USING REPLICATION TO IMPROVE RELIABILITY IN DISTRIBUTED INFORMATION SYSTEMS

Sang Hyuk Son

Computer Science Report No. TR-87-16
August 18, 1987

Using Replication to Improve Reliability in Distributed Information Systems

Sang Hyuk Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

Replication is the key factor in improving the availability of data in distributed information systems. Replicated data is stored at multiple sites so that it can be used by the user even when some of the copies are not available due to site failures. In this paper a synchronization scheme for distributed information systems is described. The scheme increases the reliability as well as the degree of concurrency of the system. A token is used to designate a read-write copy. The scheme allows transactions to operate on a data object if more than one token copies are available. The serializability theorem for replicated data and the recovery mechanisms associated with the scheme are discussed.

Index Terms - distributed system, consistency, synchronization, transaction, replication

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.

1. Introduction

A distributed information system consists of multiple autonomous computer systems (called *sites)* that are connected via a communication network. One of the advantages of distributed information systems over centralized information systems is that replicated copies of critical data can be stored at multiple sites. The main goal of having replicated data is to enhance the availability of data. By storing data at multiple sites, the system can access the data in the presence of failures, even though some of the redundant copies are not available. In addition to improved availability, replication also increases the reliability of data by reconstructing accidently destroyed copy from other copies. Replication can enhance performance by allowing queries initiated at sites where the data are stored to be processed locally without incurring communication delays, and by distributing the workload of queries to several sites where the subtasks of a query can be processed concurrently. These benefits of replicated data must be balanced against the additional cost and complexities introduced for replication control.

Correctness and availability appear to be conflicting goals in distributed information systems. A major restriction in using replication is that replicated copies must behave like a single copy, i.e., *mutual consistency* of a replicated data must be preserved. By mutual consistency, we mean that all copies converge to the same value and would be identical if all update activities cease. The inherent communication delay between sites that store and maintain copies of a replicated data makes it impossible to ensure that all copies are identical at all times when updates are processed in the system. The principal goal of a replication control mechanism is to guarantee that all updates are applied to copies of replicated data in a way that assures the mutual consistency.

Considerable research effort has been focused in recent years in developing techniques for storing and retrieving data reliably through replication, and most of them seem to focus on correctness of the database by guaranteeing global serializability while allowing only a limited amount of availability for partitioned operation. These include [BER84, EAG83, HER86, MIN82, THO79]. On the other extreme, there are methods that provide practically unlimited availability during partitions at the expense of abandoning global serializability as a correctness criterion. These include [SAR85, DAV84].

To illustrate the basic differences between the conservative methods (which allow at most one partition to process transactions) and optimistic methods, we consider a simple banking database. Suppose that the information on the balances of funds in different accounts are replicated and stored at two sites, which are connected by a communication link. Let us assume that two independent withdrawal transactions are submitted at each sites when two sites are partitioned by the link failure. If a conservative method is used, only one site will accept a user transaction for the withdrawal, although the consistency can be maintained even with the execution at the other site. In an optimistic method, both withdrawal transactions will be executed. However, after communications are restored, the inconsistency (overdrawn) will be detected, and a corrective action will be necessary. Therefore, the basic difference between these two methods is the trade-off between availability and correctness; conservative methods resulted in the loss of service availability while preventing inconsistent execution, and optimistic methods insured the operability of both sites while allowing an account to be overdrawn.

Conservative methods are quite satisfactory for the applications where high availability is not of primary concern. However, if availability is critical, optimistic methods seem to make more sense. Each of the optimistic methods has unique advantages and shortcomings, but it appears that there are problems common to all of them: computation and communication overhead. Each site must exchange the information about transactions executed during partition, and determine which transactions must be executed locally and which transactions must be backed out for the database consistency. Therefore, one of the issues that need further study in optimistic approach for replication control in distributed information systems is the overhead control: reducing overhead while maintaining high availability.

