USING REPLICATION FOR HIGH PERFORMANCE DATABASE SUPPORT IN DISTRIBUTED REAL-TIME SYSTEMS

Sang Hyuk Son

Computer Science Report No. TR-87-17
August 18, 1987

Using Replication for High Performance Database Support in Distributed Real-Time Systems

Sang Hyuk Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

Considerable research effort has been concentrated to the problem of developing techniques for achieving high availability of critical data in distributed real-time systems. One approach is to use replication. Replicated data is stored redundantly at multiple sites so that it can be used even if some of the copies are not available due to failures. This paper presents an algorithm for maintaining consistency and improving the performance of database with replicated data in distributed real-time systems. The semantic information of read-only transactions is exploited for improved efficiency, and a multiversion technique is used to increase the degree of concurrency. Related issues including the consistency of the states seen by transactions, version management, and recovery of replicated data in distributed systems are discussed.

Index Terms: distributed system, replication, read-only transaction, consistency, multiversion.

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.

1. Introduction

A distributed system consists of multiple autonomous computer systems (called *sites)* that are connected via a communication network. Since the physical separation of sites ensures the independent failure modes of sites and limits the propagation of errors throughout the system, distributed systems must be able to continue to operate correctly despite of component failures. However, as the size of a distributed system increases, so does the probability that one or more of its components will fail. Thus, distributed systems must be fault tolerant to component failures to achieve a desired level of reliability and availability. Asserting that the system will continue to operate correctly if less than a certain number of failure occurs is a guarantee independent of the reliability of the sites that make up the system. It is a measure of the fault tolerance supported by the system architecture, in contrast to fault tolerance achieved by using reliable components.

Considerable research effort has been concentrated in recent years to the problem of developing techniques for achieving high availability of critical data in distributed systems. An obvious approach to improve availability is to keep replicated copies of such data at different sites so that the system can access the data even if some of the copies are not available due to failures. 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 user requests initiated at sites where the data are stored to be processed locally without incurring communication delays, and by distributing the workload of user requests to several sites where the subtasks of a user request can be processed concurrently. These benefits of replication must be seen in the light of the additional cost and complexities introduced by replication control.

A major restriction of 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.

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 user requests may need to be interleaved and allowed to operate concurrently on data for higher throughput of the system. Concurrency control is the activity of coordinating concurrent accesses to the system 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.

A number of concurrency control schemes proposed are based on the maintenance of multiple versions of data objects[BAY80, BER83, CHA85, REE83 SON86, STE81]. The objective of using multiple versions is to increase the degree of concurrency and to reduce the possibility of rejection of user requests by providing a succession of views of data objects. One of the reasons for rejecting a user request 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 user request. 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.

A read-only transaction is a user request that does not modify the state of the database. A read-only transaction can be used to take a checkpoint of the database for recovering from subsequent failures, or to check the consistency of the database, or simply to retrieve the information from the database. Many applications of distributed databases for real-time systems can be characterized by a dominance of read-only transactions. Since read-only transactions are still transactions, they can be processed using the algorithms for arbitrary transactions. However, it is possible to use special processing algorithms for read-

only transactions in order to improve efficiency, resulting in high performance. With this approach, the specialized transaction processing algorithm can take advantage of the semantic information that no data will be modified by the transaction.

In this paper, we explore this idea of read-only transaction processing, and present a synchronization algorithm for read-only transactions in distributed environments. The algorithm is based on the idea of maintaining multiple versions of necessary data objects in the system, and requires read-only transactions to be identified to the system before they begin execution. By preventing interference between read-only transactions and other update transactions, the algorithm guarantees that read-only transactions will be successfully completed. In addition, the replication method used in the algorithm masks failures as long as one or more copies remain available.

There are several problems that must be solved by an algorithm that uses multiple versions. For example, selection of old versions for a given read-only transaction must ensure the consistency of the state seen by the transaction. In addition, the need to save old versions for read-only transactions introduces a storage management problem, i.e., methods to determine which version is no longer needed so that it can be discarded. In this paper, we focus our attention on these problems.

