# UPDATING REPLICATED DATA DURING COMMUNICATIONS FAILURES

Barbara T. Blaustein and Charles W. Kaufman

Computer Corporation of America, Alexandria, VA 22314
Computer Corporation of America, Cambridge, MA 02142

## Abstract

In applications such as banking, reservation systems, inventory, and command and control it is desirable to allow updates to replicated data even during communications delays or failures. We present a technique that allows each site to process updates regardless of the state of the network -- to continue to update its own copy of the data and to process information about updates at other sites whenever it is available. Each site acts independently to revise its copy of the replicated data when it receives information about relevant updates. The process of merging newly received updates with those already processed is made more efficient by exploiting simple semantic properties of the updates.

This research was supported by the Defense Advanced Research Project Agency of the Department of Defense and by the Air Force Systems Command at Rome Air Development Center under Contract No. F30602-84-C-0112. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

## 1. Introduction

Replicated databases, databases in which copies of critical data are stored at different sites, are useful for many applications (e.g., banking, reservation systems, command and control) that depend on the ability to retrieve data at all times. Replication offers two main advantages: efficiency -- many operations can be handled locally thus reducing communications costs and delays (this efficiency gain must be balanced against the cost of replicating all updates), and reliability -- if one site is down, or has lost some of its data, the data is likely to be available at another site.

When data is updated at one site, that site must maintain mutual consistency by broadcasting the updates to all the other sites. Communications failures, therefore, have a drastic effect on replicated databases. Although it is possible to handle communications failures by blocking updates at some sites until communications are restored [Alsberg76, Davidson84, Gifford79, Skeen84, Stonebraker78, Thomas79], such a solution negates some of the reasons for using replicated databases in the first place. This paper describes a technique for allowing all updates to continue during communications delays and failures, with each site independently restoring consistency of its own copy of the data when communications are restored.

Each site determines the actions it must take to merge newly received updates into its local database. These actions may include undoing some updates already reflected in the local database, re-running some of these updates, and running new updates. The primary objective of the work presented here is to make this merge process more efficient by exploiting simple semantic properties of transactions (such as commutativity). Sites use this information to reduce, without compromising correctness, the number of actions necessary for the merge. (More details and proofs of the theorems in this paper appear in [Blaustein85].)

For simplicity, this paper discusses the problem in terms of a two-site network, full data replication (a complete copy of all data at each site), and "clean" communications failures (sites stop all processing when they fail, partitions are detected, etc.); but the results extend to the general problem of restoring replicated data at any number of sites after any type of communication delay. In a network in which communication delays and failures may be frequent or difficult to detect, each site is always prepared to receive older updates from other sites that may have been partitioned [Blaustein83, Sarin85]. When a site receives information about updates at any other sites, it uses this information to restore any affected data. The new updates -- whenever and wherever they may have originated -- are treated as updates performed at a single other site during a partition. This process is repeated at each site whenever information about other sites' updates is received.

In a system that allows updates at all sites during a network partition, it is rather complicated to define the notions of consistency and correctness for the restored database state. Of course, mutual consistency of the copies of data is essential but not sufficient. The traditional approach of defining a correct database state as a state resulting from a serial execution of transactions is not possible if arbitrary transactions are allowed on opposite sides of a partition. We will be looking for a weaker, but nonetheless useful, definition. A discussion of our correctness criteria follows some fundamental definitions.

### 1.1 Basic Definitions

Transactions read and write sets of database values; they are an atomic collection of database operations (our definition is much more general than that in [Fischer82]). Because we assume full data replication, each transaction can be executed at any site. For simplicity, we assume all transactions are instances of uniquely named pre-defined transaction types and may be passed arguments when they are invoked; this assumption allows pre-determination of the semantic properties of the transactions, thus improving the efficiency of the system, but it is not strictly necessary.

A log is a sequence (i.e., a set ordered by local timestamp) of transactions that are executed at a single site. A log consisting of transactions T1,...,Tn is written [T1,...,Tn]. The log of transactions executed at a particular site during a network partition is called the partition log for that site.

Transactions executed at a particular site during a network partition are called local transactions for that site. Other transactions are non-local for that site.

Logs map one database state to another. Thus, given a database state D and a log [T1,...,Tm], we denote the state produced by executing the log in D as D([T1,...,Tm]). For brevity, we may leave out the parentheses and simply write D[T1,...,Tm].

Two logs L1 and L2 are equivalent, L1 = L2, if and only if (iff) for all database states D, D(L1) = D(L2).

In this paper we assume that only pre-defined update transactions are used, that the database schema does not change during a partition, and that each site has its own (relatively reliable or well-maintained) hardware clock.

