency. Bookkeeping is simplified in both algorithms by maintaining at most two versions of each data object. However, it reduces the concurrency achievable in the system, compared to systems maintaining multiple versions of data.

5. Multiversion Synchronization for Replicated Data

In order to show the methods that can be used in analyzing the correctness of synchronization mechanisms, a multiversion synchronization scheme for distributed systems with replicated data objects is sketched in this section. An outline of the correctness proof is presented in the next section after reviewing the serializability theory. A detailed description of a theory for analyzing the correctness of concurrency control algorithms for multiversion databases is presented in [BER83]. The scheme to be presented here is an extension of the token-based scheme proposed in [SON86], making use of Reed's multiversion time-stamp scheme in [REE83] for concurrency control.

Each logical data object has a predetermined number of tokens, and each token copy is the latest version of the data object. A token designates a read-write copy. Copies without tokens (read-only copies) must go through the *copy actualization phase*, if necessary, in order to satisfy the consistency constraints of the system. For read-only copies, each data object is a collection of versions.

We use $R_i(X)$ to denote a logical read operation on X issued by the transaction T_i. Similarly, $W_i(X)$ denotes a logical write operation on X by T_i. We use lower case letters to represent physical operations. Thus, $r_i(X)$ represents a physical read operation on X resulting from a logical operation $R_i(X)$, and $w_i(X)$ denotes a physical write operation on X resulting from $W_i(X)$. We denote versions of X by X_i, X_j,...., where the subscript is the identifier of the transaction that created the version.

To read a data object, the coordinator sends a read request to a read-only copy site at which it is translated into a version-read operation. Because updates of data objects occur at token sites first, it is possible that at some time instant, the latest version of a data object may not exist in a read-only copy. A copy of a data object X is said to be *actual* if the value of it reflects the latest update made to X. Each read operation $R_i(X)$ is translated into $r_i(X_k)$, where X_k is the latest version of X with the time-stamp $\leq$ time-stamp of T_i. If there exists a version of X with time-stamp $>$ time-stamp of T_i, then the value of X_k

is used for r_i. Otherwise, an Actualization Request Message (ARM) is sent to any available token site to actualize the read-only copy. At the token site, an ARM is treated as the same as a r_i(X).

Copy actualization is also used to actualize a token copy recovered from the crash. During the recovery of the site, the recovery manager of the site tries to update all the token copies at the site by sending ARMs to other available token sites. The recovered copy will be used for transaction processing only after it is successfully updated through the copy actualization procedure.

$W_i(X_i)$ is translated into $w_i(X_i)$ for all available token copies. $w_i(X_i)$ is rejected if the current version were X_j, and the time-stamp(T_i) < time-stamp(T_j), or there is a possibility that $r_k(X_j)$ is processed and time-stamp(T_j) < time-stamp(T_i) < time-stamp(T_k). Intuitively, w_i(X) is rejected if it would invalidate $r_k(X_j)$ for any k.

The two-phase commit protocol needs to be modified for this scheme such that the coordinator of a transaction commits when the following conditions are satisfied:

(R1) All the available token copies of each data object in the write-set have precommitted.

(R2) One copy of each data object in the read-set is available and has precommitted.

6. Consistency of Multiversion Systems

A concurrency control algorithm is said to be correct if the same state results as if the transactions were processed in a serial fashion. In distributed systems with multiversion, *one-serializability* (1-SR) has been used as the correctness criterion for transaction executions[BER83]. In this section, we review fundamental concepts associated with 1-SR, and show that the scheme sketched in the previous section guarantees the consistency of the system by satisfying the requirements of 1-SR.

6.1. Serializability of Transactions

A *log* is a model of execution of transactions, which captures the order in which read and write operations are executed. An *augmented log* is a log with an initial transaction that

writes to all data objects and a final transaction that reads from all data objects. Initial and final transactions are not the user transactions; they are used only to determine the initial and terminal state of the database. In the following, we consider only the augmented log. Two logs are said to be *equivalent* if each transaction sees the same state of the system and leaves the system in the same state in both logs. Equivalence relation can be expressed by *read-from* relations. Transaction T_j *reads-x-from* T_i if $R_j(X)$ is translated into $r_j(X_i)$, i.e., $w_i(X_i)$ *precedes* $r_j(X)$ and no $w_k(X_k)$ falls between these operations. Two logs are equivalent if and only if they have the same reads-from relationships.