In the next section we present the basic concepts that are needed for this paper. Section 3 describes the execution of logical operations by corresponding physical operations. Section 4 describes our synchronization algorithm for replicated data. Section 5 presents two recovery procedures that can be used for replicated data objects, and Section 6 discusses the availability of replicated data. Section 7 concludes the paper.

2. Basic Concepts

A distributed database is a collection of data objects. Each data object has a name and is represented by a set of one or more replicated copies. Each copy of a given data object is stored at a different site of the system. The value for a given data object at different sites should be the same. However, due to the update activity, the values may be temporarily different. In addition to data objects, a distributed database

has a collection of *consistency constraints*. A consistency constraint is a predicate defined on the database which describes the relationships that must hold among the data objects and their values [ESW76].

Users interact with the database 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 control and synchronization pay no attention to the local computations; they make scheduling decisions on the basis of the data objects a transaction reads and writes.

When a transaction commits, all the updates it made must be written permanently 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 techniques for the atomic commitment is a protocol called *two-phase commit* [SKE81], 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 the transaction by sending precommit message or to abort it by sending abort message, according to the result of the subtransaction it has executed. 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 written into the database. In the second phase the coordinator collects all the votes and makes a decision. If all votes were precommit, the coordinator sends "commit" messages to the participants. If the coordinator had received one or more abort messages, it sends "abort" messages to the participants.

The standard correctness requirement for transactions is serializability. It means that the concurrent execution of a group of transactions is equivalent to some serial execution of the same group of transac-

tions. For read-only transactions, correctness requirements can be divided into two independent classes: the currency requirement and the consistency requirement.

The currency requirement specifies what update transactions should be reflected by the data read. There are several ways in which the currency requirement can be specified; we are interested in the following two:

(1) Fixed-time requirement: A read-only transaction T requires data as they existed at a given time *t*. This means that the data read by the transaction must reflect the modifications of all update transactions committed in the system before *t*.

(2) Latest-time requirement: A read-only transaction T requires data it reads reflect at least all update transactions committed before T is started, i.e., T requires most up-to-date data available.

The consistency requirement specifies the degree of consistency needed by the read-only transaction. A read-only transaction may have one of the following requirements:

(1) Internal consistency: It only requires that the values read by each read-only transaction satisfy the invariants (consistency constraints) of the database.

(2) Weak consistency: It requires that the values read by each read-only transaction be the result of a serial execution of some subset of the update transactions committed. Weak consistency is at least as strong a requirement as internal consistency, because the result of a serial execution of update transactions always satisfies consistency constraints.

(3) Strong consistency: It requires that all update transactions together with all other read-only transactions that require strong consistency, must be serializable as a group. Strong consistency requirement is equivalent to serializability requirement for processing of arbitrary transactions.

We make a few comments concerning the currency and consistency requirements. First, it might seem that the internal consistency requirement is too weak to be useful. However, a read-only transaction with only internal consistency requirement is very simple and efficient to process, and at least one proposed algorithm [FIS82] does not satisfy any stronger consistency requirement. Second, it is easy to see

that strong consistency is a stronger requirement than weak consistency, as shown by the following example. Suppose we have two update transactions, T_1 and T_2, two read-only transactions, T_3 and T_4, and two data objects, X and Y, stored at two sites A and B. Assume that the initial values of both X and Y were 0 before the execution of any transactions. Now consider the following execution sequence:

T_3 reads 0 from X at A.

T_1 writes 1 into X at A.

T_4 reads 1 from X at A.

T_4 reads 0 from Y at B.

T_2 writes 1 into Y at B.

T_3 reads 1 from Y at B.

The values read by T_3 are the result of a serial execution of $T_2<T_3<T_1$, while the values read by T_4 are the result of a serial execution of $T_1<T_4<T_2$. Both of them are valid serialization order, and thus, the execution is weakly consistent. However, there is no single serial execution of all four transactions, so the execution is not serializable. In other words, both read-only transactions see valid serialization orders of updates, but they see different orders.

