Achieving Robustness in Distributed Database Systems

DEREK L. EAGER AND KENNETH C. SEVCIK
University of Toronto

The problem of concurrency control in distributed database systems in which site and communication link failures may occur is considered. The possible range of failures is not restricted; in particular, failures may induce an arbitrary network partitioning. It is desirable to attain a high "level of robustness" in such a system; that is, these failures should have only a small impact on system operation.

A level of robustness termed *maximal partial operability* is identified. Under our models of concurrency control and robustness, this robustness level is the highest level attainable without significantly degrading performance.

A basis for the implementation of maximal partial operability is presented. To illustrate its use, it is applied to a distributed locking concurrency control method and to a method that utilizes timestamps. When no failures are present, the robustness modifications for these methods induce no significant additional overhead.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design; H.2.4 [**Database Management**]: Systems—*distributed systems*; H.2.7 [**Database Management**]:Database Administration

General Terms: Performance, Reliability

Additional Key Words and Phrases: Concurrency control, serializability, robustness, network partitioning

1. INTRODUCTION

Developments in technology have made practical the interconnection of a large number of computer systems to form a computer network. The problem of distributing a database among the different computer systems, or *sites*, to form a *distributed database system* is an active research area. One topic that has received much attention is the design of *concurrency control methods* which permit multiple users to access and modify a distributed database concurrently.

A concurrency control method views a database as a collection of *entities*. In a centralized database the value of each entity is recorded but once, while in a distributed database the value of an entity may be recorded in copies at multiple sites. The *state* of a distributed database is given by the values of all of its entity copies. A distributed database is in a *consistent* state if (1) all of the copies of

Authors' address: Computer Systems Research Group, University of Toronto, Toronto, Canada M5S-1A4

each entity have the same value (the *entity value*), and (2) the entity values satisfy a set of *assertions* or *consistency constraints.* Thomas has called these two requirements *mutual consistency* and *internal consistency*, respectively [28]. The purpose of the concurrency control component of a database system is to ensure that user transactions against the database "see" consistent database states.

Additional complexity is added if the distributed database system is required to be *robust.* A robust system permits some query and update activity, preserving database consistency, even when system components fail. Robustness is desirable in a distributed system owing to the large number of components these systems contain, and the improbability that all of these components will always be operative. The fewer the restrictions on transaction processing that are imposed when failures occur, the higher the *level of robustness* against failures is said to be.

This paper is concerned with modifying known concurrency control methods to attain a high level of robustness against arbitrary site and communication link failures, without significantly impacting system operation in the absence of failures. There are two major problems that are encountered when this modification is attempted. The first concerns the management of updates to replicated entities. In general, it is desirable to allow a transaction to update an entity even though some of the entity copies may be unavailable owing to failures. However, since transactions must see consistent states, it is necessary to place some restrictions on when this is allowed. Also, the entity update must eventually be applied or accounted for at those copies that do not initially receive it. The second major problem concerns the maintenance of the concurrency control method machinery in the presence of failures. A concurrency control method must often impose access restrictions on entity copies to prevent the observation of inconsistent states. Failures may prevent these restrictions from being removed, thus impeding transaction processing.

This paper presents a methodology that can be used in solving these problems. In Section 2, a model of concurrency control is introduced. Section 3 discusses the two major consequences of site and communication link failures: network partitioning and dangling precommits. On the basis of the characteristics of these consequences, the highest attainable robustness level that does not significantly degrade system performance in the absence of failures is identified. A basis for the implementation of this robustness level is presented in Section 4. The paper concludes with two applications of the basis, one to a distributed locking method and one to a concurrency control method that utilizes timestamps.

1.1 Related Work

Numerous concurrency control methods appear in the literature (for descriptions of many of these methods, see the surveys of Bernstein and Goodman [3] and Hsiao and Ozsu [15]). However, relatively few of these methods include a careful treatment of failures. Even when such a treatment has been attempted, the resulting method has attained a lower level of robustness than one would prefer.

The method proposed by Montgomery [20] uses transaction preanalysis to develop a hierarchical locking scheme. By allowing an entity copy to have several values, *polyvalues*, it can be guaranteed that an entity copy will not be made locally inaccessible by any site or communication link failure other than a failure of the site that stores the copy. However, owing to their effect on the locking hierarchy, site failures potentially have a major impact on the updating of data at remote sites.

The *majority consensus algorithm* of Thomas [28] requires that each site have a complete copy of the database. Database sites vote on the acceptability of update transactions. An update transaction requires the acceptance of only a majority of the sites, while queries can be processed locally. The majority consensus algorithm attains a high level of robustness, in that only a majority of the network sites need to be operational for an update transaction to complete. As noted by Thomas, however, it is possible that a query may see an inconsistent database state. In some applications this may be undesirable, and when the method is modified so that consistent query execution is guaranteed, as in [6], component failures significantly reduce the extent to which database activity can continue.

The centralized locking method proposed by Menasce [19] uses local lock controllers at each site to provide backup in the event of a central site failure. However, unlike the situation in the majority consensus algorithm, an update transaction must be able to access all of the copies of the entities it will update. After the failure of a site, or the failure of the communication links to a site, those transactions that must update entities with copies stored at the inaccessible site cannot be executed.

There have also been several studies that focus on one or both of the two major robustness problems mentioned previously. Gifford [12] proposes a strategy for managing updates to replicated entities in the presence of failures. However, the restrictions on transaction processing are unnecessarily strong, resulting in a lower robustness level than that which is described here. Also, performance is impacted even in the absence of failures. Badal [2] describes alternative strategies for managing updates that place only minimal restrictions on transaction processing. However, when failures are present, the observation of database inconsistencies is not always prevented; this is undesirable in many applications.

Both Hammer and Shipman [14] and Skeen [23, 24] have proposed modifications to the method of transaction "commitment" that allow a high robustness level to be achieved; with regard to some failure types these modifications provide more protection than those proposed here. However, system performance is significantly impacted even in the absence of failures, owing to the number of additional messages that must be sent whenever a transaction commits.

2. A MODEL OF CONCURRENCY CONTROL

2.1 A Model of Transaction Processing

Numerous transaction processing models have appeared in the literature; the one used here is adapted from that used by Bernstein and Goodman [3]. In this model, a transaction is executed (initiated, terminated, etc.) by its *home site*. The home site is responsible for assigning a unique *identifier* to each of its transactions. The degree of control the home site has over its transactions varies among concurrency control methods. However, it is assumed that each site has the ability to be a home site; in particular, each site is assumed to have a catalog indicating, for each entity, the set of sites at which the entity is stored. The home site of a transaction T will be denoted by H(T).

A transaction T is modeled as a sequence of *read* and *write* operations. The *read set* of T is defined as the set of entities that T reads. Similarly, the set of entities that T writes is termed its *write set.* It is assumed that the write set of T is included in its read set. When T is initiated, a *private workspace*, possibly distributed among various sites, is created. Read operations of T cause database read operations only if the required data are not in T's workspace. Similarly, write operations only affect the workspace of T.

Once T completes its processing, it must be determined whether or not the updates of T can be installed in the database without violating consistency requirements. In many concurrency control methods, including both of the methods used as examples in this paper, this is done in two phases. In the first phase, an *intention-to-update* operation initiated by H(T) is performed on each copy of each entity in the write set of T. The concurrency control component at each site where one or more of these operations are performed either *accepts* or *rejects* each operation (possibly after a queuing delay), depending on whether the update can be performed without violating the constraints of the particular concurrency control method being used. If acceptance occurs, the site sends an acknowledgment to H(T) and restricts access, to some degree, to the entity copy. H(T) is also notified when an operation is rejected.

Depending in part on the replies to any intention-to-update operations that were performed, H(T) decides to *restart, abort,* or *commit* T. Restart procedures entail rerunning T after assigning it a new identifier. It would be necessary to abort T if system conditions were such that it could not be successfully completed. For example, data required by T could have been made unavailable by system failures. If T is restarted or aborted, *release-intention-to-update* operations are sent to the affected sites. If T is committed, the second phase is initiated.

Once committed, T cannot be restarted or aborted. At this point the database system is certain that any user output is correct and that any transaction updates will be applied (possibly after a delay if failures have occurred). H(T) can therefore inform the issuer of T that T has completed processing and can return any user output. Instead of transmitting each actual database write operation explicitly, H(T) sends a *commit* to each site that received an intention-to-update operation. Upon reception of a commit, the updates of T are permanently installed in the local database and the access restrictions imposed on the associated entity copies are removed. The updates of T are then said to be *committed* at that site. Note that, because of failures, there may be arbitrarily long time delays between the commitment of a transaction and the subsequent commitment of some of its updates.

