Computer Science Technical Reports | Department of Computer Science

1985

Site Recovery in Replicated Distributed Database Systems

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
85-564

SITE RECOVERY IN REPLICATED DISTRIBUTED DATABASE SYSTEMS

Bharat Bhargava
Zuwang Ruan

CSD-TR-564
December 1985

SITE RECOVERY IN REPLICATED DISTRIBUTED DATABASE SYSTEMS *

Bharat Bhargava
Zuwang Ruan

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

A solution to the problem of integrating a recovering site into a distributed database system is presented. The basic idea used for the correct recovery is to maintain a consistent view of the status (up or down) of all sites. This view need not be the exact current status of the sites, but is the status as perceived by other sites. The *session number* is used to represent the actual state of a site, while the *nominal session number* is used for the session number as perceived by other sites. The consistent view of the nominal session numbers are maintained by control transactions, which run concurrently with user transactions. This approach provides a high degree of availability. A data item is available to a transaction as long as one of its copies is in an operational site and the transaction knows the site's session number. The recovery procedure allows the recovering site to resume its normal operations as soon as possible.

1. Introduction

Site recovery is the problem of integrating a site into a distributed database system (DDBS) when the site recovers from a failure. There are two different problems under the term "site recovery" in the literature. The first concerns the resolution of transactions. That is, upon recovering, the site should commit or abort the transactions that were being processed when the failure occurred, consistently with the decisions made by other operational sites in the system. Termination protocols in conjunction with commit protocols make it possible for the recovering site to make correct decisions on these transactions [9, 10]. The second problem deals with the recovery of the database. This problem is caused by attempts to increase database availability. In a distributed database system with replication, we would like user transactions to proceed even if some sites are unavailable due to failures. When a failed site recovers and joins to the system, the consistency of the entire database is threatened because the data items at the recovering site may have missed some updates. The recovery algorithm should hide such inconsistency from user transactions, bring the recovery site up-to-date with respect to the rest of the system, and enable the recovering site to start processing transactions as soon as possible. This paper discusses only the second problem.

There are two main approaches to this problem. The first is to redo all missed updates at the recovering site. The use of multiple *message spoolers* [6] is one practical solution using this approach. All update messages addressed to an unavailable site are saved reliably in multiple spoolers, and the recovering site processes all of its missed messages before resuming normal operations. However, scheduling the missed operations is a nontrivial problem if there is no global clock available. Moreover, this method is not suitable for systems in which sites may be down for quite a long time.

The second approach takes advantage of data replication. In this approach, the data items at the recovering site are brought up-to-date by special transactions, called *copiers,* which read the corresponding replicas at operational sites and refresh the out-of-date copies. An advantage of this approach is that copier transactions can run concurrently with user transactions so that the recovering site can start processing user transactions as soon as possible. However, since data items are brought

* This work is supported in part by the grant from Sperry Corporation and by David Ross Fellowship.

into the database separately by independent copiers, special mechanisms are required to ensure that the database keeps its consistency while the individual data copies are merging into it. Roughly speaking, the algorithm must guarantee that no user transaction can read a copy at the recovering site before the copy is renovated, and once a copy has joined into the database, all transactions writing to the logical data item must update this copy as well. Algorithms using this approach have been proposed in [1, 2].

It should be noted that a recovery procedure alone cannot ensure a correct recovery. Without proper conventions for user transactions the situations left by site failures may not be recoverable at all. The following example illustrates the problem. In this example write operations are interpreted as writing to all currently available copies and transactions can be committed as long as all write operations succeed.