In this paper, we propose a synchronization scheme for distributed information systems with replicated data objects. Our objective is to permit each site to process transactions as much as possible, while reducing the possibility of conflicts among committed transactions by restricting the number of copies that can be used for updating data objects. The replication method used here masks failures as long as one special copy (token copy) remains available. Similar 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 scheme, 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. The scheme is designed to support a distributed information system in increasing the availability of data and the degree of concurrency without incurring too much 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 token 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 transaction. Its main drawback is its vulnerability to failures of primary copy sites.

The available copy scheme[BER84] is a 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 transactions 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 before-values 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 introduces the important notions used in the scheme. Section 4 describes the execution of logical operations by corresponding physical operations. Section 5 presents the synchronization scheme for distributed information systems with replicated data. Section 6 introduces the serializability theorem, and sketches the correctness proof of the scheme. Section 7 presents a recovery procedure that can be used for replicated data objects, and Section 8 discusses the availability of replicated data objects. Section 9 concludes the paper.

2. Model of Computation

To present our synchronization scheme, we need to introduce first the organization of distributed information systems. In this section we present a simplified model of a distributed information system, and describe how the system processes transactions.

2.1. Distributed System Environment

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

The smallest unit of data accessible to the user is called *data object*. In distributed information systems with replicated data objects, a logical data object is represented by a set of one or more replicated physical data objects. Two types of *logical operations* that can be performed on a logical data object are read and write. A logical operation requested at one site is implemented by executing *physical operations* on one or more copies of physical data objects in question.

2.2. Transactions

Users interact with the system by submitting transactions. Each transaction represents a complete and correct computation, i.e., if a transaction is executed alone on an initially consistent database, it would terminate in a finite time and produce correct results, leaving the database consistent. A transaction consists of different types of operations such as read, write, and local computations. Read and write operations are used to access data objects, and local computations are used to determine the value of the data object for a write operation. Algorithms for replication 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.

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 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. 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].

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*.

2.3. 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.

3. Token Copy and Before-Value

A token designates 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) go through the *copy actualization phase*, if necessary, in order to satisfy the consistency constraints of the system.

When a transaction performs a write operation to a data object, there are two values that are associated with the data object; after-value (the new version) and before-value (the old version). The system remembers the before-value for the duration of the transaction so that it can be restored if the transaction is rolled back. Some systems even have a permanent copy of old versions for better availability[REE83].

Because the before-value is available during the transaction processing, it is natural to ask if concurrency can be improved by giving out this value[STE81]. For example, if the transaction T_1 has been given a permission to write the new value of a data object and the transaction T_2 requests to read the same data object, then it is possible to give T_2 the before-value of the data object, instead of making T_2 wait until T_1 is finished. However, an appropriate control must be exercised in doing so, otherwise the database consistency might be violated. In the example above, assume that T_1 has written a new value for two data objects X and Y, and T_2 has read the before-value of X. T_2 wants to read Y also. If T_2 gets the after-value of Y created by T_1, there is no serial execution of T_1 and T_2 having the same effect because in reading the before-value of X, T_2 sees the database in a state before the execution of T_1, and in reading the after-value of Y, T_2 sees the database in a state after the execution of T_1.

4. Execution of Logical Operations

In a distributed information system with replicated data, 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 $W_i(X)$.

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 decision is made in selecting a read-only copy, and a logical read operation is implemented by a physical read of that copy.

Execution of logical write operations is not as simple as read operations. In a straightforward implementation of logical writes, the value to be written is broadcast to all token sites where a token 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. The level of synchronization between logical and physical write operations is 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 required update messages are sent. This eliminates the delay caused by waiting for confirmation messages before the next operation can proceed.

5. The Synchronization Scheme

Operations of transactions are executed differently, depending on whether the system is partitioned or not. In normal mode when there is no partition failure, a transaction reads from any copy and writes to all token copies. In partitioned mode, any partition that has a token copy can process transactions which need the data object. We first present normal mode of operation.