A *serial log* ("serial schedule" in [ESW76]) is a totally ordered log such that the operations from different transactions are not interleaved. Serial logs result in poor performance since there is no concurrency at all. However, from the viewpoint of the correctness of concurrency control, each serial log represents a correct execution. A log is *serializable* if it is equivalent to a serial log. Since a serial log is correct, serializable logs are also correct.

Because any two conflicting operations cannot occur at the same same, there exists a *precedence* relation (denoted by $\rightarrow$) between any pair of conflicting operations. The precedence relation between transactions are defined as the following:

Definition: $T_1 \rightarrow T_2$ implies that for some data object X, $r_i(X) \rightarrow w_j(X_j)$, or $w_i(X_i) \rightarrow r_j(X)$, or $w_i(X_i) \rightarrow w_j(X_j)$.

Let L be a log over a set of transactions. The *serialization graph* for L, SG(L), is a directed graph whose nodes are transactions, and there is an edge from T_i to T_j ($i \neq j$) if $T_1 \rightarrow T_2$. Serializability theorem that provides a necessary condition for the correctness of a log can be stated as follows[BER81].

Theorem 1: If SG(L) is acyclic, then L is serializable.

A detailed formal treatment of serializability theory can be found in [PAP79].

In a multiversion system with replication, the criteria of correctness needs to be changed because users expect their transactions to behave as if there was only one copy for each data object.

A serial log L in a multiversion system is called *one-copy serial* (or 1-serial) if for all i, j, and X, if T_j reads X from T_i then i = j or T_i is the last transaction preceding T_j that writes into any version of X. A log is *one-copy serializable* (or 1-serializable) if it is equivalent to a 1-serial log. A log L over a set of transactions in a multiversion system is equivalent to a serial log in a single version system over the same transactions if and only if L is 1-serializable[BER83].

For a multiversion system the serialization graph is extended as follows: Given a log L and a data object X, a *version order* (denoted by <<) for X is any total order over all of the versions of X in L. A version order for L is defined as the union of the version orders for all data objects in L. The multiversion serialization graph, MVSG(L), is SG(L) with the following edges added: for each $r_k(X_j)$ and $w_i(X_i)$ in L, if k≠i and $X_i << X_j$ then include $T_i \rightarrow T_j$, else include $T_k \rightarrow T_i$. The main theorem on 1-serializability can be stated as follows[BER83].

Theorem 2: A log L is 1-serializable if an only if there exists a version order << such that MVSG(L) is acyclic.

6.2. Correctness Proof

A method that can be used in proving the correctness of multiversion concurrency control mechanisms is (1) to infer properties which all logs produced by the scheme will satisfy, and then (2) to show that these log properties imply 1-serializability. We demonstrate this method by showing the correctness of the scheme sketched in the previous section. The following lemma provides precedence relationships between read and write operations.

Lemma 3: For every $r_k(X_j)$ and $w_i(X_i)$, i≠j, either time-stamp(T_i) < time-stamp(T_j) or time-stamp(T_k) ≤ time-stamp(T_i)

Proof: Since all conflicting operations are executed in sequential order, there are two possibilities between $r_k(X_j)$ and $w_i(X_i)$: $r_k(X_j) \rightarrow w_i(X_i)$, or $w_i(X_i) \rightarrow r_k(X_j)$.

Case 1) $r_k(X_j) \rightarrow w_i(X_i)$

$w_i(X_i)$ is rejected if it would invalidate $r_k(X_j)$ for any k, i.e., if time-stamp(T_j) < time-stamp(T_i) < time-stamp(T_k). Therefore possible relationships among transactions can only be either time-stamp(T_i) < time-stamp(T_j) or time-stamp(T_k) ⩽ time-stamp(T_i).