Example. Transaction T_a reads X and writes Y, transaction T_b reads Y and writes X. Both X and Y have two copies at site 1 and site 2, called x_1, x_2 and y_1, y_2, respectively. A history

$$R_a[x_1]\ R_b[y_1]\ \ (\textit{site 1 crashes})\ \ W_a[y_2]\ W_b[x_2]$$

is acceptable by a concurrency control algorithm that concerns only the serializability of physical operations. Because both T_a and T_b have written to all currently available copies, they are committed. When site 1 recovers, x_1 and y_1 may be updated by copier transactions $T_c = R_c[x_2]W_c[x_1]$ and $T_d = R_d[y_2]W_d[y_1]$. No matter how the copiers are scheduled, the database cannot be brought up to a consistent state. □

The solution in [1] does not specify its model for user transactions, hence it is unclear how such anomalies are prevented from happening. The solution proposed in [2] uses *directories*. Briefly, each data item is associated with a directory that keeps the status of the data item, i.e. where the copies of the data item are available. User transactions read the directories to decide how to interpret their read and write operations. Directories are manipulated by status transactions, including the copiers, called INCLUDE transactions in [2]. User transactions and status transactions are synchronized by the two-phase locking protocol so that user transactions can have a consistent view of the status of each individual data item.

In this paper we present a new solution, that also belongs to the class of the second approach. The solution is motivated by the following considerations. First, the conventions for user transactions should not degrade the performance of normal operations too much. Hence we try to identify the necessary information that user transactions need to know in order to avoid unrecoverable situations. We have determined that a consistent view of the status of all sites is sufficient as far as site failures are concerned. Keeping track of the status of sites turns out to be much less expensive than maintaining the status information for individual data items. Next, the recovering site is expected to start processing transactions as soon as possible. Therefore, one of our goals is to reduce the work that must be done before the recovering site resumes its normal operations. In our algorithm, as soon as the recovering site has successfully informed the other operational sites of its new status, it becomes fully operational. The recovery of the data items proceeds concurrently with user transactions. Finally, in order to eliminate unnecessary work, it is important to identify precisely the data items that have missed updates and need to be refreshed. Rather than having a mechanism built into it, our algorithm can choose many different methods for identifying the out-of-date items and make recovery efficiently.

This algorithm works with a large group of concurrency control algorithms and provides considerable implementation freedom. It is resilient to multiple site failures, even if a site crashes while another site is recovering. A failed site can recover as long as there is at least one operational site in the system. Though we foresee that a similar method can be applied to the problem of the merging of network partitions, the algorithm presented in this paper does not handle partition failures.

The next section serves as a background. Section 3 presents our basic algorithm. Section 4 sketches the correctness proof. Section 5 discusses its refinements. Finally, the last section concludes this paper and outlines further work.

2. Background

In this paper, the users' view of an object is called a *logical data item*, or *data item*, denoted X. A data item is stored in the DDBS as a set of *physical copies* or *copies*. The copy of X stored at site k is denoted x_k, and the fact that X has a copy at site k is denoted $x_k \in X$. We assume that the information regarding where the copies of data item X are located is available at least at the resident sites of X.

Users manipulate the database via *transactions*. A transaction is a program that accesses the database by issuing *logical operations READ* and *WRITE* on logical data items.

There are two major functional modules running at each site on behalf of the DDBS. The *transaction manager* (TM) supervises the execution of transactions and interprets logical operations into requests for *physical operations*. The *data manager* (DM) carries out the physical operations on the copies stored at the site. We assume that the DDBS runs a correct *concurrency control algorithm* which ensures *serializable* (SR) execution of transactions. We do not discuss transaction resolution upon site failures and recoveries in this paper, but assume that there is a correct protocol to take care of it. Hence, in the following discussions all transactions are atomic, i.e., they either meet their specifications or have no effect on the database at all.

Whether a transaction meets its specification depends on the interpretation of logical operations. For example, the strict *read-one/write-all* (ROWA) strategy can be described as

$$READ(X) = \vee \{read(x_k),\ x_k \in X\},$$

$$WRITE(X) = \wedge \{write(x_k),\ x_k \in X\},$$

where $OP = \vee \{op\}$ means that OP is interpreted as at least one of the op's, and OP fails if no op succeeds; and $OP = \wedge \{op\}$ means that OP is interpreted as all the op's, and OP fails if any one of the op's fails. Note that these notations are informal descriptions of the semantics of the logical operations, especially in presence of failure. They imply no implementation hints, such as parallel or sequential executions of op's, or the order of op's in case of sequential executions.