The problem caused by a network partition is illustrated in Figure 1. Before the partition, sites X and Y have mutually consistent copies of data. The database state D that exists at both

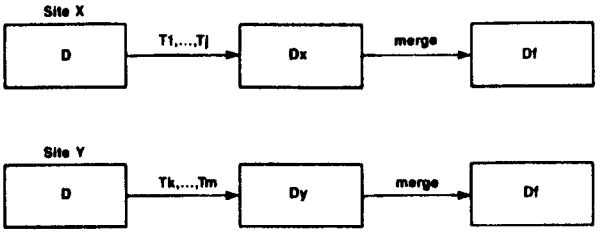



Figure 1. The Problem

sites is called the initial database state. During the partition, site X executes the log [T1,...,Tj] and site Y executes the log [Tk,...,Tm]. Therefore when the partition ends, the sites are left with the divergent partitioned database states Dx = D[T1,...,Tj] and Dy = D[Tk,...,Tm]. The goal is to devise an automatic procedure for restoring the consistency of the copies. The procedure described in this paper first uses the partition logs to define the final merge state Df -- the state that must exist at all sites after the merge -- and then finds merge logs Lx and Ly such that Dx(Lx) = Dy(Ly) = Df.

The final merge state is defined in terms of the initial state and a target log that includes all transactions executed during the partition. If D is the initial state and T is the target log, then the final merge state Df is D(T).

The key to the definition of the target log, and hence the final merge state, is a system-wide algorithm for ordering transactions. We do not rely on an algorithm for synchronizing clocks at the different sites (see, for example [Lamport78]); we do assume that there is some way for

all sites to agree on which transactions were executed "before" which other transactions during the network partition. We therefore assume that there are locally maintained clocks and some system-wide "tie-breaking" algorithm for agreeing on some ordering of transactions executed at different sites during the partition. (The algorithm must preserve each site's local timestamp order.) The system-wide algorithm makes it possible for all sites to agree on a target log without an explicit negotiation. We do not attempt to develop the details of a transaction-ordering algorithm in this paper, and we recognize that different algorithms may result in very different system behavior. We do assume that some such algorithm is used consistently within a distributed database system.

A transaction Ti a-precedes another transaction Tj if and only if Ti precedes Tj according to an algorithm a. (We use "a-precede" rather than "precede" to emphasize that we are not necessarily dealing with actual time but that precedence may be different for different algorithms.) Throughout this paper, we assume that some algorithm a is used consistently, so we simply write "Ti a-precedes Tj" as "Ti<Tj". Unless otherwise stated, we simplify notation by assuming that, given transactions Ti and Tj, Ti<Tj.

### 1.2 Correctness of the Final Merge State

The target log is the key to our notion of correctness. Operating on replicated data during a network partition plays havoc with the idea of transaction commitment. We never commit a transaction during a partition; any transaction may be backed out and re-run during the merge process, and it may have a different result when re-run than when run during the partition. What we do guarantee is that every transaction run during the partition is included in the target log and that these transactions appear in a predetermined order (a-precedence). We neither omit partitioned transactions from the target log nor change the order within the target log to simplify the merge process.

Further, even though transactions are not considered committed during a partition, they may reflect actions in the real world that cannot be changed. For example, once a person has withdrawn money from an automatic teller, it is not possible to change the fact that the money has physically been transferred from the bank to the customer. For some applications, it is preferable to compensate for actions taken during a partition than to prohibit transactions during the partition. If the customer withdrew more than his balance because the local copy of the balance was not up-to-date during a partition, the bank may perform the compensating action of requesting payment of the amount overdrawn plus a penalty charge. Therefore, sites may need to initiate compensating transactions when certain situations (such as a negative balance) are detected. Compensating transactions serve the purpose of counteracting the loss of serializability due to operation during the partition: during a partition actions may be taken which do not reflect a globally serial execution of transactions; but when communications are restored, these situations will be detected and some new compensating actions will be taken. The problem of multiple sites issuing compensations for the same situation can be handled either by making all compensations idempotent (i.e., writes that are independent of the database state) or by using some "election" protocol by which one site (or its proxy) is designated to issue some class of compensating transactions. A database state in which such compensations are reflected is deemed to be consistent. Thus, instead of the consistency constraint that no balance go below zero, the constraint would be that if a balance is below zero, a request for payment is issued. (We assume that the database would include a list of requests issued.)

To sum up, then, mutual consistency is guaranteed by requiring that all sites eventually reach the final merge state. Internal consistency is maintained by issuing any necessary compensating transactions. The final merge state is defined through the target log, and the target log is required to include all partitioned transactions in a globally agreed-upon order.