5.1. Normal Operation

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

(3) 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.

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) WR conflict: If T_2 is younger than T_1, then its write request is granted with the condition that T_2 cannot commit before the termination of T_1. If T_2 is older than T_1, then T_2 is rejected.

(3) 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.

When T_2 reads before-value of X of T_1, the token site which gives before-value, informs the coordinator of T_1 the identifier of T_2. The transaction manager at the read-only site of T_2 also informs the coordinator of T_2 the identifier of T_1. Same procedure is executed when a younger transaction is granted a write access while an older transaction is granted a read access on the same data object.

The coordinator of a transaction maintains the *before-list* (BL), a list of transactions which read before-value of any data object in its write-set, and the *after-list* (AL), a list of transactions which write after-value of any data object in its read-set. The BL and the AL are used during the commitment of a transaction as described later in this section.

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. For a read operation $R_i(X)$, if the read-only copy of X has time-stamp > time-stamp(T_i), then the value is used for $r_i(X)$. 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.

Copy actualization is also used to actualize a token copy recovered from the crash. During the recovery of the site, the Recovery Manager of the site tries to update all the token copies at the site by sending 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.

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:

(C1) All the available token sites of each data object in the write-set have precommitted T by sending Precommit Messages.

(C2) One copy of each data object in the read-set is available and has precommitted T.

(C3) There is no active transaction which has seen before-values of any data object in the write-set of T.

When the transaction T_1 reads before-value of a data object of the transaction T_2, T_1 is included in the BL of T_2, and T_2 is included in the AL of T_1, by the synchronization mechanism. When a transaction terminates (either commits or aborts), the coordinator of the terminating transaction must inform the coordinator of each transaction in its AL about the termination by sending Termination Messages (TM). On receiving a TM from the coordinator of a transaction in its BL, the coordinator of the active transaction removes the identifier of the terminating transaction (sender of the TM) from the BL. The condition C3 implies that a transaction can commit only when its BL is empty. It is required to prevent nonserializable execution sequences to occur.

5.2. Partitioned Operation

In partitioned mode, each partition can process transactions only if it has token copies of all data objects the transaction needs to access. At any given time, a partition's database must reflect those updates that it has seen, executed in time-stamp order. Since each partition cannot receive updates of other partitions, the states of each replicated data copy may diverge during partitioned operation.

When two partitions are merged, each site determines the actions it must take to construct a globally consistent state. Conceptually, these actions include undoing each transaction with higher time-stamp, executing new updates, and then rerunning some of the transactions which were undone. Undoing and redoing transactions are usually expensive, but this merge process can be made more efficient by exploiting simple semantic properties of transactions. For example, two transactions with nonintersecting sets of data objects to be accessed can always be executed in any order, i.e., commutative. Another example of exploiting semantic information for the commutativity is the transactions to withdraw from and to deposit to the same bank account. Sites can use this information to reduce the number of actions necessary for the merge, without compromising correctness of the database. When one or more transactions must be integrated with already committed transactions, an initial log is composed by placing them in their time-stamp order, and then optimization rules are applied to remove some of the merging actions based on commutativity and other semantic information. For this process, optimization methods for merge operation similar to one developed in [BLA85, SAR85] can be used.

6. Consistency of Replicated Systems

A synchronization scheme is said to be correct if, when a set of transactions is processed, the same state results as if the transactions were processed in a serial fashion. In distributed systems with replicated data, *one-copy serializability* (1-SR) has been used as the correctness criterion for transaction executions[BHA86]. In this section, we briefly review fundamental concepts associated with 1-SR, and show that our synchronization scheme ensures the consistency of the system. The correctness of our synchronization scheme is based on the facts that the BL of a transaction becomes empty in a finite time, and that there is no cycle in the serialization graph generated by the scheme as discussed below.