In a system using the strict ROWA scheme, site failures never result in inconsistent data. Consequently, the site recovery (in the sense of recovery of the database) is unnecessary. However, the degraded availability for write operations makes the strict ROWA scheme impractical. In this paper we introduce a revised scheme *read-one/write-all-available* (ROWAA). Intuitively, if a transaction knows that site k is down, it should not try to read a copy from site k, or send an update to site k. ROWAA not only saves the time otherwise wasted because of waiting for responses from an unavailable site, but also reduces the possibility of aborting or blocking transactions. As mentioned in the introduction, however, without some additional conventions user transactions may create an unrecoverable mess as they write to the "available" copies perceived by themselves. In the next section, we specify the conventions for user transactions in our ROWAA scheme, and a recovery algorithm for this scheme.

3. Basic Algorithm

3.1. Session Numbers and Nominal Session Numbers

As far as recovery is concerned, a site may be in three distinguishable states. We say a site is *down* if no DDBS activity is going on at the site. A site is *recovering* if it is in an early stage of its recovery procedure. Its TM and DM may have been turned on (for processing control transactions as described below) but are not yet ready to accept user transactions. The site is *operational* or, simply, *up* if both TM and DM at the site work normally. We also say a site is *not operational*, meaning that the site is either down or recovering.

The dividing point at which a site goes from down to recovering, or from up (or recovering) to down, is very clear. The point at which it goes from recovering to up depends on the recovery algorithm. We will define this point precisely later in this section. Note that in our algorithm some data copies may still be out-of-date when the recovering site enters the operational state. However, the out-of-date copies in an operational site must have been marked as unreadable, and will eventually be updated by copiers or user transactions.

An *operational session* of a site is a time period in which the site is up. Each operational session of a site is designated with an integer, *session number*, which is unique in the site's history, but not necessarily unique systemwide. If a site is not in an operational session, its session number is undefined. For simplicity of description, however, we say that the site has session number 0 if it is not operational (assuming 0 is never used as a session number for an operational session). The session number of site k is denoted as $as[k]$.

Because sites are up and down dynamically, it is not always possible for a site to have precise knowledge about the session number of another site. In order to have a consistent view of the session number of a particular site i in the system, we augment the database with additional data items, called *nominal session numbers*. We use the notation $NS[k]$ for the data item indicating the nominal session number of site k, and NS for the vector composing $NS[1]$, ..., $NS[n]$. note that the nominal session number of site k may differ from the actual session number $as[k]$, but the difference should be kept tolerable as far as possible.

The session number $as[k]$ can be implemented as a variable shared by the TM and DM at site k. That is, the TM and DM know the local session number precisely, and use it to control their services. For example, when a site starts recovery from an earlier failure, it turns on the TM and DM and loads its session number with 0 automatically. User transactions can not be processed at site k while $as[k]$ is 0. After the site finishes some necessary work (described in subsection 3.4), it loads a new session number into $as[k]$ and the site becomes fully operational. The current session number must also be saved in a stable storage so that the next time the site recovers, a new session number can be assigned correctly. In practice, session numbers can be recycled. Two different sessions can have the same session number as long as no single transaction is alive in both sessions.

In contrast, the nominal session numbers are data items similar to those in the database. They are readable by user transactions, and writable only by control transaction (described in subsection 3.3). These read and write operations on nominal session numbers are under concurrency control like other data items. Because the nominal session numbers are read very frequently (by user transactions) but only updated occasionally (when sites fail and recover), we assume they are fully replicated at all n sites. Since we use the capitalized letters to name logical data items and the corresponding lower-cased names with subscripts for their physical copies, the copy of $NS[k]$ at site i is denoted $ns_i[k]$, and the vector composing $ns_i[1]$, ..., $ns_i[n]$ is denoted ns_i. The value of ns_i is the status of the system as currently perceived by site i.