Clearly, strong consistency is preferable to weak consistency. However, as in the case of internal consistency, it can be cheaper to ensure weak consistency than to ensure strong consistency. For the applications that can tolerate a weaker requirement, the potential performance gain could be significant.

Finally, one might wonder why fixed-time requirement is interesting, since most read-only transactions may require information about the latest database state. However, there are situations that the user is interested in looking at the database as it existed at a given time. For an example of a fixed-time read-only transaction, consider the case of a general in the army making a decision by looking at the database showing the current position of the enemy. The general may be interested in looking at the position of the enemy of few hour ago or few days ago, in order to figure out the purpose of their moving. A read-only

transaction of a given fixed-time will provide the general with the desired results.

3. Execution of Logical Operations

In our algorithm, we use the notion of tokens to support a fault-tolerant distributed database in increasing both the availability of data and the degree of concurrency, without incurring too much storage and processing overhead. Each data object has a predetermined number of tokens. Tokens are used to designate a read-write copy, and a token copy is a single version representing the latest value of the data object. The site which has a token copy of a data object is called a *token site*, with respect to the data object.

Multiversions are stored and managed only at read-only copy sites. For read-only copies, each data object is a collection of consecutive versions. A read-only transaction does not necessarily read the latest committed version of a data object. The particular old version that a read-only transaction has to read is determined by the time-stamp of the read-only transaction (for the latest-time requirement) or by the given time (for the fixed-time requirement). The time-stamp is assigned to a read-only transaction when it begins, while the time-stamp for an update transaction is determined as it commits. When a read-only transaction with time-stamp T attempts to read a data object, the version of the data object with the largest time-stamp less than T is selected as the value to be returned by the read operation.

To simplify the presentation in this paper, we use a simple model of data objects, with only read and write operations, instead of considering an abstracted data model. As discussed in [HER86, WEI84], greater concurrency among update transactions can be achieved if more semantic information about the specification of each abstract data object is used. The algorithm presented in this paper can be easily adapted to use this kind of semantic information of data objects.

In this paper, we do not consider Byzantine type of failures. When a site fails, it simply stops running (fail-stop). When the failed site recovers, the fact that it has failed is recognized, and a recovery procedure is initiated. We assume that site failures are detectable by other sites. This can be achieved either by network protocols or by high-level time-out mechanisms in the application layer.

We assume that update transactions use two-phase locking [ESW76], with exclusive locks used for write operations, and shared locks for read operations. Lock requests are made only to token copies, and there is no locks associated with read-only copies. In addition, update transactions use the two-phase commit protocol and stable storage [LAM81] to achieve fault-tolerance to site failures. When a new version is created, it is created at all copy sites, including read-only copy site. However, any new versions are not accessible to other transactions until they are finalized through the two-phase commit protocol. Upon receiving the commit message from the coordinator, new versions of data objects created by the transaction replace the current versions at the token sites, while they are attached to the multiple versions at read-only sites.

Operations invoked by update transactions are processed using ordinary two-phase locking: when an update transaction invokes a read operation on a data object, it waits until it can lock the data object in shared mode. When an update transaction invokes a write operation, it locks the data object in exclusive mode, and then creates a new version. If the transaction later aborts, the newly created version will be discarded. Our algorithm follows the *read-one/write-all-available* paradigm [BHA86] in which a read lock request succeeds if at least one of the token copies can be locked in shared mode, and a write lock request fails if at least one of the available token copies cannot be locked in exclusive mode. In a straightforward implementation of a write operation in this paradigm, the value to be written is broadcast to all sites where a copy of the data object resides. A physical write operation occurs at each copy site, and then a confirmation message has to be returned to the site where the write operation was requested. The write operation is considered completed only when all the confirmation messages are returned. This solution is unsatisfactory because every write operation incurs waiting for responses before the next operation of the transaction can proceed. In the next section, we present an algorithm that permits an operation after a write to proceed as in a nonreplicated system, with the physical write operations being executed concurrently at other copy sites.