This two-phase structure can be integrated into the *two-phase commit* (2PC) protocol [13, 17] with no significant performance cost over that when 2PC is not used. This protocol is a well-known method of implementing atomic updates. In this integrated procedure, *precommit* operations replace intention-to-update operations; in addition to performing the function of an intention-to-update, a precommit causes the new value of the entity copy to be written onto secure storage. It is assumed that the new value cannot be read until the transaction has been committed. The second phase of 2PC requires commits to be sent exactly as before (although in 2PC these commits need not contain the actual entity values, as they have already been written onto secure storage at the receiving sites).

2.2 A Model of Concurrency Control Correctness

This section first reviews some of the standard concepts relating to concurrency control correctness. (For more details, see the papers by Bernstein, Shipman, and Wong [5] and Kung and Papadimitriou [16].) These concepts are then modified as necessitated by our more complex system model in which failures may occur.

A concurrency control method is said to *maintain consistency* if, when a set of transactions is processed, the same user output and the same database state result as if the transactions were processed sequentially. A concurrency control method is said to be *correct* if it maintains consistency while at the same time ensuring that each transaction is either committed or aborted (possibly after a number of restarts) within a finite time after being initiated.

In proving that a method maintains consistency, the "precedes" or $\rightarrow$ relation is useful. This relation is based on the notion of *conflicting* operations, which are database (read/write) operations produced by different transactions, at least one of which is a write, that operate on the same entity copy. A transaction T_1 *precedes* a transaction T_2 ($T_1 \rightarrow T_2$) if an operation of T_1 precedes and conflicts with an operation of T_2 (in which case T_1 and T_2 are also said to be *conflicting*). If the concurrency control method executes transactions so as to ensure an acyclic $\rightarrow$ relation among committed transactions, consistency is maintained. Under certain reasonable assumptions about the information available to the concurrency control method [16], an acyclic $\rightarrow$ relation is necessary as well as sufficient. In this paper the maintenance of an acyclic $\rightarrow$ relation is considered equivalent to the maintenance of consistency.

In an environment with failures it is useful to modify this standard definition of $\rightarrow$ in two ways (neither of which changes its relationship to the consistency maintenance property). First of all, suppose that an operation of a transaction T_1 precedes and conflicts with an operation of a transaction T_2. It is technically useful if the $T_1 \rightarrow T_2$ relation is implied only if the conflict is *direct*; that is, there is no operation of a committed transaction that conflicts with both of these operations and that has occurred between them. Second, consider the situation in which an entity copy has not been updated by one or more transactions because the entity copy was unavailable to these transactions owing to failures. In this case the $\rightarrow$ relation is constructed as if the updates missed had actually been performed on the entity copy (our restrictions on transaction processing will ensure that there is a well-defined ordering among these updates). This last modification is necessary since it is assumed that failures may be corrected at any time, and the updates that were missed then applied.

It is also useful to differentiate between the conflicts that lead to $\rightarrow$ relationships [5]. If a read operation of a transaction T_1 and a conflicting write operation of a transaction T_2 results in a $T_1 \rightarrow T_2$ relationship, then $T_1 \rightarrow_{RW} T_2$. "$T_1 \rightarrow_{WR} T_2$" and "$T_1 \rightarrow_{WW} T_2$" are defined similarly.

3. THE ROBUSTNESS PROBLEM

Our model of concurrency control has been presented independently of any particular concurrency control method. In this section it is our aim to similarly present the basic processing restrictions that network failures must induce, whatever the concurrency control method in use. Section 4 discusses how these restrictions can be implemented, again without loss of generality. In Sections 5 and 6, two specific concurrency control methods are used to illustrate these general concepts.

3.1 The Network Partitioning Problem

A *physical component* of a site is defined as that set of sites with which the site can communicate. A *network partitioning* has occurred if some site's physical component does not include all network sites owing to site or communication link failures. It is impossible, in general, to distinguish between site failures (in which all database activity at the site ceases until recovery from the failure) and link failures (in which a site must assume that database activity is continuing at one or more sites outside of its physical component) [19, 26]. Because failures of communication links, communications front-end machines, and even host communications software all constitute "link failures," as described above, it can only be safely assumed that database activity at a site has ceased if the site has sent positive confirmation of that fact. In this case, the surviving sites can proceed as if they constituted the entire network. The failed site must then be integrated back into the system as part of its recovery. This integration can be done in a system implementing the procedures of Section 4, in a fashion similar to that in which changes in "voting rules" are handled by Gifford [12]. An alternative procedure has been proposed by Attar, Bernstein, and Goodman [1].

A solution to the network partitioning problem provides processing restrictions on when a transaction can be allowed to commit, given that some of its write set entity copies are unavailable owing to a partitioning. The purpose of these restrictions is to ensure that consistency is maintained, while allowing as much transaction processing as possible. The restrictions used by Garcia-Molina [10] and Stonebraker [26] require that a transaction have available a majority of the copies of each read and write set entity. Gifford uses essentially the same requirement, but generalizes the concept of a majority to that of a *quorum* [12]. Under this generalization a transaction requires a "read" quorum of the copies of each read set entity and a "write" quorum of the copies of each write set entity. The essential property of Gifford's quorums is that every possible read quorum on an entity must have at least one copy in common with every possible write quorum on the entity. Our restrictions differ from those of Gifford's in that ours will usually allow a transaction to proceed without read quorums. For this reason, the distinction between read and write quorums is much less useful and has not been made here, although such a distinction could be introduced. It is sufficient for our purposes to define a *quorum* of entity copies as a set of entity copies such that any two quorums on a particular entity must intersect. In general, a weighting can be assigned to each copy of an entity. A set of copies of a particular entity is a quorum if the sum of the weightings of the copies in the set is more than half the total weighting for the entity. The quorum concept encompasses a broad spectrum, ranging from a "primary copy" requirement to a simple majority requirement.

A processing restriction requiring a quorum of the copies of each read and write set entity ensures that the last committed value (version) of an entity is always accessible. This is sufficient, since the concurrency control method can essentially present the illusion that there is only one "logical" site at which the entities are stored, hiding the effects of any partitioning that separates the actual sites. However, the requirement is not necessary, as it is possible to maintain consistency even when transactions are allowed to "run in the past," without access to newer entity values. To see why this ability is useful, consider a situation in which a site is partitioned from all other sites. The ability to run in the past allows queries to execute at the site, even though updating may be taking place in the rest of the network.

Intuitively, any transaction processing restrictions that allow running in the past must satisfy the following conditions:

(a) It must always be possible to determine which of the values of a particular entity is most current.
(b) Any transaction running in the past (using an old entity value) must not be directly or indirectly preceded by any transactions that have seen the future (have seen the new entity value).

Condition (a) ensures that after failures are corrected, there is always a well-defined order of missing update applications. This well-defined order also ensures that our modified definition of $\rightarrow$ is unambiguous. For condition (b) consider a transaction T_n that has read an old value of entity x (copy x_A). Owing to our modified definition of $\rightarrow$, T_n precedes the transaction T_1 that created the new value of x. Formally, condition (b) prohibits the existence of a direct or indirect precedence between T_1 and T_n ($T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$), preventing a cycle in the $\rightarrow$ relation. The detection and prevention of this precedence is the major problem in the design of transaction processing restrictions that are necessary to satisfy these conditions. Our solution can be intuitively described as requiring T_1 to "pass" the fact that it could not successfully update x_A down the chain of $\rightarrow$ relationships. When this information is received by T_n, T_n must be restarted or aborted, since it has read the obsolete entity value. The necessary restrictions on transaction processing will now be described in detail.

