USING REPLICATION FOR CONSISTENCY AND AVAILABILITY IN FAULT-TOLERANT DISTRIBUTED SYSTEMS

Sang Hyuk Son

Computer Science Report No. TR-86-24
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

Considerable research effort has been concentrated to the problem of developing techniques for achieving high reliability of critical data in distributed systems. One approach is to use replication. Replicated data is stored redundantly at multiple sites so that it can be used by the user even if some of the copies are not available due to failures. This paper introduces a scheme for maintaining consistency and improving availability of replicated data in distributed systems. A multiversion technique is used to increase the degree of concurrency. To reduce the storage requirement and communication overhead, multiple versions are maintained only at read-only sites. Methods for recovery of replicated data in distributed systems are discussed.

1. Introduction

A distributed system consists of multiple autonomous computer systems (called *sites*) that are connected via a communication network. Distributed systems and fault tolerant computing are closely related. In a distributed system, the physical separation of sites ensures the independent failure modes of sites and limits the propagation of errors throughout the system. Distributed systems must 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.

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 at different sites so that the system can access the data even if some of the copies are not available due to failures. 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. Two major technological developments have made the implementation of replication techniques cost-effective: inexpensive processors and memory produced by recent VLSI technology, making it possible to develop large networks, and new communication technology, making it feasible to implement distributed algorithms with substantial communication requirements. 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 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 guarantees the mutual consistency.

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. In such a case, an interleaved execution of read and write operations of user requests may produce incorrect results. 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. Unless a correct concurrency control mechanism is exercised to restrict the methods of interleaving the operations from different user requests, anomalous situations such as lost update, incorrect retrieval, and inconsistent update would occur[BER81].

A number of concurrency control schemes proposed are based on the maintenance of multiple versions of data objects[BAY80, BER83, CHA82, 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.

In this paper, we propose a resilient synchronization scheme for replicated data, based on the multiversion mechanism and the notion of token. For the concurrency control, we make use of Reed's multiversion time-stamp scheme[REE83]. The replication method used here masks failures as long as one of the token copies remains available. This approach has been taken in existing replication methods such as the *primary copy* method[ALS76, STO79], *true-copy token* method[MIN82], and *available copy* method[BER84]. In our method, there are predetermined number of tokens for each data object. Tokens are used to designate a read-write copy, and a token copy is a single version representing the current value of the data object. Multiversions are stored and managed only at non-token copy (read-only copy) sites. The mechanism is designed to support a replicated distributed system in increasing the availability of data and the degree of concurrency without incurring too much storage and processing overhead.

In contrast to true-copy token method, not all the copies are token copies, and only one type of token is used instead of separate exclusive-copy token and shared-copy token as in [MIN82]. Our scheme achieves higher availability of data objects than the true-copy scheme because a data object can be accessed and updated even if some of the copies are not available.

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 request. Its main drawback is its vulnerability to failures of primary copy sites.

The *available copy* scheme[BER84] is a complicated descendent of primary copy algorithms. 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 user requests may read from any copy, and must write to all available copies.

The replication method of our scheme might be considered as a generalization of those primary copy or available copy methods. If only one token for each data object exists, it is similar to primary copy method. If all the copies are token copies, then it is similar to the available copy method. Our scheme is different from them in that it exploits the multiple versions in increasing the degree of concurrency of the system. In addition, the scheme does not require special *status transactions* as in the available copy method, in which they are executed to keep the configuration information up-to-date as sites fail and recover.

The paper is organized as follows. Section 2 presents a model of computation used in the paper. Section 3 describes the execution of logical operations by corresponding physical operations. Section 4 presents the low cost concurrency control mechanism for replicated data in distributed systems. Section 5 sketches the correctness proof. Section 6 presents two recovery procedures that can be used for replicated data objects, and Section 7 discusses the availability of replicated data objects. Section 8 concludes the paper.

2. Model of Computation

This section introduces our model of computation. We describe the notion of transactions, logical organization of replicated data, tokens, and our assumptions about the effects of failures.

2.1. Transactions

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]. 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 system, it would terminate in a finite time and produce correct results, leaving the system 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. 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. Algorithms for replication control and synchronization pay no attention to the local computations; they make scheduling decisions on the basis of the data objects in the read set and write set of transactions.