4. The Algorithm

As noted above, our algorithm combines time-stamp ordering and locking. To generate time-stamps for update transactions, a time-stamp is maintained for each data object. The time-stamp for data object X represents the maximum of the time-stamps of update transactions that have accessed X and committed, and the time-stamps of read-only transactions that have accessed X. Time-stamps for update transactions are generated during the commit phase as follows:

In the first phase of the two-phase commit protocol, each participant attaches to the precommit message the maximum time-stamps of all data objects that it accessed. Upon receiving precommit messages from the participants, the coordinator chooses a unique time-stamp greater than all the time-stamps received. This is the time-stamp for the transaction. Then, in the second phase of the commit protocol, this time-stamp is broadcast to all participants (piggybacked on the commit message). Each participant, upon receiving this message, updates the time-stamp of each data object to the maximum of its current value and the received time-stamp, and releases any locks held by the transaction. Any version written by the transaction is marked with the time-stamp of the transaction.

When a participant searches for the maximum time-stamp to attach to a precommit message, read-only copies are also included in the set of copies for the search. By including the time-stamps of read-only copies in determining the time-stamp for an update transaction, the system can ensure that any potential conflicts between read-only transactions and the current update transaction are resolved in the correct order of their time-stamps.

Time-stamp assignment for read-only transactions with latest-time requirement is quite different from that for update transactions. When a read-only transaction begins, the coordinator sends messages to the participants telling them the data objects the transaction needs to read. When a participant receives such a request, it checks the current time-stamp of each data object at the site, and sends the maximum time-stamps among them to the coordinator. Each data object accessed by a read-only transaction in this way records the pair of the identifier of that transaction and the current time-stamp it reported. After

receiving responses from all participants, the coordinator chooses a unique time-stamp greater than all the responses. The time-stamp recorded for the read-only transaction at each object is thus a lower bound on the time-stamp of the transaction, and it will be used in making a decision to discard or retain versions of the data object. For a fixed-time read-only transactions, time-stamp is provided by the user, and hence the system needs not bother to assign a new time-stamp for it.

When a read-only transaction with time-stamp T invokes a read operation on a data object, the participant chooses the version of the object with the largest time-stamp less than T. This invocation of read operation is nothing but sending the time-stamp T to the participants, since each participant already knows which data object to read. If T is larger than the current time-stamp of the data object, it will be updated as T. This will force update transactions that commit later to choose time-stamps larger than T, ensuring that the version selected for the read-only transaction does not change.

It is easy to show that the above algorithm ensures strong consistency. The mechanism for generating time-stamps for update transactions ensures that any conflicting update transactions are ordered according to their time-stamps, and hence they are serializable in the time-stamp order. Read-only transactions then read versions of data objects consistent with all transactions executing in their time-stamp order.

To achieve the high performance by reducing the cost of write operations in our algorithm, the level of synchronization between write operation and its physical implementation can be relaxed by allowing physical write operations to be completed by the commit time of the transaction. A write operation is considered completed when the required update messages are sent. This eliminates the delay caused by waiting for confirmation messages before the next operation can proceed.

To this point we have assumed that all versions are retained forever. We now discuss how versions can be discarded when they are not needed by read-only transactions. Recall that each data object keeps track of the read-only transactions that have accessed the data object, along with a lower bound on the time-stamp chosen by each transaction. Data objects can use the following rule to decide which versions

to keep and which to discard.

Rule for retention:

A version with time-stamp T must be retained if

(1) there is no version with time-stamp greater than T (i.e., current version), or

(2) there is a version with time-stamp T' > T, and there is an active read-only transaction whose time-stamp might be between T and T'.