First of all, a transaction must precommit a quorum of the copies of each entity in its write set. This requirement is necessary since otherwise two different updates of an entity could be committed at the same time on disjoint sets of copies, causing a cycle in the $\rightarrow$ relation and violating condition (b). (In some concurrency control methods this requirement is also needed to ensure that condition (a) can be satisfied.) The remaining restrictions are needed to allow identification of read operations that violate condition (b). These restrictions require the concepts of *missing updates, missing update awareness* and *missing update information.* When a transaction T_1 cannot update an entity copy because of failures, the corresponding update is said to be *missing.* T_1 becomes *aware* of the missing update (acquiring *missing update information*) when its precommit is unsuccessful (the precommit acknowledgment might be timed out as a result of a network partitioning, for example). T_1 is required to pass all of its missing update information to any other transaction T_2 such that $T_1 \rightarrow T_2$ (a successful "pass" is required only if both T_1 and T_2 are committed at some point). T_2 would then be under a similar requirement to transfer all of its missing update information (which includes that received from T_1) to each transaction that it precedes. The result of this transfer of information, assuming that all transactions are eventually committed, is that a transaction T_n such that $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ becomes aware of all the unapplied missing updates of which $T_1, T_2, \ldots, T_{n-1}$ are aware. (It will be seen in Section 4 that once a missing update has been applied, the pertinent missing update information can be discarded.) The crucial final point is that a transaction aware of an unapplied missing update for an entity copy that it read is prohibited from being committed, satisfying condition (b).

Missing update information is transferred from transaction to transaction by posting it, together with sufficient information to determine to which transactions it should be transferred, at network sites. This level of indirection is necessary, since a transaction cannot know in advance which particular transactions it will precede. All that is known is which database operations will cause such a precedence. Consider a transaction T_1 that is aware of a missing update. To ensure that its missing update information is passed to each transaction T_2 such that $T_1 \rightarrow_{WR} T_2$, this information must eventually be posted at each site storing a copy of a write set entity. However, since a transaction's uncommitted updates cannot be read, posting need be performed only when the update is committed. The commitment of the transaction itself is not affected by this requirement, in that it may commit even if some of these sites are inaccessible owing to failures. (Recall that a transaction is committed prior to its updates being committed.) It need only be ensured that an update is committed only when the necessary information has been posted (perhaps as part of a missing update application procedure if failures have occurred).

On the other hand, consider a transaction T_2 such that $T_1 \rightarrow_{RW} T_2$ or $T_1 \rightarrow_{WW} T_2$. Since T_2 must precommit only a quorum of the copies of each of its write set entites, T_1 must post its information at sites storing a quorum of the copies of each of its read and write set entities. This must be done before a transaction such as T_2 performs a conflicting precommit, which, depending on the concurrency control method involved, may be before the commitment of T_1. T_1 may therefore have to post prior to commitment, with commitment conditional on whether its posting is performed successfully. The processing necessitated by posting prior to commitment and the details of missing update information management are presented in Section 4.

3.2 The Dangling Precommit Problem

The possibility of site and communication link failures during the 2PC procedure is the source of the *dangling precommit* problem. Recall that when a site accepts a precommit, it is making a pledge to accept an update for the precommitted entity copy. The site must then restrict access to the entity copy, in a manner depending on the concurrency control method, until a transaction commit or a release-precommit arrives. However, if a partitioning occurs, the site may not receive a transaction commit or release-precommit within an acceptable time period. For example, the transaction's home site may fail after sending the precommit to the site but before sending the corresponding commit. The partitioning would probably be detected by the site unsuccessfully attempting to communicate with the home site. This communication would be attempted after the precommit has been present for some maximum time interval.

Depending on the form of the partitioning, the site may have no way of telling whether to commit the update (apply the precommitted update and remove the access restrictions) or reject the update (remove the access restrictions). This is demonstrated in the case in which the site is partitioned from all other sites. The site cannot commit the update, since the corresponding transaction may have been restarted or aborted. It cannot reject the update, since some other site may have received a transaction commit. The site has no choice but to continue to restrict access to the entity copy.

The unsolvability of the dangling precommit problem (even through some finite length extension of 2PC), in environments where communication link failures may occur, has been proved formally by many authors (e.g., Gray [13] and Montgomery [20]). In this section, procedures that alleviate this problem without significantly impacting performance in the absence of failures are considered.

Suppose that a dangling precommit of a transaction T is left at a site A owing to a network partitioning. Consider the three possible environments of A: (1) none of the sites in its physical component have received information on the final status of T (committed, aborted or restarted), but all of the necessary precommits for entity copies stored in this component have been received; (2) at least one entity copy in its physical component must be precommitted before T can be committed, but such a precommit has not arrived; and (3) at least one site in its physical component knows the status of T.

In the first case, A cannot learn whether T has been committed, and has no choice but to continue the access restrictions imposed by the precommit of T. In the second case, A can determine that T has not been committed and can reject the update. Site A must first ensure that any precommits for T received by the affected sites will be rejected. In the third and final case, A can determine the status of T and can therefore act accordingly.

Assuming that these cases can be distinguished (which is demonstrated in Section 4), it should be possible to resolve most dangling precommits. For example, consider the failure of a site A in an otherwise fault-free network. Unresolvable dangling precommits would be left only by those site A transactions for which all precommits have been transmitted, but for which no commitment (or restart or abort) messages have been transmitted.

3.3 Maximal Partial Operability

This section characterizes the highest level of robustness that can be attained without significantly impacting performance in the absence of failures, given our models of concurrency control and robustness. A concurrency control method that attains this level is said to attain *maximal partial operability*.[1] In Section 4,

[1] This terminology is derived from the term *partial operability*, which is introduced by Montgomery [20].

it is shown that maximal partial operability can be attained while preserving concurrency control method correctness.

The restriction that performance not be impacted in the absence of failures is reflected in the following assumption regarding the manner in which transactions interact. Each transaction is viewed as normally interacting with its environment only through database read operations, precommits, and commits. Only when a transaction becomes "aware" of failures is it permitted the overhead of additional interaction (such as special posting operations). This assumption is central to the following assertion, which characterizes maximal partial operability based on the previously discussed strategies for dealing with dangling precommits and network partitioning.

ASSERTION 3.1. *The following two restrictions on transaction processing specify a limit on the robustness level that can be attained against site and communication link failures without impacting performance in the absence of failures.*

Restriction 1. Suppose that site and/or communication link failures create a dangling precommit of a transaction T_1 at a site A. In addition, suppose that A is unable to determine the final status of T_1 from the sites in the physical component of A and that all of the necessary precommits for entity copies stored in this component have been received. The first restriction on transaction processing then prohibits the removal of the access restrictions imposed by the dangling precommit.

Restriction 2. The second restriction is on transaction commitment. Each transaction T_1 is required to have chosen a quorum of the copies of each write set entity. If T_1 is aware of missing updates, a quorum of the copies of each read set entity must also have been chosen. T_1 may only be committed if the following conditions are met: (1) precommits have been acknowledged for each entity copy in the chosen write set entity quorums; (2) it has been ensured that any missing update information of T_1 will be transferred to any following transaction T_2 ($T_1 \rightarrow T_2$) that reads an update of T_1 ($T_1 \rightarrow_{WR} T_2$) or that performs a precommit on an entity copy in one of the chosen read or write set entity quorums ($T_1 \rightarrow_{RW} T_2$ or $T_1 \rightarrow_{WW} T_2$); and (3) there are no unapplied missing updates for any entity copy read by T_1 as far as T_1 is aware.

DISCUSSION. Restriction 2 warrants further comment. Condition (1) of Restriction 2 is necessary to prevent concurrent updates of the same entity on disjoint sets of entity copies. Consider conditions (2) and (3). The robustness level is maximized when only that processing resulting in a cycle in the $\rightarrow$ relation (due to weak entity copy availability requirements) is prohibited. It is a consequence of our assumptions regarding transaction interaction that $\rightarrow$ relationships can be recognized only by posting (or an equivalent operation). It is important to note in this regard that although some transactions may be "aware" of failures, many others may not be and cannot, therefore, take part in any special form of interaction. Condition (2) is necessary to ensure that posting is carried out correctly, while condition (3) is necessary to break detected cycles.

One useful measure of a robustness level is the effect of a failure of an arbitrary database site on a database system implementing the robustness level. For maximal partial operability, the entities made unavailable for updating are those for which a quorum was lost. The entities made unavailable for reading by a transaction that will generate missing updates are at worst the same as for updating. This is also true for any transaction T_n such that $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ and missing updates of T_1 remain unapplied. For any other transaction, only thoses entities for which the last available copy was lost are made unavailable. In the Appendix, three simple examples are presented that illustrate how the robustness extensions suggested in this paper permit the successful completion of transactions that would otherwise be blocked or aborted.

4. A BASIS FOR IMPLEMENTING MAXIMAL PARTIAL OPERABILITY

The previous section presented two restrictions to be enforced by the concurrency control method. In this section, a basis for the implementation of these restrictions is developed and shown to preserve concurrency control method correctness. This basis is not unique, in that we make a number of low-level design choices which reflect a trade-off between transaction processing ability (although the basic restrictions remain the same) and performance in the presence of failures.