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.

When a transaction commits, all the updates it made must be written permanently into the system. 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. We assume that the system runs a correct commit algorithm (e.g., [SKE81]), and hence assures the atomic commitment of transactions.

A time-stamp is a number that is assigned to a transaction when initiated, and is kept by the transaction. Each site generates a unique local time-stamp, and 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 (1) no two transactions have the same time-stamp, and (2) 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*.

2.2. Logical Data Objects and Tokens

The smallest unit of data accessible to the user is called *data object*. Data objects do not correspond directly to real items; they are an abstraction. In a particular system, the data objects might be files, pages, records, etc. Each data object has a *value*. In distributed systems with replicated data objects, a logical data object X is represented by a set of one or more replicated physical data objects {x1, x2, x3, ... }. 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 system, 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 system. Even if the actual system executes many transactions at a time using several replicated physical data objects, the system must provide the user a view of a

single-copy of each logical data object.

A token designates a read-write copy. Each logical data object has a predetermined number of tokens, and each token copy is the latest version of the data object. The site which has a token copy of a logical data object is called a *token site*, with respect to the logical data object. In order to control the access to data objects, the system uses time-stamps. 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 consecutively numbered versions. There is a special storage unit, called the *version storage*, which is used for storing the old versions of data objects that have read-only copies at the site. Old versions in the version storage is used for providing consistent view of the system to transactions.

A transaction does not necessarily read the latest committed version of a data object. The particular old version that the transaction has to read is determined by the time-stamp of the transaction. Before the transaction actually reads the desired old version, any number of committed versions might have been created by other transactions. Therefore, it is not appropriate to maintain a fixed number of old versions of each data object in the version storage. In order to accommodate an arbitrary number of old versions of a data object, we may chain the old versions in a reverse chronological order.

Garbage collection of old versions that are no longer required by ongoing transactions is one of the technical problems that must be solved in order to implement a multiversion mechanism efficiently. Two issues involved in the problem of garbage collection are (1) when to collect the garbage, and (2) how to determine the old versions not required any more. We refer to the "ring buffer" data structure proposed in [CHA85] for these issues, and do not consider them in this paper.

2.3. Failure Assumptions

A distributed information system can fail in many different ways, and it is almost impossible to make an algorithm which can tolerate all possible failures. In general, failures in distributed information systems can be classified as failures of *omission* or *commission* depending on whether some action required by the system specification was not taken or some action not specified was taken[MOH83]. The simplest failures of omission are *simple crashes* in which a site simply stops running when it fails. The hardest failures are *malicious runs* in which a site continues to run, but performs incorrect actions. Most real failures lie between these two extremes.

In this paper, we do not consider failures of commission such as the "malicious runs" type of failure. 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[BER84].

We assume that network partitioning never occurs. This assumption is reasonable for most local area networks and some long-haul networks. Necessary modifications of the scheme to relax this assumption is discussed in Section 7. We also assume that routine communication errors such as lost and duplicate messages are handled by the network, and hence between any two sites, messages are received in the order they are sent.

3. Execution of Logical Operations

In a replicated system, the system must provide the same effect in executing logical operations as if data objects were nonreplicated. 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 updated the

version.

To read the data object X, the coordinator sends a request to a read-only copy site of X. For now, we assume that an appropriate version always exists at a read-only copy site where a logical read is requested, and that it is implemented by a physical read of that version. Since multiple versions of a data object are used for read operations, read operations are translated into *version-read operations*, in which $r_i(X)$ is translated into $r_i(X_j)$ for some j.

Execution of logical write operations is not as simple as logical read operations. In a straightforward implementation of logical writes, 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 logical write was requested. The logical 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 implementation of logical write operations 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 Synchronization Scheme

To reduce the cost of logical write operations, the level of synchronization between logical and physical write operations can be relaxed by allowing physical write operations to be completed by the commit time of the transaction. A logical write operation is considered completed when the update request messages are sent. This eliminates the delay caused by waiting for confirmation messages before the next operation can proceed.

4.1. Conflict Resolution

We use time-stamp ordering for concurrency control. Each read and write carries the time-stamp of the transaction that issued it, and each copy carries the time-stamp of the transaction that wrote it. A *conflict* occurs when a transaction issues a request to access a data object for which other transaction has previously issued a request to access, and furthermore at least one of these requests is a write request. If both requests are read requests, both will be granted. There are two kinds of conflicts.