## 2. Restoring Divergent Database States

### 2.1 Initial Merge Logs

Once each site is informed about all the partitioned transactions, it can independently generate the final merge state. To achieve the final merge state, each site generates and executes a different merge log. This section shows how a site generates an initial merge log and then transforms it to produce a more efficient final merge log.

To achieve the final merge state it may be necessary for a site to undo, or rollback, the effects of a local partitioned transaction. When this occurs, a rollback transaction will be included in the final merge log. The rollback transaction used may depend on the state in which the original transaction was executed.

A transaction Ti' is called the rollback of a transaction Ti iff given the state D in which Ti was executed D[Ti,Ti'] = D. When Ti' is a rollback of Ti in any database state (i.e., [Ti,Ti'] = []), it may also be called an inverse of Ti.

Merge logs contain two categories of transactions: rollback transactions for transactions that appeared in the site's partition log, and forward transactions that appear in the target log. (Forward transactions include local and non-local partitioned transactions.) In what follows, Ti, Tj, etc. refer to any transaction, either forward or rollback. If we need to refer specifically to a forward transaction we use Ti~; a rollback transaction is denoted by Ti'. We match forward and rollback transactions by their subscripts: a transaction denoted Ti' is the rollback for the transaction denoted Ti~.

A merge log M is correct for an initial state D, a partition log P, and a target log T iff D([P,M]) = D(T), the final merge state.

Using this definition, then, our goal is to execute correct merge logs at each site. The interesting problem, and the focus of the rest of the paper, is to generate efficient correct merge logs.

Once the target log is defined, it is trivial to define a correct merge log for each site. Suppose we have a target log T and an initial state D. Then, for a site X with partition log [T1,...,Tn], a correct merge log is ([Tn',...,T1'],T) (i.e., [Tn',...,T1'] concatenated with T), where T1',...,Tn' are the rollback transactions for T1,...,Tn. This merge log is called the initial merge log for X. (The action prescribed by the initial merge log is similar to cycle breaking in optimistic concurrency control; see for example [Badal79, Bhargava82, Davidson82, Kung79].)

To define an efficient merge log, each site begins with its initial merge log and exploits pre-defined semantic properties of the transactions in the log to transform it into a shorter, and so more efficiently executed, equivalent final merge log. The merge log at each site can be transformed independently of the others.

## 2.2 Log Transformations

Our goal in transforming the initial merge log is to delete transactions that have no effect on the final state produced. We do this by exploiting semantic properties of the transactions in the target log. Consider the following two examples:

1. Suppose that during a partition site X processes transaction T1, fixing Boston as the location of delivery truck 458; and site Y processes transaction T2, fixing Annapolis as the location of the same delivery truck. (We assume that the ordering algorithm places T1 previous to T2, yielding the target log [T1,T2].) At site X, the partition log is simply [T1], so the initial merge log would be [T1',T1,T2]. But notice that this log can be shortened without affecting the resulting database state: the log says to rollback T1 and immediately rerun it -- therefore having no effect on the database state. So, we can delete both T1' and T1 from the log, leaving the final merge log [T2]. For site Y, the situation is even better. Note that since T2 follows T1 in this target log, "Annapolis" overwrites "Boston;" i.e., the log [T1,T2] = [T2]. Therefore, rather than executing the initial merge log [T2',T1,T2] at site Y, we can simply execute the log [T2',T2]. By the reasoning above, we can then delete both T2' and T2 from the merge log -- site Y needs to do nothing to achieve the final merge state.

2. Suppose that, during a partition, site X records the shipment of 50 widgets to the central warehouse by processing transaction T3: add 50 to the number of widgets in the warehouse. During the same partition site Y processes transaction T4, adding 75 to the number of widgets in the warehouse. Suppose that the ordering algorithm results in the target log [T3,T4]. Although the initial merge log for site Y is [T4',T3,T4], we know that T3 and T4 commute, that is [T3,T4] = [T4,T3]. Therefore, we can transform the initial merge log to be [T4',T4,T3]. Just as in the previous example, then, we can then delete T4' and T4, leaving the final merge log [T3]. Similarly for site X. Thus, each site merely executes the partitioned transaction from the other site to obtain the final merge state.

To help determine which transactions safely may be deleted, we characterize ordered pairs of transactions. The order of the transactions in a merge log determines which ordered pair we use to transform that merge log: if Ti a-precedes Tj in a merge log we consider the characterization of the pair (Ti,Tj). We consider every ordered pair of transactions to be either an overwrite pair, a commutative pair, or a conflicting pair.