With our design choices each site must maintain information of several types. (The quantity of information depends on the number of failures that have occurred and the rate of system activity.) The files that must be kept at each site are the following:

(a) a *missing-update-postings* file that contains posted missing update information;
(b) a *missing-update-value* file that contains values of missing updates (needed after failure recoveries);
(c) a *transaction-status* file that indicates the transaction restarts, aborts, or commits of which the site has knowledge;
(d) a *missing-update-applications* file that provides a record of the missing updates that have been applied at the site.

The following sections describe the contents of these files in more detail, and the manner in which file entries are created, deleted, and transferred from one site to another. This description is organized around the basic processing tasks that require these files.

4.1 Transaction Commitment

Transaction commitment is necessarily more complex in the presence of failures than in the basic transaction processing model. This complexity is incurred when a transaction is aware of missing updates. Otherwise, commitment is exactly as in the basic model.

The case in which a transaction T_1 becomes aware of missing updates for an entity copy that it read is illustrated in Figure 1. Consider the case in which T_1 is aware of missing updates, but none apply to any of the entity copies that it read. If the concurrency control method under consideration permits the conflicting precommit of a transaction T_2 where $T_1 \rightarrow_{RW} T_2$, to be performed before T_1

Context:

A transaction T_1 becomes aware of missing updates for an entity copy that it read. The site A storing the entity copy is required to determine if the missing updates have been applied (how this is done will be specified in section 4.3).

Case I - The missing updates have been applied:

T_1 can continue processing. As will be seen, the properties of the missing update application procedure ensure that T_1 must have read the current value.

Case II - The missing updates have not been applied, and cannot be obtained:

T_1 must be aborted.

Case III - The missing updates have not been applied, but can be obtained:

T_1 must be restarted after A has applied the most recent of the received updates.

Fig. 1. The processing necessitated by missing update information for a read entity copy.

For each missing update, the following information is required:

(1) The entity copy identifier (this identifier is assumed to also specify the site at which the copy is stored).

(2) The identifier of the transaction that produced the missing update.

(3) A specification of conflicting operations that should cause a transfer of (1) and (2) above. Many of these operations can be specified implicitly, such as read or precommit operations on entity copies that are currently precommitted by the transaction. Each operation is specified by the appropriate entity identifier and by a flag indicating whether only precommits or both reads and precommits should cause the information transfer.

Fig. 2. The contents of supplemented missing update information.

commits, then the missing update information of T_1 must be posted at sites storing a quorum of the copies of each read set entity before T_1 is committed. Correspondingly, if the concurrency control method allows a $T_1 \rightarrow_{ww} T_2$ conflict to develop prior to the commitment of T_1, the missing update information must be posted at sites storing a quorum of each of the write set entities before T_1 is committed. In order to minimize the total amount of information transferred, each site might record all knowledge it gains of the information posted at other sites.

A *missing-update-data* message is sent to each site at which posting must be performed prior to commitment. This message contains supplemented missing update information, as shown in Figure 2. In all the cases where posting is not required until commitment (for example, posting needed to pass information to $\rightarrow_{WR}$ conflicting transactions), any appropriate supplemented missing update information may be included in the messages that indicate the commitment of T_1.

Missing-update-data messages require acknowledgment. A site receiving a missing-update-data message of T_1 must ensure that each entity copy on which a conflicting precommit is possible has not been precommitted or updated by some other transaction that T_1 precedes. The procedure for checking this is one of the two procedures in the basis that can be specified only within the context of a particular concurrency control method. If such a precommit or update has been performed, the site includes a *copy-lost* flag for this entity copy in its missing-update-data message acknowledgment (either explicitly or implicitly indicated by other information returned). The reception of a copy-lost flag for an entity copy or the time-out of an acknowledgment from the site storing the copy requires that an alternate copy be selected for quorum membership. If T_1 ever loses the ability to obtain a quorum for an entity, T_1 must be aborted (or restarted if it is believed that the failures are transient). Otherwise, T_1 is committed when it receives the necessary missing-update-data message acknowledgments.

4.2 Transferring Missing Update Information

A site stores posted missing update information in its missing-update-postings file. This file is organized on the basis of the conflicting operation specifications in supplemented missing update information. When a site processes a read or precommit operation of a transaction T_1, it checks for the presence of missing update information that should be transferred to any transaction performing such an operation. If present, the site sends back the corresponding missing update information with its response to the operation request. It should be noted that the missing-update-postings file need be checked only if it is not empty. Also, the "granularity" of this file implies, for example, that network failures directly affecting only one database application will not impact any other application that operates on a disjoint set of entities.

A site may receive missing update information for an entity copy stored at the site. In such a case the site should try to retrieve the appropriate updates using the procedures in the next section (if these updates are indeed unapplied). A special case occurs when a transaction T is aware of missing updates for an entity copy on which a precommit of T has been acknowledged. When the site storing the entity copy commits the update of T, the missing update information of T will have been posted at the site. In this case, even though the site does not have the missing update values, it can consider the missing updates as being applied just prior to its application of T's precommitted update, since the effect on the database is the same. However, in this case the update of T can be applied only when it would be permissible to apply a missing update (see the following section).

4.3 Missing Update Application

As noted in Section 3.1, it must be ensured that a missing update is applied at a site only after the missing update information of the transaction involved has been posted at the site. It is the responsibility of the sites at which the entity update has been applied to retain the missing update information, as well as the missing update values, for any sites at which the update may not have been applied owing to failures.

In the commit message sent to a site *A* by a transaction *T*, H(*T*) includes the identifiers of those sites that failed to acknowledge precommits for the entities that will be committed at *A*. Any missing update information that would normally have been sent to these sites is also included. This information is used by *A* to maintain its missing-update-value file. This file stores the entity copy identifiers and values of the missing updates, the identifier of *T*, and the missing update information for the sites involved. This information is also stored in the corresponding file at H(*T*). Note that the value of a missing update is stored at a site only if the site stores another copy of the entity involved or if it is the home site of the transaction that produced the update. A missing-update-value file may have more than one value (for different transactions) for a particular entity copy identifier. In such a case these values are maintained in the order of reception.

Suppose that the updates and the missing update information corresponding to a particular entity are retrieved from a missing-update-value file at a site *B* and sent to a site *A*. Assume that some of these missing updates apply to a copy of the entity that is stored at *A*. The missing-update-applications file at *A* records the entity identifier and the transaction identifier of each missing update that has been applied at the site. Using this file, *A* checks to see if the most recent of the received updates for the entity copy stored at *A* has been applied. If it has, *A* instructs *B* to remove the appropriate entries from its missing-update-value file.

Otherwise, *A* should apply the most recent applicable update. Before doing so it must ensure either that no uncommitted active transaction has read the old entity copy value or that applying the update when such a transaction exists cannot cause a cycle in the $\rightarrow$ relation. The implementation of this last step is the second (and last) basis procedure that can only be specified in the context of a particular concurrency control method. Also, if there is any uncertainty about the order of application with respect to any precommits held on the entity copy, *A* must determine the status of the owning transactions. If a transaction with a precommit has not yet committed, then its precommit does not affect missing update application. If *A* finds that a transaction with a precommit is committed, it will obtain the transaction's missing update information (see Section 4.4). In conjunction with the *completeness property* of missing-update-value files illustrated in Figure 3, and a corresponding property for the missing update information of a committed transaction, this implies that the order of application of the committed precommit can be determined.

When these steps are completed, *A* processes the missing update information (by adding any necessary entries to its missing-update-postings file), applies the most current update, and adds all those previously unapplied received missing updates to its missing-update-applications file. Also, appropriate entries in the missing-update-value file at *A* are created for any sites that have still not applied all or some of the received updates for the entity. Site *A* can then inform *B* that the appropriate entries may be removed from its missing-update-value file. This processing is correct in that updates are never applied out of order, because of the completeness property shown in Figure 3.

The Completeness Property:

A missing-update-value file contains a missing update of some entity copy only if all earlier unapplied missing updates for the entity copy are also present.

This property is due to the following four facts:

(1) Missing update information must be posted at sites storing a quorum of copies of a write set entity prior to acceptance by these sites of any later precommits for the entity.

(2) Precommit acknowledgments contain all applicable missing update information.

(3) A quorum of the copies of a write set entity must be precommitted before the entity can be updated.

(4) Any two quorums of the copies of an entity must intersect.

Fig. 3. The completeness property of missing-update-value files.