(1) Read-write (RW) conflict: A transaction requests to read a data object for which other transaction already issued a write request.

(2) Write-write (WW) conflict: A transaction requests to write a data object for which other transaction already issued a write request.

In each case, we say that the transaction requesting the new access has *caused* the conflict.

A conflict can occur only at token sites, since there are no locks for read-only copies. A RW conflict occurs when a token copy is required to give the current value while it is locked for an update, and a WW conflict occurs when a write request arrives before the termination of the transaction which issued the write operation of the token copy.

$W_i(X_i)$ is translated into $w_i(X_i)$ for all available token copies. A write-read conflict does not occur since each write operation creates a new version which does not invalidate the previous version. However, a w_i(X) 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.

Let T_1 be the transaction which already issued an access request, and T_2 cause the conflict. For each token copy of X, conflicts are resolved as the following:

(1) RW conflict: If T_2 is younger than T_1, then it waits for the termination of T_1. If T_2 is older than T_1, then it reads before-value of X.

(2) WW conflict: If T_2 is younger than T_1, then it waits for the termination of T_1. If T_2 is older than T_1, then T_2 is rejected.

4.2. Copy Actualization

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 time-stamp $\leqslant$ 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 $r_i(X)$, and the current version of the data object will be returned. The latest version can be determined at the read-only copy site by comparing the time-stamp of the read-only copy and that of the token copy.

The copy actualization procedure 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 ARM 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.

4.3. Commitment

Since we use token copies and before-values in transaction processing, simple two-phase commit in which unanimous Precommit Messages from all the participants are enough, is not sufficient for the commitment of transactions. The commit rule used in our scheme is as follows:

Commit Rule

The coordinator of a transaction T decides to commit when the following conditions are satisfied:

(R1) All the available token sites of each data object in the write-set have precommitted T by sending Precommit Messages.

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

When an update transaction T_i working on a data object X is committed, each token copy sends Remote Update Messages (RUM) to read-only copy sites of X. On receiving a RUM, a new version of the data object, X_i, is created and tagged with the time-stamp of T_i, and saved in the version storage.

5. Consistency of Replicated 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 replication, *one-serializability* (1-SR) has been used as the correctness criterion for transaction executions[BER83]. In this section, we briefly review fundamental concepts associated with 1-SR, and show that our low cost concurrency control technique guarantees the consistency of the system by satisfying the requirements of 1-SR.

5.1. 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 copies and a final transaction that reads from all data copies. Initial and final transactions are not the user transactions; they are used only to determine the initial and terminal state of the system. We consider only the augmented log in this paper. 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 when all the activity ceases 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.

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.

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 of 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 $k \neq i$ 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.

5.2. Correctness Proof

To show the correctness of our algorithm, we first infer properties which all logs produced by the algorithm will satisfy, and then show that these log properties imply 1-serializability.

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 the 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 correctness of our algorithm can be presented as the following theorem.

Theorem 4: All logs produced by our algorithm are 1-serializable.

Proof: Let L be a log produced by the mechanism. 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 $\leqslant$ time-stamp(T_i), $r_j(X_i)$ implies that time-stamp(T_i) $\leqslant$ time-stamp(T_j). From the uniqueness property of time-stamp that for all i and j, i$\neq$j, time-stamp(T_i) $\neq$ time-stamp(T_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) $X_i \ll X_j$

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

Case 2) $X_j \ll X_i$

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), and therefore 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. □

6. Site Recovery

Sites of a distributed system may fail and recover from time to time during the life-time of the system. In a distributed system with replication, transactions are allowed to be executed even if some of the copies of data objects are not available due to failures, in order to increase the availability of the system. When a failed site recovers, the consistency of the entire system may be threatened if proper recovery mechanisms are not exercised.

The recovering site must perform local recovery using the transaction log to bring the non-replicated data objects at the site to a most recent committed state. It then begins global recovery to bring the replicated data objects up-to-date with respect to the rest of the system. 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 local as well as global recovery in order to bring the system into a consistent state. In this section we discuss only the global recovery of replicated data objects. A more detailed discussion on site recovery is given in [SON86].