3.2. User Transactions

In the ROWAA scheme, user transactions must obey the following convention. Each user transaction implicitly reads the local copy of the nominal session vector prior to any other operations. This gives the transaction a view of "current" configuration of the system, which is used by the transaction throughout its execution. More precisely, if a transaction, initiated at site i, reads the nominal session vector ns_i, its logical operations are interpreted by the TM at site i as:

$$READ(X) = \vee \{read(x_k),\ x_k \in X \text{ and } ns_i[k] \neq 0\},$$

$$WRITE(X) = \wedge \{write(x_k),\ x_k \in X \text{ and } ns_i[k] \neq 0\}.$$

Each request for reading or writing a physical copy at site k carries $ns_i[k]$, the session number of site k perceived by the transaction. The DM at site k first checks this number against its actual session number, $as[k]$. If they are not equal, the request is rejected. Otherwise, the DM carries out the request. For a data copy marked as unreadable, a write operation upon it removes the mark when the transaction commits, while a request for reading it triggers a *copier transaction* that renovates the physical copy. The user transaction can either be blocked until the copier finishes, or may read some other copy instead. We consider this decision as an implementation issue rather than a part of the convention for user transactions, and hence leave it unspecified.

A copier transaction is responsible for refreshing a particular unreadable data copy. It reads (a copy of) the nominal session number, locates a readable copy, uses its content to renovate the local copy and removes the unreadable mark. If the copier cannot find a readable copy of this data item among the currently operational sites, this item is considered *totally failed*. A separate protocol is needed to resolve this problem, which is not discussed in this paper. Copier transactions may be initiated by the recovery procedure one by one for individual unreadable data copies, or on a demand basis, i.e., triggered when DM receives read requests for them. Such choices may influence the performance but not the correctness. In any case, copier transactions are executed concurrently with all other transactions after the recovering site has entered the operational state. They follow the concurrency control protocol like all other transactions.

3.3. Control Transactions

Now we discuss the possible transitions of nominal session numbers. We impose the restriction that any changes to the nominal session numbers must be done by a special kind of transactions, named *control transactions*.

There are two types of control transactions. A control transaction of type 1 claims that a site is nominally up. It can only be initiated by the recovering site itself when it is ready to change its state from recovering to operational. For example, when site k is ready to claim it is operational, it initiates a control transaction, that reads an available copy of the nominal session vector, say, $(ns_i[1], ..., ns_i[n])$, and refreshes its own copy of the nominal session vector $(ns_k[1], ..., ns_k[n])$, then it chooses a session number to be used for the next operational session and writes it to $ns_k[k]$ as well as $ns_j[k]$ for all $1 \leq j \leq n$ such that $ns_i[j]$ is non-zero. A control transaction of type 2 claims that one or more sites are down. Any site can initiate this type of transactions, as long as it is sure that the sites being claimed down are actually down. This requirement can be satisfied in systems where site failures are the only possible failures. A transaction of this type reads a copy (likely the local copy) of the nominal session vector, and writes 0 to all available copies of the nominal session numbers for the sites to be claimed down.

Control transactions, like all other transactions, follow the concurrency control protocol and the commit protocol used by the DDBS. A control transaction may be aborted due to a conflict with another one, or due to a write failure (e.g. another site failure occurs during the execution of the control transaction). One difference between control transactions and user transactions is that user transactions can be processed only by sites that are operational, while control transactions can be processed by recovering sites as well.

3.4. Site Recovery Procedure

The site recovery procedure proceeds as follows:

1. When a site k gets up, it turns on its TM and DM and loads its actual session number $as[k]$ with 0, meaning that TM and DM are ready to process control transactions but not user transactions.

2. The recovering site marks all data copies at its own site unreadable. Actually, only the data copies that have missed updates since the site failed need to be marked. There are different ways to identify such out-of-date items. We will discuss this issue in section 5.

3. After the step 1 and 2, the site initiates a control transaction of type 1. Note that the control transaction writes a newly chosen session number into $ns_i[k]$ for all operational sites i, but not $as[k]$.

4. If the control transaction in the step 3 commits, the site is nominally up. The site can convert its state from recovering to operational by loading the new session number into $as[k]$. If the step 3 fails due to a crash of another site, the recovering site must initiate a control transaction of type 2 to exclude the newly crashed site, and then try step 3 again. Note that the recovery procedure is delayed by the failure of another site, but the algorithm is robust as long as there is at least one operational site in the system.