Missing updates may be retrieved and applied in two situations. First of all, an explicit request may be made for a particular missing update. Such a request would be made by a site that has received missing update information that pertains to an entity copy that it stores, for which it has no record in its missing-update-applications file. This request would be sent to one or more of the other sites that store the entity and/or to the home site of the transaction that created the missing update. Note that if the home site has no knowledge of the missing update, the missing update information was generated by a transaction that was restarted or aborted and is invalid. Although this situation is the only one in which it is really necessary to apply missing updates, data are kept more current by attempting to apply missing updates at other times as well. For this purpose each site that has a nonempty missing-update-value file periodically attempts communication with the relevant sites. This periodic communication also ensures that these files are eventually emptied. Entries must also be periodically removed from missing-update-postings and missing-update-applications files. Procedures to accomplish this are based on network traversals but are straightforward and can be made robust; examples are given in [6].

4.4 Dangling Precommit Processing

The transaction-status file at each site allows the site to answer requests for transaction status information. This file records transaction restarts, aborts, and commits of which the site has knowledge. For example, if a site received a release-precommit message, it would create an entry in its transaction-status file to record the fact that the corresponding transaction had been restarted or aborted.

A transaction commit entry also contains any missing update information intended for sites that acknowledge precommits, but that may not have received this information (e.g., if the information was sent with a commit message, there

Context:

A site *A* receives from a site *B* a query on the status of a transaction *T* that has left a dangling precommit at *B*. The required processing depends on the status of *T* at *A*. Site *A* follows the first applicable case below:

Case I - There is a commit entry for *T* in the transaction-status file at *A*:

Site *A* informs *B* that *T* is committed. In addition, all of the missing update information stored with (or referenced by) the file entry is transferred to *B*. If this information comes from the missing-update-value file (indicating that the precommit acknowledgment was not received by H(*T*)), *B* must process the dangling precommit as if it were a missing update (which it essentially is). Otherwise, *B* processes any information intended for itself and stores the remaining information in its transaction-status file when it commits its precommit. This ensures that the precommit is committed only after the missing update information of *T* is posted at *B*.

Case II - An entry for *T* indicates that *T* was aborted or restarted:

B is notified of this fact.

Case III - Each write set entity copy stored at *A* (of which there is at least one) has been precommitted at *A*:

B is notified of this fact.

Case IV - A precommit for a write set entity copy stored at *A* has not been received:

Site *A* makes a pledge to *B* that it will reject any precommits for *T* that it receives in the future.

Fig. 4. Status query processing.

may be no guarantee that this message was received). This information is contained in the messages that create these entries (such as commit messages and responses to requests for missing updates). Since a precommit could be received but not acknowledged, transaction commit entries also reference any relevant entries in the missing-update-value file.

When a site has a dangling precommit, it queries the other sites that may be aware of the status of the owning transaction. For this purpose, a list of the identifiers of these sites is present (either explicitly or implicitly) in each precommit message. This list would certainly include the identifier of each site storing a copy of a write set entity. The processing necessary when a site receives a status query is shown in Figure 4.

On the basis of the received replies to its status queries, a site may or may not be able to determine whether its dangling precommit can be committed. If the site cannot determine whether the precommit should be committed, it periodically attempts additional status queries. It should be clear that this processing implements the previously discussed strategies for dealing with dangling precommits.

4.5 Correctness of the Basis

In this section it is shown that a concurrency control method that maintains consistency under the requirement that all write set entity copies be precommitted (in other words, a requirement prohibiting missing updates) will continue to maintain consistency after implementing the presented basis for maximal partial operability. The following proposition is required for this result.

PROPOSITION 4.1. *Suppose that there exists a set of committed transactions* $\{T_i\}$, $1 \leq i \leq n$, *such that* $T_i \rightarrow T_{i+1} (1 \leq i \leq n - 1)$. *If* T_1 *was aware of a missing update at the time of its commitment, then either* T_n *was also aware of this missing update when* T_n *was committed, or the update had been applied by this time.*

ARGUMENT. The argument proceeds by induction on n. For $n = 1$, the proposition is trivially true. Assume that the proposition holds for n, and consider the proposition for $n + 1$.

By the inductive hypothesis, either T_n was aware of the missing update when T_n was committed, or the update had been applied by this time. If $T_n \rightarrow_{RW} T_{n+1}$ or $T_n \rightarrow_{WW} T_{n+1}$, the proposition holds owing to the rule requiring T_n to post its missing update information at sites storing a quorum of the copies of each read and write set entity before any later conflicting precommits are accepted at these sites. If $T_n \rightarrow_{WR} T_{n+1}$, the proposition holds, since the commitment, dangling precommit processing, and missing update application procedures ensure that an update at a site is committed only after the missing update information of the corresponding transaction is posted at that site. By induction the proposition is now established. □

From this proposition the main result can be derived:

ASSERTION 4.2. *Any concurrency control method that maintains consistency when no missing updates are allowed will continue to maintain consistency after implementing the presented basis for maximal partial operability.*

ARGUMENT. Suppose, on the contrary, that there exists a set of committed transactions $\{T_i\}$, $1 \leq i \leq n$, such that $T_i \rightarrow T_{i+1} (1 \leq i \leq n - 1)$ and $T_n \rightarrow T_1$.

Since the concurrency control method maintains consistency when no missing updates are allowed, one of these conflicts must have arisen owing to a missing update. Without loss of generality, assume that T_1 read an entity copy that missed an update of T_2.

Since the basis prohibits the application of a missing update while any transaction that read the old value is active (if this application could cause a cycle in the $\rightarrow$ relation), the missing update must have been unapplied up to the time T_1 was committed. By Proposition 4.1, this implies that T_1 was aware of the missing update when it was committed. As this is a contradiction to the rules of Figure 4, the assertion is shown. □

5. AN APPLICATION TO A DISTRIBUTED LOCKING METHOD

This section illustrates the application of the presented basis for maximal partial operability to a distributed locking method. In Section 6, application to a concurrency control method that utilizes timestamps is considered. It should be realized that the basis can be applied to a wide variety of other methods, as is further illustrated elsewhere [6].

5.1 The Basic Method

Distributed two-phase locking (*distributed 2PL*) is based on the *wound-wait* concurrency control method of Rosenkrantz, Stearns, and Lewis [22], and on the distributed 2PL method as described by Bernstein and Goodman [3]. More complex enhancements of this method are possible [6, 25, 27]; our particular choice is based on the suitability of the method as an example.

In this method, a transaction wishing to perform a read operation on an entity must obtain a read lock on a copy of the entity granting permission. A transaction wishing to perform a write operation must obtain write locks on all of the entity's copies. Write locks are said to *conflict* with read and write locks in the sense that two conflicting locks of different transactions are not allowed on the same entity copy.

A transaction is *well formed* if it obeys these locking rules and if it eventually releases all of its locks. It is *two phase* if no lock requests occur after the first lock release [7]. In our transaction processing model, write lock requests may be carried implicitly by precommit operations. Similarly, read lock requests may be carried implicitly by database read operations. Commit operations serve to release all of a transaction's locks at the receiving sites. For those sites at which a transaction read, but did not write, explicit *read-lock-release* messages are required. As shown by Eswaran et al. [7], a concurrency control method that requires all committed transactions to be well formed and two phase maintains consistency.

To complete our description of distributed 2PL, the problem of deadlock must be considered. To prevent deadlock, each transaction has an associated *priority*. As the age of a transaction increases, its priority increases relative to that of other currently executing transactions. Distributed 2PL prevents deadlock by ensuring that transactions essentially only wait for transactions of higher priority. When a lock request of a transaction conflicts with existing locks held on the entity copy, the transaction waits only if its priority is lower than that of one of the lock holders. Otherwise, a *wound* message is sent to the home site of each of the lock holding transactions. If a transaction that is not already committed is "wounded," it is restarted, and messages are sent to release all of the transaction's locks. If the transaction is already committed, it does not require any additional locks, and its locks are, in fact, in the process of being released. A proof that this priority system guarantees that each transaction is eventually either committed or aborted (possibly after a number of restarts) is given by Rosenkrantz, Stearns, and Lewis [22].