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. All update messages addressed to an unavailable site are saved in multiple spoolers so that they can be delivered when the site recovers unless all the spoolers fail. The recovering site executes all the missed updates before resuming normal operation. We do not pursue 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, hence the availability of the system is increased. Algorithms using this approach have been studied in [BER84, BHA86]. In this section we present two recovery procedures, that belong to the class of the second approach. We also discuss the trade-offs between two recovery procedures.

6.1. 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 site failures. 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. This task can be achieved by executing an algorithm similar to the algorithms proposed in [GOO83, SKE85].

During a total failure of the data object, no transaction using the data object can be processed. Therefore, if the token-copy removed was the only one available, then the transaction currently using that data object must be aborted. If the read-only copy removed was used for the processing of a transaction, the transaction must find another copy for read operation, and can be continued only when the substitution is successfully completed.

To recover from a partial failure, the recovering token copy must be updated to the current value of the logical data object before being included in the list of available token 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 list simply by appending the identifier of the recovered site, without being updated on recovery.

6.2. Updating Site Status

The second recovery procedure is based on keeping track of the status of sites instead of maintaining the status information 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.

6.3. Trade-offs in Recovery

There is a trade-off between the processing time during normal operation and the time required to perform recovery procedures. In the second approach, 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.

7. Discussion

One of the important properties of our mechanism is the flexibility. By manipulating the number of tokens for each logical 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 logical data object. In the first case, there is no need to have tokens and any copy can be used for read-write purposes. The copy actualization procedure can be omitted, resulting in a simpler scheme at the expense of increased number of sites involved in updating a data object.

The second case is similar to primary copy algorithms. As pointed out in [GIF79], primary site 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.

If the network does not become partitioned (i.e., when two sites are "up", they can always communicate), transaction control at failure situations is simple: a transaction can commit when all the participants have precommitted. However, in some networks which does not provide multiple paths between sites, we need to consider network partitioning. What makes the problem more complex is that it is not easy to distinguish the network partitioning from site crashes. To adapt our scheme to the system where network partitioning can occur, we need to use different commit rules as the following:

(R1') Majority of the token sites of each data object in the write-set have precommitted.

(R2') One copy of each data object in the read-set is actualized from the majority of token copies and is precommitted.

In order to make this modified scheme to work, the number of original token-copies must be stored with the directory of a data object. This modified scheme is able to handle the network partitioning, but reduces the robustness of the original scheme because the system cannot process transactions if majority of the token sites are not available (original scheme can process a transaction with one token copy available).

No matter how many token copies exist, it is always possible to enter a state in which no token 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. These probabilities represent the expected fraction of time each component is not able to provide the service

correctly. Network topology plays a critical role in determining the availability of data objects when partitioning can occur. 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. A more detailed discussion on the availability of replicated data objects is given in [SON87].

8. 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.

We have presented a scheme for synchronization and consistency of replicated data in distributed systems. It reduces the time required to execute physical operations when updates are to be made on data objects that are replicated, by relaxing the level of synchronization between logical and physical write operations. At the same time, the consistency of the replicated data is not violated, and the atomicity of transactions is maintained. Our scheme extends primary site algorithms such that a transaction can be executed provided at least one token copy of each logical 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 scheme also exploits the old versions of a data object in increasing the degree of concurrency. Multiple versions are maintained only at the read-only copy sites, hence the storage requirement is reduced in comparison to other multiversion mechanisms[REE78, CHA85].

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.

BER79 Bernstein, P., Shipman, D., Wong, W., Formal Aspects of Serializability in Database Concurrency Control, IEEE Trans. on Software Engineering, May 1979, pp 203-215.

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..

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, CACM 19, Nov. 1976, pp 624-633.

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., 1978, 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.

LAM78 Lamport, L., Time, Clocks and Ordering of Events in Distributed Systems, CACM, 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.

MOH83 Mohan, C., Strong, R., and Finkelstein, S., Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors, Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, August 1983.

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.

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

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

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

SON87 S. H. Son, On Multiversion Replication Control in Distributed Systems, International Journal of Computer Systems Science and Engineering, Vol. 2, No. 1, January 1987 (to appear).

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

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.