- An ordered pair of transactions (Ti,Tj) is an overwrite pair (Tj overwrites Ti) iff [Ti,Tj] = [Tj].

- The ordered pair (Ti,Tj) is commutative iff

  1. [Ti,Tj] = [Tj,Ti]

  2. neither (Ti,Tj) nor (Tj,Ti) is an overwrite pair. For example, the transactions

     T1: if x > 0 then x := x + 1
     T2: x := 0

     do not fulfill this condition and, even though [T1,T2] = [T2,T1], (T1,T2) is not characterized as commutative.

- The ordered pair (Ti,Tj) is conflicting iff it is neither an overwrite pair nor a commutative pair.

To determine whether a particular pair of transactions is an overwrite pair, a commutative pair, or a conflicting pair, we must of course analyze the actual values of the transactions' arguments and the values read by the transactions. This analysis may be very detailed, or it may be nothing more than a "worst case" based on the transactions' types and their roles (rollback or forward) and positions in the merge log. More study is needed to decide how much analysis should be done at the time of the database merge, as opposed to the basic characterization done when a transaction type is defined, and to determine when the cost of analyzing transactions offsets the losses involved in using the "worst case" analysis. In any case, once pairs of transactions are characterized, the characterizations can be used to delete transactions from the initial merge log.

We begin with the initial merge log and successively delete transactions to yield new merge logs. There are two main situations in which we delete transactions. The first is when transactions in an overwriting pair are adjacent; deleting the overwritten transaction preserves the correctness of the merge log. The second situation is a bit more complicated. Under certain conditions, we can show that when a rollback transaction is immediately followed by its matching forward transaction, the two transactions taken together do not affect the final database state: both can be deleted. When a log meets the conditions necessary for this double deletion, we say it is regular.

A merge log [T1,...,Ti,Tj',...] that is correct for target T and partitioned state Dx is regular iff for each rollback transaction Tj' in the log, Dx[T1,...,Ti,Tj',Tj~] = Dx[T1,...,Ti].

Essentially, regularity ensures that rollback transactions operate correctly. Each rollback transaction is responsible for restoring values that existed in a particular state during the partition, and the order in which these transactions are executed can be critical. Regularity enforces this order by prescribing that each rollback transaction has the correct effect, i.e. that its effects are nullified by running its corresponding forward transaction immediately following. Each log transformation, then, must preserve regularity (the role of regularity in governing the interaction of transformations is discussed in Section 3.3).

Commutativity is also essential for log transformations. Given a regular merge log L, we use commutativity to find (whenever possible) an equivalent log L' in which either transactions in an overwriting pair are adjacent, or in which a rollback and its matching forward transaction are adjacent. The definition of an overwriting pair or the definition of regularity allows us to delete either the overwritten transaction or the rollback and its forward transaction, respectively. This deletion produces a new merge log L". In the next section we prove that, as long as some minor restrictions are respected, L" is both correct and regular; therefore the process of deleting transactions may continue.

## 3. Transforming Merge Logs Using Graphs

### 3.1 Merge Log Graphs

We use a graph to describe the properties that hold between pairs of transactions in a merge log. Graph transformations that delete nodes in the graph mimic log transformations that delete transactions in the corresponding merge log. After defining the graph of a merge log, we define some graph transformations and prove that they are correct. The graph transformations presented here are not an exhaustive set, but we believe that they are among the most generally applicable and powerful.

Merge Log Graph Definition: The graph of a merge log L, Graph(L), is the directed acyclic graph (N,O u C u A), where

- N is a set of nodes representing transactions that appear in L (for simplicity nodes have lower case names that correspond to the transactions they represent -- so, node ti represents transaction Ti, tj' represents Tj', etc.).
- O (for "overwrite") is the set of directed edges {(ti,tj)|ti and tj are nodes in N, Ti a-precedes Tj in L, and Tj overwrites Ti}.
- C (for "conflict") is the set of directed edges {(ti,tj)|ti and tj are nodes in N, Ti a-precedes Tj in L, and Tj conflicts with Ti}. (It is occasionally useful to distinguish certain conflict edges in each merge log graph: R (for "read") is the set of conflict edges (t1,t2) such that T2's read-set intersects T1's write-set. These edges will be used in the RDD Transformation Rule, Section 3.2.)
- A (for "added") is the union of the set of directed edges.

  {(ti',tj~)|ti' and tj~ are nodes in N, Ti' a-precedes Tj~ in L, (ti',tj~) is neither an overwrite nor a conflict edge, and there is an overwrite or conflict edge between ti~ and tj~ (Ti~ is the transaction rolled back by Ti')}.

  and the set of directed edges.