5.2 The Robustness Level of the Basic Method

Before attempting any modification of distributed 2PL, the robustness level of the basic method should be considered. This robustness level is illustrated by the effect of a site failure in a system using basic distributed 2PL. Such a failure makes unavailable for updating all of the entities with copies at the site, as well as all of the entities for which copies were precommitted or read-locked by transactions executing at the site at the time of the site failure. Any entities

Execution Sequence (with a missing update)

T_1 and T_2 both read x_A and y_B
A partitioning separates $H(T_1)$ from A
T_1 writes x, missing copy at A
T_1 commits, releases locks
T_2 writes y, commits

(a)

Execution Sequence (with a release of a dangling read lock)

T_1 and T_2 both read x_A and y_B
A partitioning separates $H(T_2)$ from A
The read lock of T_2 on x_A is released by A due to the partitioning
T_1 writes x
T_1 commits, releases locks
T_2 writes y, commits

(b)

In both (a) and (b)
$T_1 \rightarrow T_2$ due to a conflict on y
$T_2 \rightarrow T_1$ due to a conflict on x

Fig. 5. The limitations of basic distributed 2PL.

whose only available copies were either stored at the site or were precommitted by transactions executing at the site are made unavailable for reading.

Basic distributed 2PL does not attain maximal partial operability for three reasons: (1) missing updates cannot be allowed (see Figure 5a); (2) there are no procedures to resolve dangling precommits; and (3) *dangling read locks* (read locks that cannot be released by the owning transactions owing to network partitioning) cannot be released by the affected sites (see Figure 5b).

5.3 Robustness Modifications

Before describing the details of our robustness modifications, a few general remarks will be made here about the application of the concepts of the previous section to distributed 2PL. Perhaps the most significant point is that most of the material presented (e.g., dangling precommit processing), is truly independent of the concurrency control method involved. For this reason the majority of our robustness modifications entail a straightforward application of the basis for maximal partial operability and are not described further. Our modifications to transaction commitment, on the other hand, depend in part on when conflicting operations can be performed and therefore need further specification once a context has been fixed. In addition, there were two procedures left unspecified in the basis: (1) a procedure to check for later precommits or updates upon reception of a missing-update-data message (see Section 4.1), and (2) a procedure required in missing update application to check for the presence of an active transaction that read an entity copy (see Section 4.3). Finally, procedures to allow the release of dangling read locks also need to be discussed, as these must be present if distributed 2PL is to attain maximal partial operability. These procedures are not considered in the basis, since read operations do not impose access restrictions in a significant number of proposed concurrency control methods.

Consider the processing required when a transaction T wishes to commit. If T is unaware of any missing updates, it is committed exactly as before. The robustness modifications to distributed 2PL do not significantly degrade method performance in the absence of failures. Consider the case in which T is aware of missing updates. In robust distributed 2PL, dangling read locks may be released, but write locks cannot be released before transaction status is known. Only precommits that conflict with a read operation (as opposed to a precommit) of T can be performed before T commits. Due to this fact, H(T) must transmit a missing-update-data message to a site A only if A stores a quorum copy of an entity that was read but not written. The additional required posting can occur when the updates of T are committed.

Each site A that receives a missing-update-data message must ensure that each read set entity copy has not been precommitted or updated by some other transaction that T precedes. In robust distributed 2PL, A returns sufficient information in its missing-update-data message acknowledgment to enable H(T) to make this check itself. This information consists of any entries for the read set entity copies that are found in A's missing-update-postings file, as well as the identities of any precommits that exist on these entity copies. This information is utilized as shown in Figure 6. In this figure it is assumed that acknowledgments have been received from a sufficient set of sites to form the necessary quorums.

The last basis feature that will be considered is the missing update application procedure. In this procedure, a missing update can be applied only after it is ensured either that no uncommitted active transaction has read the old entity copy value, or that applying the update when such a transaction exists cannot cause a cycle in the $\rightarrow$ relation. In robust distributed 2PL it is first necessary to check whether the entity copy is read locked. If it is, application of the update is postponed until the read lock is released by the transaction or by the site. If the entity copy is not read locked, a check must be made for any "recent" (since the last update of the entity copy) releases of dangling read locks held on the entity copy. If such releases exist, the update cannot be applied until the site learns that the owning transactions have been committed (or restarted or aborted). If such releases do not exist, either no uncommitted active transactions have read the entity copy, or applying the update when such a transaction exists cannot cause a cycle in the $\rightarrow$ relation. This last method characteristic is due to the procedures for releasing dangling read locks.

Dangling read locks have different properties than do dangling precommits. In the case of a dangling read lock it is not necessary to know whether the owning transaction is committed or whether it had to be restarted or aborted. It is only necessary to know whether a cycle in the $\rightarrow$ relation would arise by releasing the lock.

Consider the situation in which a site A decides to time out a read lock of a transaction T due to suspected failures. Presumably A has unsuccessfully tried to communicate with H(T) to determine the status of T. The read lock at A is

Precommit Checking (for a transaction *T*):

If a precommit exists on a read set entity copy of *T*, three cases can be distinguished:

(1) The precommit corresponds to missing update information that *T* received through missing-update-data message acknowledgments.

(2) The precommit is a dangling precommit left by the transaction that created the entity version read by *T*.

(3) The precommit belongs to a transaction that *T* precedes.

In the first case, the application procedures of section 4.3 may be followed. The processing of *T* is not impacted. H(*T*) may query the necessary sites to distinguish between the last two cases, or may assume the worst case, namely that the precommit belongs to a transaction that *T* precedes.

Update checking:

One of the read set entity copies of *T* has been updated to a later version if and only if *T* received a missing-update-data message acknowledgment that contained one of the following:

(1) Missing update information for an entity copy read by *T*.

(2) Information indicating that one of *T*'s read locks has been released (due to the dangling read lock release procedure).

Fig. 6. Checking for a later update or precommit in robust distributed 2PL.

removed and an entry is placed in the missing-update-postings file at *A*. This *transaction entry*, to be transferred to transactions that precommit the entity copy that was read locked, contains the identifier of *T* instead of a missing update identification. A transaction entry is passed from transaction to transaction as is missing update information. If a transaction becomes aware of an entry for itself, it must be restarted. Dangling read lock processing is illustrated in Example 3 in the Appendix.

5.4 The Correctness of Robust Distributed 2PL

Robust distributed 2PL attains maximal partial operability, since (1) dangling precommits are resolved to the extent possible under maximal partial operability, (2) transactions may be executed and committed as long as the entity copy availability requirements under maximal partial operability are satisfied, and (3) no other restrictions on transaction processing are induced by site or communication link failures (e.g., dangling read locks may be released). As it is clear that each transaction is eventually either committed or aborted, as in basic distributed 2PL, only the following assertion is required to demonstrate correctness:

ASSERTION 5.1. *Robust distributed 2PL maintains consistency.*

ARGUMENT. Suppose, on the contrary, that the use of robust distributed 2PL resulted in a set of committed transactions $\{T_i\}$, $1 \leq i \leq n$, such that $T_i \rightarrow T_{i+1}$ $(1 \leq i \leq n-1)$ and $T_n \rightarrow T_1$.

By Assertion 4.2, the basis implementation cannot be responsible for the inconsistent execution history. Since the only other modification is the dangling read lock release procedure, this procedure must be responsible. Assume, without loss of generality, that the $T_1 \rightarrow T_2$ conflict arose owing to the updating by T_2 of an entity copy after the release of a dangling read lock held by T_1.

If this updating had been performed by the missing update application procedure, T_2 would have been aware of a missing update for an entity copy read by T_1. Also, note that the read lock release would have been "recent" when the missing update was applied. By the properties of the missing update application procedure, T_1 must have been committed before this updating was performed. By Proposition 4.1, T_1 would have been aware of the missing update. This is a contradiction.

Now consider the alternative possiblity, in which T_2 obtained a precommit acknowledgment for the entity copy. Our dangling read lock release procedures then ensure that T_2 was made aware of a transaction entry for T_1. Since transaction entry information is passed from transaction to transaction, in much the same way as missing update information is passed, a result analogous to Proposition 4.1 holds. This result implies that T_1 was aware of the transaction entry for itself when it was committed. As this is a contradiction, the assertion is shown. □

6. AN APPLICATION TO A TIMESTAMP ORDERING METHOD

6.1 The Basic Method

Basic Timestamp Ordering (*basic T/O*) is based on a similarly named method of Bernstein and Goodman [3]. In this method each home site assigns each of its transactions a unique *timestamp*. Timestamps need have no relation to physical time, although it is assumed that they are generated at each site in increasing order. The timestamp of a transaction T is denoted by $\text{TS}(T)$.