6.1. Serializability of Transactions

A *log* is a model of execution of transactions, which captures the order in which read and write operations are executed. 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.e., $w_i(X)$ *precedes* $r_j(X)$ (denoted by $w_i(X) \rightarrow r_j(X)$), and no $w_k(X)$ falls between these operations. Two logs are equivalent if and only if they have the same reads-from relationships.

A *serial log* 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:

$T_i \rightarrow T_j$ implies that for some data object X, $r_i(X) \rightarrow w_j(X)$, or $w_i(X) \rightarrow r_j(X)$, or $w_i(X) \rightarrow w_j(X)$.

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 $T_1 \rightarrow T_2$. The serialization graph is a convenient tool to deal with the inconsistency problem. A cycle in the serialization graph reflects an inconsistency. This is because the precedence relations among committed transactions must be acyclic in order to get a serializable execution sequence of committed transactions. Transactions can be arranged according to a topological ordering of the transaction precedence defined in the graph. This can be summarized by the serializability theorem that provides a necessary condition for the correctness of a log as follows[BER81].

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

A serial log L in a replicated 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 copy of X. A log is *one-copy serializable* (or 1-SR) if it is equivalent to a 1-serial log. A log L over a set of transactions in a replicated system is equivalent to a serial log in a non-replicated system over the same transactions if and only if L is 1-SR[BER84].

6.2. Correctness Proof

To show the correctness of our synchronization scheme, we first need to prove that all the conditions of the commit rule will become true in a finite time. Since the third condition of the commit rule requires the BL being empty, the following lemma shows that the system is not blocked (i.e., transactions are processed and terminated in a finite time).

Lemma 2: The BL of a transaction becomes empty in a finite time.

Proof:

From the property of the time-stamp, only a finite number of transactions can have time-stamps less than that of a given transaction. The fact that the transactions in the BL of a transaction T_1 are older than T_1, combined with the property of the time-stamp, implies that only a finite number of transactions can be in the BL of a transaction.

Among active transactions in the system, there must be the oldest transaction which has empty BL because all the transactions older than that are terminated (otherwise, it cannot be the oldest). When this transaction terminates, it sends a Termination Message to each transaction in its AL. Therefore the oldest transaction will be removed from the BLs of active transactions, making the size of the BL decreased by 1. The second oldest transaction now becomes the oldest, and has no transaction in its BL. When it terminates, the same thing will occur and the size of the BL of the transactions in the AL of the terminating transaction will be decreased. Since at any instant of time there exists at least one transaction whose BL is empty and hence can terminate and decrease the size of the BLs of other transactions, the BL which can have only a finite number of transactions in it will become empty in a finite time. □

Since our scheme uses time-stamps in resolving conflicts between transactions, it is not necessary to maintain the serialization graph. The serialization graph is introduced only to show the working principle of the scheme and to prove the correctness of it.

By the following theorem, we prove that our scheme maintains the consistency by showing that there cannot be a cycle in the serialization graph.

Theorem 3: All the logs generated by the scheme are 1-SR.

Proof:

Let L be a log produced by the scheme. We prove that there cannot be a cycle in SG(L). Assume that there is a cycle in the serialization graph SG(L). Let the cycle consist of transactions $T_1, T_2, ..., T_n$ in that $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow ... \rightarrow T_n \rightarrow T_1$. The last edge in the cycle, $T_n \rightarrow T_1$ can be added into the serialization graph by two ways: T_n reads the before-value of a data object on which T_1 has a write-lock, or T_1 reads the after-value of a data object written by T_n.