  {(ti',tj')|ti' and tj' are nodes in N, Ti' a-precedes Tj' in L, (ti',tj') is neither an overwrite nor a conflict edge, and there is an overwrite or

conflict edge between ti~ and tj~}.

This restricted use of commutativity is similar to [Graham84], and its necessity is illustrated later in Section 3.3.

We use dotted arrows to designate overwrite edges, double solid arrows for conflict edges, and single solid arrows for added edges (O-, C-, and A-edges respectively). Figure 2 is an illustration of a merge log graph.

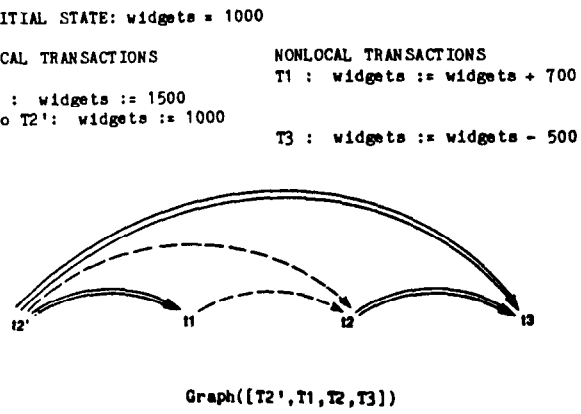



Figure 2. Example of a Merge Log Graph

There is a path from a node ti to a node tk iff: 1) there is an edge (of any type) from ti to tk (a trivial path), or 2) there is some node tj such that there is an edge from ti to tj and a path from tj to tk. In the previous example, then, there is a path from t1 to t3.

We will only talk of equality between graphs representing the same transactions, so because of our naming convention for nodes, two graphs G = (N,O u C u A) and G' = (N',O' u C' u A') will be equal, G = G', iff N = N', O = O', C = C', and A = A'.

Merge log graphs define special sets of equivalent logs: logs comprised of the same group of transactions, but showing every permutation of order allowed by commutativity. Our transformation strategy, then, is to start with a merge log L and find a log L' such that Graph(L) = Graph(L'), where L' includes a particular pair of transactions adjacent to each other. We then show that one or both of these transactions can be deleted from L' to produce an equivalent log L". We continue this process until we can no longer delete transactions.

To pursue this strategy, we must show that we may pick any log L' such that Graph(L) = Graph(L') without compromising correctness.

A merge log graph is correct for an initial database state D, a partition log P, and a target log T if it is the graph of a correct merge log for D, P, and T.

Graph Correctness Theorem: Let G be a graph that is correct for an initial database state D, a partition log P, and a target T. Then every log L such that Graph(L) = G is correct for D, P, and T.

We rely on commutativity to find equivalent merge logs in which a particular pair of transactions are adjacent. The following lemma describes this situation in terms of a merge log graph.

Commutativity Lemma: If there is no nontrivial path between two nodes ti and tj in a merge log graph Graph(L), then there is a log L' = L, such that Graph(L') = Graph(L), in which Ti and Tj are adjacent.

As we mentioned above, and will illustrate below, the notion of regularity is essential to the transformation strategy. Here we extend this notion to graphs and show that every log with a graph equal to the graph of an initial merge log is regular. We will prove that the transformations preserve regularity.

A merge log graph G is regular if every log L such that Graph(L) = G is regular.

Initial Regularity Theorem: Let L be the initial merge log for an initial database state D, a partition log P, and a target log T. Then Graph(L) is regular.

In the next section we define three merge log graph transformations which allow us to delete nodes from the graph while preserving correctness.

## 3.2 Graph Transformations

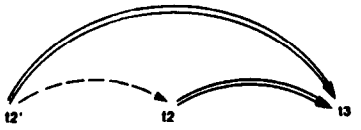
The graph transformations in this section mimic the log transformations illustrated above. Overwritten transactions, on which no other transactions depend, are deleted, as are pairs of rollback and forward transactions that have no net effect on the log. All the transformations preserve the correctness of the merge log, and regularity ensures that they may be applied in any order. However, following the heuristic of deleting pairs of transactions, instead of single overwritten transactions, whenever there is a choice will result in a shorter final merge log. We are currently working on other similar heuristics.

The transformation rule which deletes overwritten transactions is the most simple, and so it is presented first.