4. Correctness

4.1. Correctness Concepts

One-serializability (1-SR) has been used as the correctness criterion for transaction executions in distributed database systems with replication. In this subsection, we briefly define the fundamental concepts that are necessary for the correctness proof presented in the next subsection. These concepts are based on [3, 8], but presented in an extendible way.

An *execution history* of a set of transactions $T = \{T_a, T_b, \cdots\}$ is a partially ordered set containing all operations of these transactions. An *augmented execution history* is a history with an initial transaction that writes to all data copies and a final transaction that reads from all data copies. Two augmented execution histories are *equivalent* if and only if they have the same *read-from* relations. To simplify our arguments, we consider only the augmented execution histories in this paper. The relation T_b *reads−x−from* T_a is denoted as $T_a \Rightarrow_x T_b$. A *serial* history is a history with total order such that the operations from different transactions are not interlaced. A history H is *serializable* if there exists a serial history, H_s, equivalent to H.

A *serializablility testing graph*, STG, of a history H is a graph $(T, \rightarrow)$ with the following properties:

(i) if $T_a \Rightarrow_x T_b$ then there exists an edge $T_a \rightarrow T_b$ (i.e., the graph contains all read-from edges);

(ii) there is an edge between any two transactions that write to the same data copy (called *write-order* edge, and a edge between two transactions writing to x is denoted as $\rightarrow_x$);

(iii) if $T_a \Rightarrow_x T_b$ and $T_a \rightarrow_x T_c$ then there is an edge $T_b \rightarrow T_c$ (called *read-before* edge).

Intuitively, to construct a STG for a history H, we start with the *read−from* relation graph of H, and arbitrarily add edges until the resulted graph satisfies the above properties. It is easy to see that the STGs are functionally equivalent to the bigraphs used in [8], and the main theorem on serializability theory can be stated as:

Theorem 1. A history H is serializable if and only if H has an acyclic STG.

One example of a STG is the *conflict graph* (CG), in which all transactions with conflicting operations (read-write or write-write on the same data copy) are $\rightarrow$ related according to the order in which their conflicting operations actually take place. Obviously, the CG of a history is one of its STGs, hence the histories with acyclic CGs are serializable. The set of histories with acyclic CGs are called DCP in [4] and DSR in [8].

If we modify the definition of STG by replacing the word "edge" by "path" in (ii) and (iii), the Theorem is still correct. We use the notation $T_a \rightarrow^* T_b$ for the fact that there is a path from T_a to T_b.

Now we generalize these concepts to distributed databases with replication. First we define the *READ−FROM* relations. Transaction T_b *READS−X−FROM* T_a, denoted $T_a \Rightarrow_X T_b$, if there is some $x_i \in X$ such that $T_a \Rightarrow_{x_i} T_b$. Two histories are *equivalent* if they have the same *READ−FROM* relations. A *one-copy serial* history is a serial history with all physical operations replaced by corresponding logical operations. A history H is *one-serializable* (1-SR) if there exists a one-copy serial history, H_{1s}, equivalent to H.

A *one-serializablility testing graph*, 1-STG, of a history H is a graph $(T, \rightarrow)$ with the following properties:

(i) if $T_a \Rightarrow_X T_b$ then there exists an edge $T_a \rightarrow T_b$ (read-from);

(ii) there is an edge between any two transactions that write to the copies of the same data item X (write-order, denoted as as $\rightarrow_X$);

(iii) if $T_a \Rightarrow_X T_b$ and $T_a \rightarrow_X T_c$ then there is an edge $T_b \rightarrow T_c$ (read-before).

As in STG, the word "edge" in (ii) and (iii) can also be replaced by "path". However, the CG of a history is not necessarily a 1-STG. The main theorem on one-serializability can be stated as:

Theorem 2. A history H is one-serializable if and only if H has an acyclic 1-STG.