Consider the first case. It has two subcases: either T_1 is already committed or is still active. In the first subcase, T_n cannot read the before-value because the before-value of a committed transaction is not allowed to access, and hence cannot create the edge. In the second subcase, first consider the case when T_n is younger than T_1. Reading of the before-value is not allowed in this case by the locking mechanism. If T_n is older than T_1, there must be a transaction in the cycle which is older than its left neighbor. Let two transactions T_x and T_y be those transactions such that $T_x \rightarrow T_y$ and T_x is younger than T_y. Since T_x is not allowed to read the before-value of T_y, T_y must have read the after-value of T_x to generate this edge. It implies that T_x is committed because reading of after-value of nonterminated transaction is not allowed. From the commit rule, a transaction can commit only when its BL is empty. Therefore the commitment of T_x implies the commitment of T_1, which contradicts the assumption that T_1 is still active.

Consider the second case. Reading the after-value written by T_n implies that T_n is committed. However T_n cannot be committed because unless T_1 is committed, no transaction in the cycle can be committed. Since T_1 is requesting a read-lock it cannot be terminated. Again we have a contradiction on the commitment of T_n. Therefore there is no cycle in the serialization graph SG(L), and all the logs produced by the scheme are 1-SR. □

7. Site Recovery

Sites of a distributed information system may fail and recover from time to time during the life-time of the system. In a distributed information system with replication, transactions should be allowed to execute 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 might 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. Global recovery needs to be executed 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*. In order to bring the system into a consistent state, site recovery must perform local as well as global recovery. 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 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 the increase of the availability of the system. 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.

7.1. Updating Directories

Our 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 a Update

Directory Message (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 [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 read-only copy can be included in the available list simply by appending the identifier of the recovered site, without being updated on recovery.

7.2. Updating Site Status

Our 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 other 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.

7.3. 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 include (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 in 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 to be made 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.

8. Discussion

One of the important properties of our scheme 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 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.

There are applications in which the database inconsistency during partition cannot be allowed, i.e., the database must remain consistent at all times. To adapt our scheme to the system for those applications, we need to use different commit rules 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 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 availability of data objects of the original scheme because now the system cannot process transactions if majority of the token sites are not available (original scheme is able to 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 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].

9. 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 viewpoint of logical correctness.

We have presented a synchronization scheme for distributed information systems with replicated data. Two recovery mechanisms associated with the scheme are presented, and the availability of replicated data objects are discussed. The scheme reduces the time required to execute physical operations when updates are to be made on replicated data objects, 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.

The scheme extends primary copy 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. It also exploits the old version of a data object in increasing the degree of concurrency.

Reliability does not come for free. There is a cost associated with the replication of data: storage requirement and complicated control in synchronization. In some applications of distributed information systems, this cost of replication may not be justifiable. However, for certain applications in which high reliability and availability of the information system is critical (e.g., ballistic missile defense or air traffic control system), it is worthwhile to use replication techniques with an appropriate synchronization scheme.

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.

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

BER84 Bernstein, P., Goodman, N., An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases, ACM Trans. on Database Systems, Dec. 1984, pp 596-615.

BHA86 Bhargava, B., Ruan, Z., Site Recovery in Replicated Distributed Database Systems, Proc. 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 621-627.

BLA85 Blaustein, B. and Kaufman, C., Updating Replicated Data during Communications Failures, Proc. of 11th VLDB, 1985, pp 1-10.

DAV84 Davidson, S., Optimism and Consistency in Partitioned Database Systems, ACM Trans. on Database Systems, Sept. 1984, pp 456-482.

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, Vol. 13, December 1979, pp 150-162.

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.

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 Symposium on Principles of Distributed Computing, August 1983.

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

SAR85 Sarin, S., Blaustein, B., Kaufman, C., System Architecture for Partition-Tolerant Distributed Databases, IEEE Trans. on Software Engineering, Dec. 1985, pp 1158-1163.

SKE81 Skeen, D., Nonblocking Commit Protocols, Proc. ACM SIGMOD International 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. K., An Algorithm for Database Reconstruction in Distributed Environments, 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 532-539.

SON87 Son, S. H., Synchronization of Replicated Data in Distributed Systems, Information Systems, Vol. 12, No. 2, 1987.

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.