Overwrite Deletion (OD) Transformation Rule: Let G=(N,O u C) be a merge log graph with two nodes ti and tj~ such that the only path from ti to tj~ is a single overwrite edge. OD(G) = (N',O' u C' u A') where:

N' = N-{ti}
O' = {(tk,tm)|(tk,tm) is an element of O, and tk and tm are in N'}
C' = {(tk,tm)|(tk,tm) is an element of C, and tk and tm are in N'}
A' = {(tk',tm)|(tk',tm) is an element of A, and tk' and tm are in N'-{ti'}}

The graph in Figure 2 has two overwrite edges, but only nodes t1 and t2 meet the criterion in the OD Transformation Rule. The rule states that we can delete node t1, yielding the graph in Figure 3.




Graph in Figure 2
After OD Transformation

Figure 3. Example of OD Transformation

The proofs that this rule and the ones that follow preserve graph regularity are omitted here; they appear in [Blaustein85].

The next two transformation rules prescribe the deletion of pairs of rollback and forward transactions that have no net effect on the final database state. The first rule is actually a special case of the second, but one which comes up sufficiently often to merit a special discussion. Because the special case is more simple, it is presented first.

Double Node Deletion (DND) Transformation Rule: Let G=(N,O u C u A) be a regular merge log graph such that there are nodes ti' and ti~ with no non-trivial path between them. DND(G) = (N',O' u C' u A') where

N' = N-{ti',ti~}
O' = {(tk,tm)|(tk,tm) is an element of O, and tk and tm are in N'}
C' = {(tk,tm)|(tk,tm) is an element of C, and tk and tm are in N'}
A' = {(tk',tm)|(tk',tm) is an element of A, and tk' and tm are in N'}

The graph in Figure 4 meets the condition of the the DND Transformation Rule; t2' and t2 can be deleted.

INITIAL STATE: widgets = 500

LOCAL TRANSACTIONS

T2 : widgets := widgets + 200
so T2': widgets := 500

NONLOCAL TRANSACTIONS

T1 : widgets := widgets + 300


Graph([T2',T1,T2])

Figure 4. Example of DND Transformation

With a little additional information, this rule can be relaxed to allow double deletions in more cases. (The DND transformation is a special case of the Relaxed Double Deletion (RDD) rule below.) Using R-edges (the set of C-edges (t1,t2) such that T2's read-set intersects T1's write-set), and keeping track of which nodes represent local transactions and which non-local, we can better determine which transactions have the same effects when re-run as they did when originally run during the partition. A local node tj is a repeater if there is no path consisting entirely of R-edges from a non-local node to tj. Essentially, then, a repeater has the same effect when run during the merge as it had when run during the partition, i.e., repeaters read the same values both times. With the notion of repeaters, we can relax the DND Transformation Rule as follows:

Relaxed Double Deletion (RDD) Transformation Rule: Let G=(N,O u C u A) be a regular merge log graph having nodes ti' and ti~ such that:
for every node tj such that (tj,ti~) is in C,

1. either tj is a repeater or tj is overwritten by a repeater tk with an edge to ti~, and

2. all paths from ti' to tj include an A-edge

Then RDD(G) = (N',O' u C' u A') where

N' = N-{ti',ti~}
O' = {(tk,tm)|(tk,tm) is an element of O, and tk and tm are in N'}
C' = {(tk,tm)|(tk,tm) is an element of C, and tk and tm are in N'}
A' = {(tk',tm)|(tk',tm) is an element of A, and tk' and tm are in N'}

Figure 5 shows an example of the RDD rule (nodes t5' and t5 to be deleted). Only the most relevant edges are shown for clarity and R-edges appear as thick black arrows. Node t4 is a repeater. There are conflict edges from t4, t2, t4' and t6' to t5; since t4 overwrites t2, t4' and t6', the first condition of the RDD rule is met. Since all paths from t5' to these nodes include an A-edge, the second condition is met as well. Therefore, t5' and t5 may be deleted. Note that this transformation now makes it possible to use the fact that T6 overwrites T4 to delete t4.