By having a read-only transaction inform data objects when it completes, versions of data objects that are no longer needed can be discarded. This process of informing data objects that a read-only transaction has completed need not be performed synchronously with the commit of the transaction. It imposes some overhead on the system, but the overhead can be reduced by piggybacking information on existing messages, or by sending messages when the system load is low.

When a read-only transaction sends a read request to an object, the read-only site effectively agrees to retain the current version and any later versions, until it knows which of those versions is needed by the read-only transaction. When the read-only site finds out the time-stamp chosen by the transaction, it can tell exactly which version the transaction needs to read. At that point any versions that were retained only because the read-only transaction might have needed them can be discarded. By minimizing the time during which only a lower bound on the transaction's time-stamp is known, the system can reduce the storage needed for maintaining versions. One simple way of doing this is to have each read-only transaction broadcast its time-stamp to all read-only sites when it chooses the time-stamp.

The version management described above is effective at minimizing the amount of storage needed for versions. For example, unlike the "version pool" scheme in [CHA85], it is not necessary to discard a version that is needed by an active read-only transaction because the buffer space is being used by a version that no transaction wants to read. However, ensuring that each read-only site knows which versions are needed at any point in time has an associated cost; a read-only transaction cannot begin execution

until it has chosen a time-stamp, a process that requires communicating with all data objects it needs to access.

Because the time-stamp for a fixed-time read-only transaction is determined by the user, the number of versions that needs to be retained to process fixed-time read-only transactions cannot be bounded as in the case for latest-time read-only transactions. In order to process all the potential fixed-time read-only transactions, the system must maintain all the versions created up to the present, which may require huge amount of storage. There are several alternatives to keep a history instead of saving all the versions created for each data object. One of the simplest and efficient alternative would be to keep a log of all the update transactions. A transaction log is a record of all the transactions and the updates they performed. Fixed-time read-only transactions can be processed by examining the log in reverse chronological order until the desired version of the data object can be reconstructed. Since fixed-time read-only transactions must examine the log, their execution depends on the availability of transaction log, and their execution speed would be slower than that of latest-time read-only transactions. One important advantage of the transaction log mechanism is that in many systems the log is required anyway for crash recovery. Thus, in these systems, keeping the log for fixed-time read-only transactions represents no real overhead.

5. Recovery of Replicated Data Objects

Sites of a distributed system may fail and recover from time to time during the life-time of the system. When a failed site recovers, the consistency of the entire system may be threatened if proper recovery mechanisms are not exercised. A task of integrating a site into the rest of the system when the site recovers from a failure is called the *site recovery*. Site recovery must perform the recovery of non-replicated as well as replicated data objects in order to bring the system into a consistent state. In this section we discuss only the recovery of replicated data objects. A more detailed discussion on site recovery is given in [SON86b].

There are two main approaches to this problem. The first is to perform all missed updates in a correct order at the recovering site. Multiple message spoolers used in SDD-1 [HAM80] is one practical

solution using this approach. We do not discuss this approach further in this paper because (1) it is difficult to determine a correct schedule for all the missed operations, and (2) it is not suitable for systems in which some sites may not be operational for a long period of time.

The second approach is to use other replicated copies by reading the current values at operational sites and refresh out-of-date values at the recovering site. An advantage of this approach is that the recovering site can start normal operation on the data objects as soon as they are refreshed, without waiting for the completion of the recovery procedure for other data objects, resulting in improved availability of the system. Algorithms using this approach have been studied in [BHA86]. In this section we present two recovery procedures that belong to the class of the second approach, and discuss the trade-offs between two recovery procedures.

5.1. Updating Directories

The first recovery procedure is based on updating *directories*. Each data object is associated with a directory that keeps the status of each copy, i.e., the availability of each copy of the data object. User transactions read the directories of the data objects in its read-set and write-set to determine the participants of the transaction. Directories are replicated at each copy site and updated by the processing of Update Directory Messages (UDM) which contains information of the status change of other sites. A UDM is used to include a copy as well as to exclude a copy.