Basic T/O also requires a *read* and a *write* timestamp to be present on each database entity copy. After a "sufficiently long" period without access, an entity copy's explicit timestamp may be discarded and replaced by an implicit timestamp with value greater than or equal to its former explicit value. By comparing transaction timestamps and entity copy timestamps before performing operations, the concurrency control method can verify that conflicting operations are being performed in timestamp order. If such a comparison reveals that an operation cannot be processed so as to preserve timestamp order, the operation is rejected. The corresponding transaction must then be assigned a new timestamp and restarted. Basic T/O rarely delays operations, unlike many other timestamp-based concurrency control methods, but instead relies on transaction restarts.

The processing of read, precommit, and commit operations are now specified in greater detail. First of all, consider a precommit of a transaction T for an entity copy x_A. If the read timestamp of x_A is greater than $\text{TS}(T)$, the precommit is rejected, since conflicting operations should not be processed in nontimestamp order. Otherwise, the precommit is accepted. Note that in basic T/O, unlike the situation in distributed 2PL, an entity copy may have several outstanding precommits.

When a read operation of T for an entity copy x_A is received, the write timestamp of x_A is compared to TS(T). If the former is greater, the read is rejected. Otherwise, the timestamps of any outstanding precommits on x_A are checked. If there is an outstanding precommit with timestamp less than TS(T), the read request is queued. This action is necessary, since the precommitted update must be applied in timestamp order with respect to any conflicting read operations. If no such precommit exists, the read is processed and the read timestamp of x_A is updated to the maximum of its current value and TS(T).

Finally, consider the reception of a commit of T at a site A. Assume that a precommit of T exists on an entity copy x_A. If the write timestamp of x_A is greater than TS(T), the value of x_A is not updated. Note that this action processes the update so as to preserve a timestamp ordering of conflicting operations. If the write timestamp of x_A is less than TS(T), any precommits with timestamp less than TS(T) are essentially discarded. This processing is correct owing to the fact that the corresponding updates will never be applied. The value of x_A is then updated, and the write timestamp of x_A updated to TS(T). Finally, any read requests queued for x_A are reevaluated, using the same rules as those applying to arriving reads.

Basic T/O always processes conflicting operations so as to preserve timestamp order. Therefore, the result of processing a set of transactions is the same as if they were processed sequentially in timestamp order. Basic T/O maintains consistency. To guarantee that each transaction is eventually either committed or aborted, a randomized increment may be applied to the timestamp of a restarted transaction.

6.2 The Robustness Level of the Basic Method

The robustness level of basic T/O is illustrated by the effect of a site failure in a system using basic T/O. Consider only those transactions that have a larger timestamp than that of any transaction that was executing at the site at the time of the failure. The entities made unavailable for updating are then all of the entities with copies at the site, as well as all of the entities that are made unavailable for reading. The entities made unavilable for reading are those entities whose only available copies were either stored at the site or were precommitted by transactions executing at the site.

Note that the dangling precommit problem is partly "solved" in basic T/O, in that many dangling precommits can essentially be discarded (in the sense that their imposed access restrictions become redundant). This is illustrated in Figure 7a. However, basic T/O cannot resolve all theoretically resolvable dangling precommits, since there is no mechanism to link the absence of a precommit on one entity copy to a resolution of a precommit on a copy of a different entity.

Basic T/O does not achieve maximal partial operability. This is due to the fact that missing updates cannot be allowed (see Figure 7b), and due to the limitations on dangling precommit resolution.

Execution Sequence
(discarding a dangling precommit)

$\{\mathrm{TS}(T_n) = n\}$

T_1 leaves a dangling precommit on x_A,
but successfully updates x_B
T_2 reads x_B, obtains precommits on
all copies of x
When T_2 commits its update of x_A, the
precommit of T_1 will be effectively
discarded

(a)

Execution Sequence
(with a missing update)

$\{\mathrm{TS}(T_n) = n\}$

T_1 and T_2 both read x_A and y_B
A partitioning separates $\mathrm{H}(T_1)$ from A
T_1 writes x, missing copy at A
T_1 commits
T_2 writes y, commits

$T_1 \rightarrow T_2$ due to a conflict on y
$T_2 \rightarrow T_1$ due to a conflict on x

(b)

Fig. 7. The limitations of basic T/O.

6.3 Robustness Modifications

Only those basis features that need further specification in the context of basic T/O are discussed. First of all, note that commitment is performed exactly as before for those transactions that are not aware of any missing updates. In a failure-free environment the robustness modifications to basic T/O induce no significant performance overhead. Consider the commitment of a transaction T that is aware of missing updates. Unfortunately, owing to the absence of locking in basic T/O, T must post its missing update information prior to commitment at sites storing a quorum of the copies of each of its read and write set entities.

Each site A that receives a missing-update-data message must ensure that each relevant entity copy has not been precommitted or updated by a transaction that T precedes. For a write set entity this is quite simple. It is only necessary to compare the timestamp of T to the write timestamp of the entity copy and the timestamps of any other outstanding precommits.

Consider the detection of a conflicting operation on a read set entity that was not updated. To facilitate this detection, each missing-update-data message includes the write timestamps of the read set entity copies read by T. Due to the fact that updates are always processed to preserve timestamp order, it is only necessary to compare the timestamps in the missing-update-data message with the entity copy write timestamps and the timestamps of any outstanding precommits. This action is sufficient to detect a precommit or update of a transaction that T precedes.

The last basis feature that is discussed is the missing update application procedure. In this procedure, a missing update can be applied only after it is ensured either that no uncommitted active transaction has read the old entity copy value or that applying the update when such a transaction exists cannot cause a cycle in the $\rightarrow$ relation. Unfortunately, in basic T/O there is no simple way to recognize the existence of such a transaction.

One possible solution will be described here. Each commit message from a site A includes the minimum timestamp value of any uncommitted transactions executing at A (or the current "time" if there are no such transactions). This information is added to the receiving site's transaction-status file. In conjunction with entity copy read timestamps (or a log of database operations), the extended transaction-status file enables each site to recognize the possible existence of active transactions that read some entity at the site. When missing update application is blocked by such a possibility, the site should query the relevant sites and postpone applying the missing update until learning that any applicable transactions have been committed (or restarted or aborted).

An additional aspect of the missing update application procedure should also be noted. Recall that in our basis a site checks its missing-update-applications file to determine if a missing update has already been applied. Since updates are always processed so as to preserve timestamp order, and since each entity copy has a write timestamp, this file is not required in robust basic T/O. It is only necessary to check the write timestamp of the entity copy to be updated.

6.4 The Correctness of Robust Basic T/O

Since no modifications other than the basis implementation were made to basic T/O, robust basic T/O maintains consistency by Assertion 4.2. Also, the robustness modifications do not impair the ability of basic T/O to ensure that each transaction is eventually either committed or aborted. However, it should be noted that a missing update is not, in general, processed in timestamp order with respect to the read operations performed on the entity copy. The result of processing a set of transactions may be the same as a sequential ordering different from timestamp order.

7. CONCLUSIONS

It has been demonstrated that a high level of robustness against site and communication link failures can be attained by a concurrency control method without significantly impacting performance when failures are infrequent. When failures do occur, each site must accumulate information in several files in order to safely proceed in the presence of missing updates. In an active distributed database system partitioned by failures, these files could grow quite rapidly. However, this must be accepted as the price of a high robustness level.

Our results also have implications for concurrency control method performance analysis. Performance evaluations for environments in which failures are relatively rare need only consider basic methods, even if the methods would actually be enhanced when implemented to provide a high robustness level. The separation of robustness considerations from performance evaluation adds further credibility to the previous and ongoing performance studies (e.g., [8, 9, 11, 18, 21] that have for the most part considered concurrency control methods that lack robustness. However, it should be noted that the relative performance of a set of concurrency control methods that attain maximal partial operability may differ according to the frequency of failures. Performance evaluations for environments in which failures are relatively frequent should therefore consider concurrency control methods that have been enhanced to provide a high robustness level.

Distributed database concurrency control methods that attain a high robustness level are likely to be of significant practical benefit. Although one is usually safe in assuming that failures will be rare in a centralized database system, in a large distributed system an inoperative site would likely be commonplace. In many applications it is essential that processing continue in the presence of such failures.