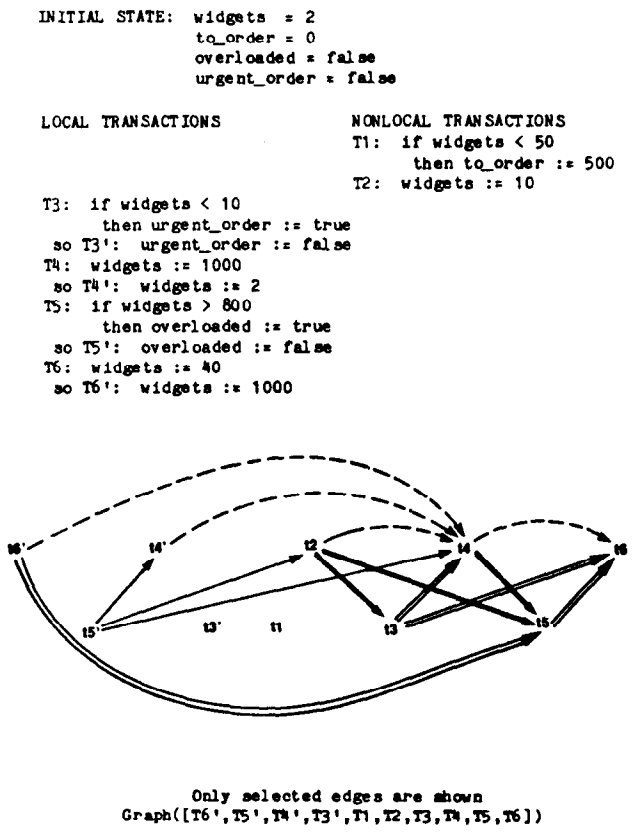



Figure 5. Example of RDD Transformation

The next section discusses how the definitions of A-edges and regularity ensure that the log transformations presented here work together correctly.

### 3.3 Interactions of Definitions

So far, we have considered separately each step in the process of transforming an initial merge log into a final one. Presented this way, the reasons for some of the definitions, especially regularity and A-edges, may have seemed somewhat peculiar. There are important, and sometimes subtle, interactions among the different steps of the transformation process, and these interactions motivate the definitions of the transformation rules, commutativity, and regularity. This section illustrates the way in which the steps described in previous sections combine to transform merge logs.

At a high level, the process of log transformation begins with the following steps at each site.

1. Define the target log
2. Use the target log to generate the initial merge log
3. Use semantic properties of transactions to graph the initial merge log
4. Apply the OD, DND, and RDD graph transformations until no more transformations are possible

All logs represented by the resulting graph are guaranteed to be correct.

The definitions of commutativity, A-edges, and regularity are essential to ensure that all the steps work together correctly: that each transformation leaves the graph (log) in a state in which any other applicable transformation will produce a correct graph (log). In particular, A-edges were introduced to ensure that the transformations result in regular graphs. Once all other transformations are made, however, A-edges may be ignored in a final step which transforms the log into a correct, but not regular, final merge log. This section gives some examples that illustrate the way in which the definitions of the transformations, added edges, commutativity, and regularity interact, and it defines the final step in the log transformation process.

A-edges impose necessary restrictions on the transformations. In the example in Figure 6, the added edge between t2' and t1 serves to ensure that the DND Transformation Rule does not apply to t2' and t2, thus guaranteeing that T2 reads the value of widgets written by T1.

INITIAL STATE: to_order = 100

| LOCAL TRANSACTIONS | NONLOCAL TRANSACTIONS |
|---|---|
| | T1: widgets := 1500 |
| T2: to_order := 5000 - widgets | |
| so T2': to_order := 100 | |
| | T3: widgets := 1200 |

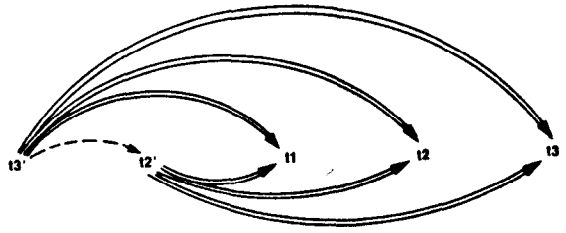

Graph([T2',T1,T2,T3])

Figure 6. Necessity of A-Edges

The notion of regularity ensures that forward transactions see the effects of their own rollback transactions. The example in Figure 7 illustrates the problem that occurs when regularity is not maintained (for example, by ignoring the requirement in the OD Transformation Rule that only forward transactions may overwrite). T2' overwrites T3', but deleting t3' "orphans" t3: it can no longer be deleted using the DND Transformation. T3 would then be included (without T3') in the incorrect final merge log.

INITIAL STATE: widgets = 1000

| LOCAL TRANSACTIONS | NONLOCAL TRANSACTIONS |
|---|---|
| | T1: widgets := widgets + 100 |
| T2: widgets := widgets + 300 | |
| so T2': widgets := 1000 | |
| T3: widgets := widgets - 700 | |
| so T3': widgets := 1300 | |




Graph of initial merge log [T3',T2',T1,T2,T3]

Figure 7. Use of Regularity

The preceding examples illustrate some restrictions that ensure that successive transformations operate correctly. These restrictions, though, may be eased in a final pass through the log. Certainly, rollback transactions may be used to overwrite other rollback transactions as long as all OD, DND, and RDD transformations have been made. Furthermore, the A-edges have no more purpose in such a case, and they may be ignored in performing final overwrites (notice that the overwritten transactions would all be rollbacks). In Figure 6, for example, while the A-edge from t2' to t1 is important, there is no reason to include T2' in the final merge log: no transaction reads the value of to_order and it is overwritten by T2. Therefore, t2' can be deleted in the final transformation.