To exclude a copy, a UDM is broadcast by the network protocol which detects a site failure. In this case, a UDM contains only the identifier of the crashed site. On receiving a UDM of this type, the recovery manager of each site checks directories of all the data objects at the site and removes the site from the available copy lists.

From the viewpoint of data objects, there are two types of the system failures: a *partial failure* and a *total failure*. They are distinguished by the availability of token copies of a logical data object. In a partial failure, one or more token copies are available; in a total failure, none of them is available. To recover from a total failure, the site which failed last must be determined, because this copy has the latest version

of the data object. This task can be achieved by executing an algorithm similar to those proposed in [GOO83, SKE85].

To recover from a partial failure, the recovering token copy must be updated to the current value of the data object before being included in the list of available copies. A token-copy cannot be included in the directory while the data object is being used. The recovery manager of the site generates a special transaction which requests a write operation on the data object. When the request is granted, the transaction updates the directory by including the identifier of the recovered site into the list of available copies, instead of updating the value of the data object. A read-only copy can be included in the available copy list simply by appending the identifier of the recovered site, without being updated on recovery.

5.2. Updating Site Status

The second recovery procedure is based on keeping track of the status of sites instead of maintaining the status for each data object. In this approach, each site maintains the *site status table*, in which each site is represented in one of three distinguishable states: *up*, *down*, and *recovering*. A site is down if no activity is going on at the site. A site is up if it executes user transactions normally. A site is recovering if it performs recovery actions but no user transactions.

When a site recovers from a failure, the first action it should take is to change its own state to recovering state so that no user transactions can be accepted. It then performs local recovery for non-replicated data objects. Finally, it marks all replicated copies at the site unreadable. If there is a method to find out the replicated copies that have actually missed updates since the site failed, only those copies are marked unreadable. The site then becomes up, and broadcasts its state change to all operational sites. During normal operation after the site becomes up, unreadable mark of a replicated copy will be removed by a write operation of a committed transaction, or by a read operation which is performed through the copy actualization procedure for bringing the copy up-to-date.

5.3. Trade-offs in Recovery

There is a trade-off between the transaction processing time during normal operation and the time required to perform recovery procedures. In the second approach (maintaining site status), the participants of a transaction is not determined simply by looking at the directories as in the first approach. Each transaction should read the local copy of the site status table prior to any other operations. The transaction can use this table in deciding which sites to be included (up sites) and which not (down sites) in the participant list. This requires the transaction processing time longer than that in the first approach during normal operation.

The second approach performs better than the first approach for the storage requirement and the cost of recovery processing. According to the second approach, the storage necessary for maintaining the availability information of data objects can be reduced by the factor of the product of the number of replicated data objects and the number of copies used in the replication. Consider an extreme case in which almost all data objects are replicated at each site. In the first approach, the number of updates is proportional to the number of replicated data objects when a site status changes, while only a single table needs updating in the second approach. Although a straightforward method to reduce the number of updates is possible in this case, the first approach remains more expensive than the second approach in these regards.

6. Availability of Replicated Data Objects

One of the important properties of our algorithm is the flexibility. By manipulating the number of tokens for each data object, a system administrator can alter the performance and the reliability characteristics of the system.

There are two interesting extremes out of a spectrum of possible token numbers: a situation where all copies are token copies, and a situation where there is only one token copy for each data object. In the first case, no multiversions are maintained, resulting in a less storage requirement and a simpler algorithm. However, read-only transactions cannot be processed effectively in that case.

The second case is similar to primary copy algorithms. As pointed out in [GIF79], primary copy algorithms are inflexible even though they are relatively simple. It is simple in the sense that a transaction needs only one copy to update a data object. However, primary site algorithms are not reliable in that transactions cannot be executed if the token site is crashed. Although we can make the system robust through the regeneration of the token when the token is lost, the detection and the regeneration of a unique token may bring the complexity to the system, spoiling the simplicity of the original scheme.