Case 2) $w_i(X_i) \rightarrow r_k(X_j)$

Between two write operation $w_i(X_i)$ and $w_j(X_j)$, we have two possible orders: $w_i(X_i) \rightarrow w_j(X_j)$ or $w_j(X_j) \rightarrow w_i(X_i)$. $w_i(X_i) \rightarrow w_j(X_j)$ implies that time-stamp(T_i) < time-stamp(T_j) since w_j would be rejected otherwise. $w_j(X_j) \rightarrow w_i(X_i)$ implies that time-stamp(T_k) ⩽ time-stamp(T_i) since other orders would not satisfy both conditions of $w_j(X_j) \rightarrow w_i(X_i)$ and $w_i(X_i) \rightarrow r_k(X_j)$. □

The following theorem shows that any log satisfying these precedence relationships is 1-serializable.

Theorem 4: All logs produced by the scheme are 1-serializable.

Proof: Let L be a log produced by the scheme. We prove that there cannot be a cycle in MVSG(L) by showing that all edges are in time-stamp order of the transactions in L.

Let $T_i \rightarrow T_j$ be an edge of MVSG(L). This edge can correspond to a simple read X of T_j from the version written by T_i: $r_j(X_i)$. Because a read operation r_i(X) is translated into $r_i(X_j)$, where X_j is the version of X with largest time-stamp ⩽ time-stamp(T_i), $r_j(X_i)$ implies that time-stamp(T_i) ⩽ time-stamp(T_j). From the uniqueness property of the time-stamp, time-stamp(T_i) ≠ time-stamp(T_j) for all i and j, i≠j. Since an edge is not allowed for i=j in MVSG(L), we have time-stamp(T_i) < time-stamp(T_j).

Now consider an edge introduced in L by the relationships among $r_k(X_j)$, $w_j(X_j)$, and $w_i(X_i)$. We have two different possible version orders.

Case 1) $X_i << X_j$

By the MVSG(L) generation rule, the edge is $T_i \rightarrow T_j$. Since the scheme maintains the version order in time-stamp order, we have time-stamp(T_i) < time-stamp(T_j).

Case 2) $X_j << X_i$

By the MVSG(L) generation rule, the edge is $T_k \rightarrow T_i$. By Lemma 3, either time-stamp(T_i)

$<$ time-stamp(T_j) or time-stamp(T_k) $\leqslant$ time-stamp(T_i). The first possibility, time-stamp(T_i) $<$ time-stamp(T_j), is impossible because of the version order $X_j \ll X_i$. From the uniqueness property of the time-stamp, time-stamp(T_k) $\neq$ time-stamp(T_i), it must be the case that time-stamp(T_k) $<$ time-stamp(T_i).

We have shown that all possible edges in MVSG(L) are in time-stamp order. To have a cycle, there must be an edge $T_i \rightarrow T_j$ in MVSG(L) where time-stamp(T_j) $<$ time-stamp(T_i). Since time-stamps are totally ordered, we cannot have such a anti-chronological edge to build a cycle, and hence MVSG(L) is acyclic. By Theorem 2, L is 1-serializable. □

7. Concluding Remarks

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

Multiversions of data objects can increase the degree of concurrency by providing transactions with successive versions of data. The major cost in multiversion systems is the storage requirement. In addition to that, concurrency control using multiversions of data objects becomes more complicated because the system must consider the translation of operations into appropriate version-operations as well as the order they are processed.

In this paper we have surveyed techniques proposed to achieve consistency of the distributed system with replicated data objects. We have attempted to review different replication control techniques and synchronization techniques using multiple versions of data objects. We also have reviewed a theory for analyzing the correctness of concurrency control algorithms, and have demonstrated a way the theory can be used by proving the correctness of a simple synchronization scheme.

It would be impossible to adequately discuss or cite all important work in the area of synchronization in multiversion distributed systems with replicated data objects. However,

we hope that the presentation given here will help interested readers to understand the issues related in controlling replication and multiversions of data objects in distributed systems, 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.

BAY80 Bayer, R., Heller, H., and Reiser, A., Parallelism and Recovery in Database Systems, ACM Trans. on Database Systems, June 1980, pp 139-156.