So, the final step in the transformation process is:

5. Ignoring A-edges, delete all rollback transaction nodes ti' such that the only path between ti' and a node tj is a single O-edge

## 4. Conclusion

The need to update replicated data during network partitions requires different correctness criteria and update protocols than may be used for distributed databases in which blocking is tolerable. In our approach, each site acts independently to revise replicated data when it receives information about relevant updates that were made at other sites. The information may describe updates that were made previous to other updates already reflected in the site's database, and thus it may be necessary to roll back or rerun some transactions to merge these updates. Depending on the resulting values, the consistency constraints may prescribe issuing some compensating transactions. We rely on a system-wide ordering algorithm to ensure that each site will eventually reach the same database state when it has been informed of the same set of updates.

The main thrust of the research is to merge local transaction logs with logs of new updates efficiently, by exploiting simple semantic properties of pre-defined transactions. The concepts of regularity and added edges are used to capture critical and sometimes subtle dependencies among transactions. More work is needed to determine new and useful transformations (in particular, we are beginning work on transactions which overwrite only part of the data written by another transaction), to investigate modifications needed for partially replicated data, and to design efficient implementations of these strategies.

Acknowledgments

Sunil Sarin and Umesh Dayal have been closely involved with our work; they have contributed greatly to the direction and presentation of our work. Dan Ries and Hector Garcia-Molina were instrumental in the development of many of the ideas presented here. We thank the VLDB Program Committee for their valuable comments.

## 5. References

[Alsberg76]
Alsberg, P. and Day, J. "A Principle for Resilient Sharing of Distributed Resources," Proc. of the Second Intl. Conf. on Software Engineering, Pittsburgh, October 1976.

[Badal79]
Badal, D. "Correctness of Concurrency Control and Implications in Distributed Databases," Proc. COMPSAC 79 Conference, Chicago, November 1979.

[Bhargava82]
Bhargava, B. "Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems," Proc. of the Second Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, July 1982.

[Blaustein83]
Blaustein, B., Garcia-Molina, H., Ries, D., Chilenskas, R., and Kaufman, C. "Maintaining Replicated Databases Even in the Presence of Network Partitions," Proc. of the IEEE EASCON Conference, Washington, D.C., September 1983.

[Blaustein85]
Blaustein, B. and Kaufman, C. "Distributed Updates Without Blocking," CCA Technical Report, to appear.

[Davidson82]
Davidson, S. "An Optimistic Protocol for Partitioned Distributed Databases," PhD Thesis, Dept. of Electrical Engineering and Computer Science, Princeton University, August 1982.

[Davidson84]
Davidson, S.B., Garcia-Molina, H. and Skeen, D. "Consistency in a Partitioned Network: A Survey," Technical Report MS-CIS-84-04, University of Pennsylvania, August, 1984.

[Fischer82]
Fischer, Michael J. and Michael, Alan. "Sacrificing serializability to attain high availability in an unreliable network," ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1982, pp. 70-75.

[Gifford79]
Gifford, D. "Weighted Voting for Replicated Data," Operating Systems Review, Vol. 13, No. 5, December 1979.

[Graham84]
Graham, M., Griffeth, N., and Moss, E. "Recovery of Actions and Subactions in a Nested Transaction System," Technical Report GIT-ICS-84/12, Georgia Institute of Technology, March 1984.

[Kung79]
Kung, H. and Robinson, J. "On Optimistic Methods for Concurrency Control," Proc. VLDB 1979, Rio de Janeiro, October 1979.

[Lamport78]
Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. of the Assoc. for Computing Machines, Vol. 12, No. 7, July 1978.

[Sarin85]
Sarin, S., Blaustein, B., and Kaufman, C. "System Architecture for Partition-Tolerant Distributed Databases," to appear.

[Skeen84]
Skeen, D. and Wright, D. "Increasing Availability in Partitioned Database Systems," Proc. of the Fourth Annual ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, Waterloo, Ontario, April 1984.

[Stonebraker78]
Stonebraker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," Proc. of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, August 1978.

[Thomas79]
Thomas, R. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979.