In the presence of copier transactions, however, the conditions for 1-SR, in general, cannot be satisfied if we treat copier transactions in the same way as user transactions. For example, if we consider a copier T_c that refreshes a copy x_j as a writer to X, the requirement (iii) will rule out many correct executions. In order to include such correct histories in the class of 1-SR, we modify the definition of 1-STG by taking the semantics of copiers into consideration.

First, we define *READ−FROM* as a relation between a transaction and a non-copier transaction. That is, a transaction T_b *READS−X−FROM* a non-copier transaction T_a either directly, i.e. $T_a \Rightarrow_{x_i} T_b$ for a copy $x_i \in X$; or indirectly, i.e., there exists a copier transaction T_c such that $T_a \Rightarrow_X T_c$ and $T_c \Rightarrow_{x_i} T_b$ for a copy $x_i \in X$. Next, the one-copy serial history is defined as a serial history of the transactions, excluding copiers, with their physical operations replaced by logical operations. Finally, we modify the definition of 1-STG. In (i), we replace $T_a \Rightarrow_X T_b$ by $T_a \Rightarrow_x T_b$, because *READ−FROM* relations now do not reflect the *read−from* relations of copiers. In (ii), we change the word "transaction" by "non-copier transaction" because we are concerned only with the write order among non-copier transactions. Note that an indirect $READ-FROM$ relation now is a path rather than an edge. But adding a $READ-FROM$ edge will not change the acyclicness of the graph. Under these modification we find that the "if" part of Theorem 2 is still valid. That is,

Corollary. A history H is 1-SR if H has an acyclic 1-STG (under the revised definition).

It should be noted that all concepts in the theory of serializability are relative to the *database*, i.e. the domain containing all data items that transactions operate on. For example, the abstraction of a transaction in serializability theory is a sequence (or more general, a partially ordered set) of read/write operations upon the data items in the database. All other activities are ignored by this abstraction. Therefore, we can consider abstract transactions with respect to a particular subset of the database, meaning that only the operations upon data items within this subset are of concern. Similarly, we can consider all serializability concepts with respect to this subset. In our algorithm, the database, DB, is augmented by the nominal session numbers, NS. Hence, we can consider the abstract transactions with respect to DB, NS and $DB \cup NS$. For example, with respect to NS, all but control transactions are read-only transactions. Note that a correct concurrency control algorithm ensures serializability with respect to $DB \cup NS$, but what we really want is one-serializability with respect to DB.

4.2. Correctness Proof

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

Theorem 3. Based on the algorithm stated in the section 3, the conflict graph (CG) with respect to $DB \cup NS$ is a 1-STG with respect to DB.

Intuitively, the theorem implies that our algorithm, together with a concurrency control algorithm within the class of DCP, ensures correct executions of transactions.

Proof. We prove the theorem in two steps. First, we show that the CG with respect to $DB \cup NS$ embodies the write-order and read-before paths with respect to NS. Then, we use the result to prove that the CG also embodies the write-order and read-before paths with respect to DB.

Write-order and read-before with respect to NS. Recall that a control transaction of type 1 initiated by site k writes to all available copies of $NS[k]$ and brings the local copies of other nominal session numbers up-to-date. This control transaction is treated as a writer only to $NS[k]$, because to the other session numbers this transaction acts as a copier. The read-from relations are defined accordingly.

That is, we consider that a transaction reads $NS[k]$ from the control transaction that assigned the session number originally rather than from the one that renovates the local copy of the session number. Under this interpretation we can verify that any two control transactions writing to the same $NS[k]$ are connected via a CG path in which each pair of two contiguous vertices (control transactions) have intersected write sets, assuming that the system always has at least one site operational. Similarly for read-before relations. We omit the details here.

Write-order with respect to DB. Consider two transactions T_a and T_b, both writing to X. If their write sets intersect, the two transactions are connected by a CG edge. Otherwise, if T_a writes x_i but T_b does not write to x_i, they must read $NS[i]$ from different control transactions, say, $T_c \Rightarrow_{NS[i]} T_a$ and $T_d \Rightarrow_{NS[i]} T_b$. Because the CG embodies write-order and read-before paths with respect to NS, we can assume, without loss of generality, that $T_c \rightarrow^{*} T_d$, and conclude that $T_a \rightarrow^{*} T_d \rightarrow T_b$ ($\rightarrow^{*}$ stands for a path in CG).