No matter how many copies exist, it is always possible to enter a state in which no copy is available. We call a data object state *unavailable* if any update operation cannot be performed by any transaction. Since unavailable states of data objects reduce the system availability (i.e., some transactions must be rejected because they cannot update unavailable data objects), it is obviously desirable to reduce the probability of unavailable states.

For a given number of copies, we can evaluate the probability that the data object is available, given the failure probabilities of each component of the system. We assume that for each site and link, the probability of being operative is known. These probabilities represent the expected fraction of time each component is able to provide the service correctly. We refer to these probabilities as the *component availabilities*.

For the simplicity of the discussion, we restrict ourselves to the homogeneous system where all sites have the same availability p and all links availability s. We first consider the case in which the links are perfectly reliable, i.e., s=1. Let q=1-p. For a data object with n copies including m token copies, $m \leq n$, let the probability that the data object is available be P_n^m. If there is only one token copy, i.e., m=1, then P_n^1=p. The probability in general is

$$P_n^m = \sum_{k=1}^{m} \binom{m}{k} p^k q^{m-k}$$

It is clear from this equation that the availability of a data object increases as m increases. However, the marginal increase of the availability of a data object achieved by adding one more token copy decreases as m increases. Let δ_n^m be the marginal increase of the availability of a data object by increasing the

number of token copies from m to m+1:

$$\delta_n^m = P_n^{m+1} - P_n^m$$

The marginal increase of the availability is the gain we can achieve by making one more token copy of the data object. Figure 1 shows the behavior of δ_n^m as a function of m. As shown in the figure, δ_n^m decreases very abruptly as m approaches to n. This implies that from the availability viewpoint, the cost of adding more token copies cannot be justified by the increase of availability, once we achieve a desirable availability of a data object by certain number of token copies.

The network never becomes partitioned if all the links are perfect, i.e., s=1. If we relax the assumption of perfect links, network partitioning should be considered. To adapt our scheme to the system where network partitioning can occur, we need to change the conditions for the coordinator to make a commit decision as the following:

(C1) Majority of the token sites of each data object in the write-set have precommitted.

(C2) One copy of each data object in the read-set belongs to the majority of token copies and is precommitted.

In order to make this modified algorithm to work, the number of token copies must be stored with each data object. This modified algorithm is able to handle the network partitioning, but reduces the availability of data objects of the original algorithm because now the system cannot process transactions if majority of the token sites are not available (the original algorithm is able to process update transactions with one token copy available).

Network topology plays a critical role in determining the availability of data objects when partitioning can occur. The expression for P_n^m in such situations is the same as the majority of voters connected in a voting scheme, because the system should have at least a majority of token sites connected. We do not derive the expression for P_n^m in this paper. The probability that the system is operative in different network topology using voting mechanism has been studied in [BAR85], which shows that for the same set of component availability, a fully connected topology provides higher probability of operative system

than Ethernet or ring topologies. However, full connectivity is expensive to support, and it may not be feasible to have a full connectivity in a system with a large number of sites. For a system of an arbitrary graph topology with possibly different component availabilities, we may need good heuristics to determine the best token assignments to achieve a desirable system availability.

7. Concluding Remarks

Replication is the key factor in making distributed systems more reliable than centralized systems. However, if replication is used without proper synchronization 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.

We have presented a synchronization algorithm for distributed real-time systems with replicated data. It reduces the time required to execute physical write operations when updates are to be made on replicated data objects, by relaxing the level of synchronization between write operations on data objects and physical write operations on copies of them. At the same time, the consistency of replicated data is not violated, and the atomicity of transactions is maintained. The algorithm extends the notion of primary copies such that an update transaction can be executed provided at least one token copy of each data object in the write set is available. The number of tokens for each data object can be used as a tuning parameter to adjust the robustness of the system. The algorithm also exploits the multiple versions of a data object and the semantic information of read-only transactions in achieving improved system performance. Multiple versions are maintained only at the read-only copy sites, hence the storage requirement is reduced in comparison to other multiversion mechanisms[REE83, CHA85].