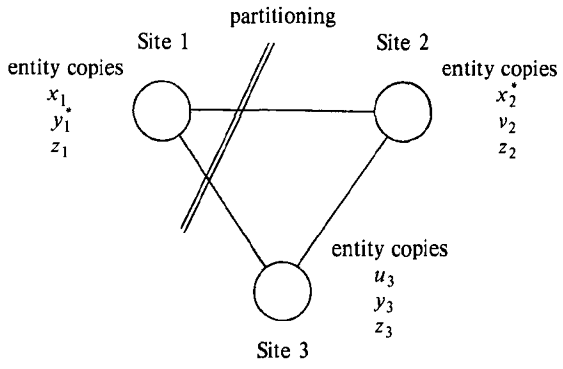
APPENDIX

Three examples that illustrate the operation of concurrency control methods modified to achieve maximal partial operability are presented. In these examples a nonrobust method (such as nonrobust distributed 2PL or basic T/O) could not successfully execute any of the transactions treated. The distributed database shown in Figure 8 is a common context.

Example 1. *Posting Missing Update Information.* After the partitioning occurs, a transaction T_1 is initiated at site 3. T_1 wishes to read entities u and z and update entity z. In a method achieving maximal partial operability, it may do so, even though the update for entity copy z_1 cannot be initially performed. Missing update information for entity copy z_1 is posted at site 3 (to be transferred on a read or precommit of z_3 or a precommit of u_3) and at site 2 (to be transferred on a read or precommit of z_2). Entries in the missing-update-value files at sites 2 and 3 are created for z_1.

Example 2. *Dangling Precommit Processing.* Suppose that before the partitioning occurred, a transaction initiated at site 1 precommitted all the copies of entities x and y. However, the commit for site 2 was received before the partitioning, while that for site 3 was not. After the partitioning, a transaction T_2 initiated at site 3 wishes to read y_3. However, suppose that it is blocked by the dangling precommit (this will always occur in distributed 2PL; in basic T/O it would occur if the timestamp of T_2 is larger than the timestamp of the transaction that produced the dangling precommit). Precommits are required to specify (either explicitly or implicitly) the sites that will learn the final transaction status. Therefore, site 3 has sufficient information to recognize that site 2 should be queried. When site 2 informs site 3 that the transaction in question has been committed, site 3 can safely commit its precommitted update, allowing T_2 to proceed.

Example 3. *Dangling Read Lock Processing.* (This example is relevant only to distributed 2PL.) Suppose that when the partitioning occurred, a read lock of a transaction initiated at site 1 was still outstanding on entity copy v_2. A transaction T_3 initiated at site 2 wishes to read and update v_2, but is blocked by the dangling



(in the case of two entity copies, a '*' denotes that copy necessary and sufficient for a quorum; otherwise, a quorum is a simple majority)

Fig. 8. A distributed database partitioned by a failure.

read lock. After site 2 has timed out the read lock, it is discarded, and a transaction entry is created in the missing-update-postings file at site 2 (to be transferred on a precommit of v_2). T_3 can now proceed successfully. The transaction entry is passed to T_3, which posts it at site 3 again (to be transferred on a read or precommit of v_2).

ACKNOWLEDGMENTS

We wish to thank P. Bernstein, B. Galler, and B. Lindsay along with several anonymous referees for their constructive comments on this work and its presentation.

REFERENCES

1. Attar, R., Bernstein, P. A., and Goodman, N. Site initialization, recovery, and back-up in a distributed database system. Tech. Rep. TR-13-81, Aiken Computation Lab., Harvard Univ., Cambridge, Mass., Aug. 1981.
2. Badal, D. Z., and Popek, G. J. A proposal for distributed concurrency control for partially redundant distributed database systems. In *Proc. 3rd. Berkeley Conf. Distributed Data Management and Computer Networks* (Berkeley, Calif., Aug. 1978) Lawrence Berkeley Lab., Univ. of California, pp. 273-285.
3. Bernstein, P. A., and Goodman, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185-221.
4. Bernstein, P. A., Shipman, D. W., and Rothnie, J. Concurrency control in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (Mar. 1980), 18-51.
5. Bernstein, P. A., Shipman, D. W., and Wong W. S. Formal aspects of serializability in database concurrency control. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 203-216.
6. Eager, D. L. Robust concurrency control in distributed databases. M.Sc. thesis, Tech. Rep. CSRG 135, Computer Systems Research Group, Univ. of Toronto, Toronto, Ont., Oct. 1981.
7. Eswaran, K. R., Gray, J. N., Lorie, R. A., and Traiger, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624-633.
8. Galler, B. I. Concurrency control performance issues. Ph.D. dissertation, Univ. of Toronto, Toronto, Ont., Sept. 1982.
9. Garcia-Molina, H. Performance comparison of two update algorithms for distributed databases. In *Proc. 3rd. Berkeley Conf. Distributed Data Management and Computer Networks*, (Berkeley, Calif., Aug. 1978) Lawrence Berkeley Lab., Univ. of California, pp. 108-119.

10. Garcia-Molina, H. A concurrency control mechanism for distributed databases which uses centralized locking controllers. In *Proc. 4th Berkeley Conf. Distributed Data Management and Computer Networks,* (Berkeley, Calif., Aug. 1979) Lawrence Berkeley Lab., Univ. of California, pp. 113-125.
11. Gelenbe, E., and Sevcik, K. Analysis of update synchronization for multiple copy databases. *IEEE Trans. Comput. C-28,* 10 (Oct. 1979), 737-747.
12. Gifford, D. K. Weighted voting for replicated data. In *Proc. 7th ACM Symp. Operating Systems Principles* (Pacific Grove, Calif., Dec. 10-12, 1979) ACM, New York, pp. 150-159.
13. Gray, J. Notes on database operating systems. Tech. Rep. RJ2188, IBM, New York, Feb. 1978.
14. Hammer, M., and Shipman, D. W. Reliability mechanisms for SDD-1: a system for distributed databases. *ACM Trans. Database Syst. 5,* 4 (Dec. 1980), 431-466.
15. Hsiao, D. K., and Ozsu, T. M. A survey of concurrency control mechanisms for centralized and distributed databases. Tech. Rep. CISRC-81-1, Ohio State Univ., Columbus, Ohio, Feb. 1981.
16. Kung, H. T., and Papadimitriou, C. H. An optimality theory of concurrency control for databases. In *Proc. 1979 ACM-SIGMOD Int. Conf. Management of Data* (Boston, Mass., May 30-June 1, 1979) ACM, New York, pp. 116-126.
17. Lampson, B. W., and Sturgis, H. E. Crash recovery in a distributed data storage system. Xerox PARC Report, Xerox Palo Alto Research Center, Palo Alto, Calif., April 1979.
18. Lin, W. K. Performance evaluation of two concurrency control mechanisms in a distributed database system. In *Proc. 1981 ACM-SIGMOD Int. Conf. Management of Data* (Ann Arbor, Mich., April 29-May 1, 1981) ACM, New York, pp. 84-92.
19. Menasce, D. A., Popek, G. J., and Muntz, R. R. A locking protocol for resource coordination in distributed databases. *ACM Trans. Database Syst. 5,* 2 (June 1980), 103-138.
20. Montgomery, W. A. Robust concurrency control for a distributed information system. Ph.D. dissertation, Tech. Rep. MIT/LCS/TR-207, MIT, Cambridge, Mass., Dec. 1978.
21. Ries, D. The effect of concurrency control on the performance of a distributed data management system. In *Proc. 4th Berkeley Conf. Distributed Data Management and Computer Networks,* (Berkeley, Calif., Aug. 1979) Lawrence Berkeley Lab., Univ. of California, pp. 221-234.
22. Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3,* 2 (June 1978), 178-198.
23. Skeen, D. Nonblocking commit protocols. In *Proc. 1981 ACM-SIGMOD Int. Conf. Management of Data* (Ann Arbor, Mich., April 29-May 1, 1981) ACM, New York, pp. 133-142.
24. Skeen, D. A quorum-based commit protocol. Tech. Rep. 82-483, Cornell Univ., Ithaca, N. Y., Feb. 1982.
25. Stearns, R. E., and Rosenkrantz, D. J. Distributed database concurrency controls using before-values. In *Proc. 1981 ACM-SIGMOD Int. Conf. Management of Data* (Ann Arbor, Mich., April 29-May 1, 1981) ACM, New York, pp. 74-83.
26. Stonebraker, M. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng. SE-5,* 3 (May 1979), 188-194.
27. Stucki, M. J., et al. Coordinating concurrent access in a distributed database architecture. In *Proc. 4th ACM Workshop Computer Architecture for Non-Numeric Processing* (Blue Lake, N. Y., Aug. 1-4, 1978) ACM, New York, pp. 60-64.
28. Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4,* 2 (June 1979), 180-209.

Received December 1981; revised May 1982; accepted December 1982