Read-before with respect to DB. Consider the following cases. (i) $T_a \Rightarrow_{x_i} x_b$ and $T_a \rightarrow_{x_i}^{*} T_c$. Obviously $T_b \rightarrow T_c$ because CG is a STG. (ii) $T_a \Rightarrow_{x_i} x_b$ and $T_a \rightarrow_{X}^{*} T_c$, but T_c does not write to x_i. In this case we assume $T_d \Rightarrow_{NS[i]} T_a$ and $T_e \Rightarrow_{NS[i]} T_c$, where T_d and T_e are different control transactions. Based on our results with respect to NS, T_d and T_e are $\rightarrow_{NS[k]}^{*}$ related, and the only possibility is $T_d \rightarrow_{NS[k]}^{*} T_e$ (otherwise CG is cyclic). Since T_b reads x_i directly, T_b must see the same session number as T_a, hence T_b also reads from T_d. Then we have $T_b \rightarrow^{*} T_e \rightarrow T_c$. (iii) $T_a \Rightarrow_X T_b$ indirectly via copier transactions. For example, $T_a \Rightarrow_{x_i}$ a copier CP and $CP \Rightarrow_{x_k} T_b$, and $T_a \rightarrow_{X}^{*} T_c$ where T_c writes to some copy of X. We can apply the arguments in (i) and (ii) to the triples of T_a, CP, T_c, and CP, T_b and T_c, and conclude that $T_b \rightarrow^{*} T_c$. Similarly for the cases in which multiple copies are involved. □

5. Refinements

In the basic algorithm described in the section 3, we ignored the problem of identifying the data items that have missed updates due to site failure, and simply assume that all data at the recovering site are out-of-date. No particular mechanism has been built into our algorithm, but the algorithm is able to work with various methods to eliminate the unnecessary work.

One way to identify the data items that have missed updates is to use *fail-lock* [5]. Similar to a lock on a data item used in concurrency control algorithms to specify that the locked object is being used by a transaction, a fail-lock is used in recovery algorithms as the notion that the data item is being updated when a site is down. Our recovery algorithm can work with the fail-lock mechanism. When a site is recovering, it collects the fail-locks set during its failure, and marks the copies of fail-locked data items at the recovering site as unreadable.

Another practical mechanism is to use *missing list* (ML). Conceptually, a missing list is a two-dimensional array ML: $\{item\} \times \{site\} \rightarrow \{1, 0\}$, where $ML[X, i] = 1$ means x_i has missed updates. In order to save storage space, an ML can be implemented in various ways, for example, as a list of pairs (X, i) for non-zero elements in the ML. The elements of the ML can be seen as data items augmented to the database, but need be stored in volatile storage only. Access to elements should be under concurrency control. Each site maintains an ML. Unlike NS, MLs at different sites are considered as different data items, rather than copies of the same logical data. A pair (X, k) in ML at site i means that $x_i \in X$, $x_k \in X$, and x_k has missed an update which is done to x_i. Our algorithm can work with MLs as follows. A write operation $WRITE(X)$ writes to x_i for all $i \in X$ such that site i is nominally up. It removes (X, i), if any, from the MLs at the sites to which it writes a copy of X successfully, and adds (X, j) into these MLs for all j such that $x_j \in X$ and site j is not available for the transaction. When site i is recovering, it looks up the MLs at all operational sites. If (X, i) appears in an ML, site i removes the entry (X, i) from all MLs of nominally operational sites, and marks its own copy x_i as unreadable. Site i also forms its own ML using the entries (X, j), $i \neq j$, seen in the MLs at other operational sites. It should be noted that under this mechanism, as long as a site has an up-to-date copy of a data item, the ML of this site has the precise information on where the copies of the data item have missed updates. We will discuss further details in a future paper.