BER81 Bernstein, P., Goodman N., Concurrency Control in Distributed Database Systems, ACM Computing Surveys, June 1981, pp 185-222.

BER83 Bernstein, P., Goodman N., Multiversion Concurrency Control - Theory and Algorithms, ACM Trans. on Database Systems, Dec. 1983, pp 465-483.

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..

BRE82 Breitwieser, H. and Leszak, M., A Distributed Transaction Processing Protocol Based on Majority Consensus, Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, Aug. 1982.

CHA82 Chan, A., Fox, S., Lin, W., Nori, A., and Ries, D., The Implementation of An Integrated Concurrency Control and Recovery Scheme, ACM SIGMOD Conf. Proc. 1982, pp 184-191.

CHA85 Chan, A., Gray, R., Implementing Distributed Read-Only Transactions, IEEE Trans. on Software Engineering, Feb. 1985, pp 205-212.

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.

DUB82 Dubourdieu, D. J., Implementation of Distributed Transactions, Proc. Berkeley Workshop on Distributed Data Management and Computer Networks, 1982, pp 81-94.

EAG83 Eager, D. and Sevcik, K., Achieving Robustness in Distributed Database Operations, ACM Trans. on Database Systems, September 1983, pp 354-381.

ESW76 Eswaran, K.P. et al, The Notion of Consistency and Predicate Locks in a Database System, Communications of ACM 19, Nov. 1976, pp 624-633.

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.

GOO83 Goodman, N., Skeen, D. and et al., A Recovery Algorithm for a Distributed Database System, Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 1983, pp 8-15.

GRA79 Gray, J. N., Notes on Database Operating Systems, Operating Systems: An Advance Course, Springer-Verlag, N.Y., 1979, pp 393-481.

GRA81 Gray, J. N., The transaction Concept: Virtues and Limitations, Proc. 7th International Conference on Very Large Databases, September 1981, pp 144-154.

HAM80 Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, December 1980, pp 431-466.

HER86 Herlihy, M., A Quorum-Consensus Replication Method for Abstract Data Types, ACM Trans. on Computer Systems, February 1986, pp 32-53.

JOS86 Joseph, T. A., Birman, K. P., Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems, ACM Trans. on Computer Systems, February 1986, pp 54-70.

KUN81 Kung, H. T. and Robinson, J. T., On Optimistic Methods for Concurrency, ACM Trans. on Database Systems, June 1981, pp 213-226.

LAM78 Lamport, L., Time, Clocks and Ordering of Events in Distributed Systems, Communications of ACM, July 1978, pp 558-565.

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.

NOE85 Noe, J. D., Proudfoot, A. B., Pu, C., Replication in Distributed Systems: The Eden Experience, Technical Report TR-85-08-06, Dept. of Computer Science, Univ. of Washington, September 1985.

PAP79 Papadimitriou, C. H., The Serializability of Concurrent Database Updates, *Journal of ACM*, October 1979, pp 631-653.

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.

REE78 Reed, D., Naming and Synchronization in a Decentralized Computer System, Tech. Rep. TR-205, MIT, September 1978.

REE83 Reed, D., Implementing Atomic Actions on Decentralized Data, ACM Trans. on Computer Systems, Feb. 1983, pp 3-23.

SKE85 Skeen, D., Determining The Last Process to Fail, ACM Trans. on Computer Systems, Feb. 1985, pp 15-30.

SON85 Son, S. H., Agrawala, A. K., A Multiversion Concurrency Control Scheme for Replicated Databases, Tech. Rep. TR-1564, Dept. of Computer Science, University of Maryland, October 1985

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.

STE81 Stearns R. E., Rosenkrantz, D. J., Distributed Database Concurrency Controls Using Before-Values, Proc. ACM SIGMOD Conf. 1981, pp 74-83.

STO79 Stonebraker, M., Concurrency Control and Consistency of Multiple Copies in Distributed INGRES, IEEE Trans. on Software Engineering, May 1979, pp 188-194.

SVO80 Svobodoba, L., Management of Object Histories in the SWALLOW repository, Tech. Rep. TR-243, Laboratory for Computer Science, MIT, July 1980.

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.