Reliability does not come for free. There is a cost associated with the replication of data: storage requirement and complicated control in synchronization. For appropriate management of multiple versions, some communication cost is inevitable to inform data objects about activities of read-only transactions. There is also a cost associated with maintaining the data structures for keeping track of versions and time-stamps. In many real-time applications of distributed databases, however, this cost of replication is

justifiable. Further work is clearly needed to develop alternative approaches for maintaining multiversions and exploiting semantic information of read-only transactions, and to study performance of different approaches.

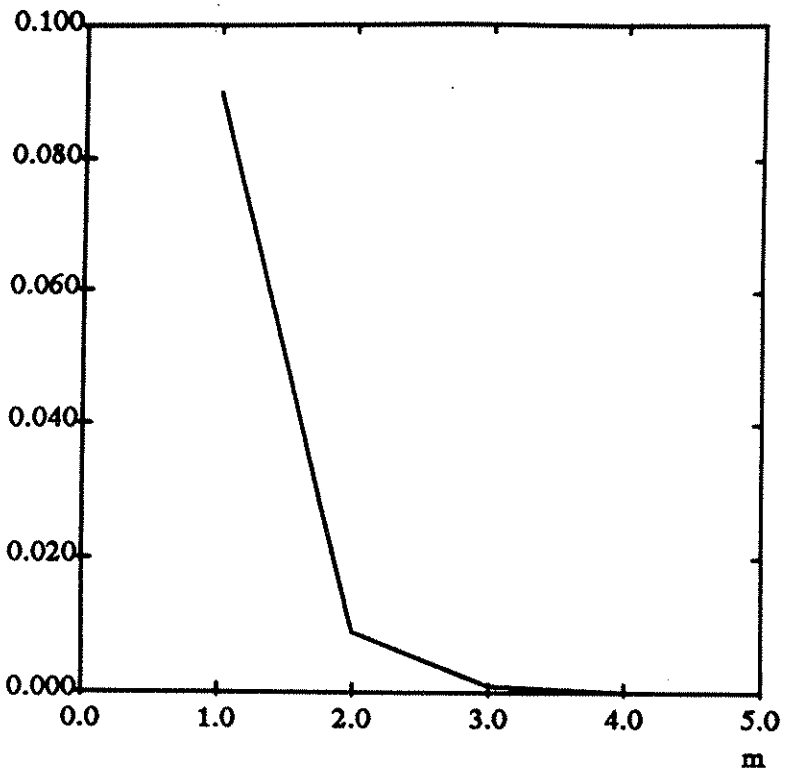

Fig. 1 Marginal increase of data object availability
Parameters: p=0.9, s=1.0, n=5.

REFERENCES

BAR85 Barbara, D., and Garcia-Molina, H., Evaluating Vote Assignments with A Probabilistic Metric, Proc. 15th International Symposium on Fault-Tolerant Computing, 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.

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

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..

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

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

FIS82 Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, IEEE Trans. on Software Engineering, May 1982, pp 198-202.

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.

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.

LAM81 Lampson, B., Atomic Transactions, Distributed Systems: Architecture and Implementation, Lecture Notes in Computer Science, Vol. 105, Springer-Verlag, 1981, pp 246-265.

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

SKE81 Skeen, D., Nonblocking Commit Protocols, Proc. ACM SIGMOD Conference on Management of Data, 1981, pp 133-142.

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

SON86 Son, S. H. and Agrawala, A., A Token-Based Resiliency Control Scheme in Replicated Database Systems, Proc. Fifth Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp 199-206.

SON86b Son, S. H. and Agrawala, A., An Algorithm for Database Reconstruction in Distributed Environments, Proc. 6th International Conference on Distributed Computing Systems, May 1986, pp 532-539.

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

WEI84 Weihl, W., Specification and Implementation of Atomic Data Types, Ph.D. dissertation, MIT Tech. Rep. MIT/LCS/TR-314, 1984.