It should be noted that there is a tradeoff between the costs of the recovery procedure and the increased cost of normal operations caused by the use of mechanisms for identifying the out-of-date data items. In systems using version numbers or timestamps, even without identifying the out-of-date copies our basic recovery algorithm is not very expensive, because a copier can compare the version numbers or timestamps of the two copies first, then decide whether coping data is necessary.

6. Conclusion and Further Work

In this paper, we introduce a new algorithm for site recovery, including the conventions for user-transactions (ROWAA), and the recovery procedures.

This scheme provides a very high degree of availability. A logical read or write operation on a data item can succeed as long as one of its copies is in an operational site, and the site's session number is known by the transaction.

In this approach the extra cost to user transactions is negligible. Although all user transactions are required to read the local copies of the nominal states, there is little overhead because these reads do not conflict with each other. The control transactions which update the nominal session numbers are only necessary when sites fail or recover.

The ideas presented in this paper deal with the problem of failed site integration. We believe that the solution to the site failure problem and the concept of nominal session numbers are applicable to the merging of network partitions. Full details have not been worked out but the direction of research is outlined as follows.

The distinction between the problems of the network partition and site failure is clear. In a site failure problem, the operational sites in the system can assume that no activity occurs at the failed site. Thus the failed site needs to integrate with the rest of the system and obtain updates missed during its failure. This means that the integration is only required in one direction (from the failed site to the operational sites). In a network partition problem, the system may allow updates on different data items in different partitions. For example, updates can be allowed on data items holding true-copy tokens [7]. When two partitions merge, each partition needs to obtain missed updates from the other partition. This can be accomplished by integrating the sites of a partition one by one with the other partition. When a site obtains all updates from another partition, it is considered integrated in one direction. A site is fully integrated with another partition if the integration in both directions has completed. Two partitions are fully integrated when all sites in each partition have fully integrated. The integration in either direction follows a protocol similar to a failed site integration protocol discussed in this paper. The granularity at which the integration takes place is up to the implementation.

References

[1] R. Attar, P. A. Bemstein, and N. Goodman, ''Site initialization, recovery, and backup in a distributed database system,'' *IEEE Trans. Software Eng.*, vol. SE-10, No. 6, 645-650, Nov. 1984.

[2] P. A. Bemstein, and N. Goodman, ''An algorithm for concurrency control and recovery in replicated distributed databases,'' *ACM Trans. Database Syst.*, vol. 9, No. 4, 596-615, Dec. 1984.

[3] P. A. Bemstein, and N. Goodman, "The failure and recovery problem for replicated database,'' *Proc. of the second ACM Symp. on Principles of Distributed Computing*, 114-122, Aug. 1983.

[4] B. Bhargava, and C. T. Hua, ''A causal model for analyzing distributed concurrency control algorithm,'' *IEEE Trans. Software Eng.*, vol. SE-9, No. 4, 470-486, July 1983.

[5] B. Bhargava, ''Fail-lock for a consistent database recovery during multiple failures,'' working paper, Purdue University, Sept. 1985.

[6] M. M. Hammer and D. W. Shipman, ''Reliability mechanism for SDD-1: A system for distributed databases,'' *ACM Trans. Database Syst.*, vol. 5, No. 4, 431-466, Dec. 1980.

[7] T. Minoura and G. Wiederhold, ''Resilient Extended True-Copy Token Scheme for a Distributed Database System,'' *IEEE Trans. Software Eng.*, vol. SE-8, No. 3, 173-188, May 1982.

[8] C. H. Papadimitrou, ''Serializability of concurrent updates,'' *JACM*, vol. 26, 631-653, Oct. 1979.

[9] D. Skeen, and M. Stonebreaker, ''A formal model of crash recovery in d distributed system,'' *IEEE Trans. Software Eng.*, vol. SE-9, No. 3, 219-227, May 1983.

[10] D. Skeen, ''Nonblocking commit protocols,'' *Proc. 1981 ACM-SIGMOD Conf. Management of Data*, ACM, New York, 133-147.