# Replication and Consistency: Being Lazy Helps Sometimes

**Yuri Breitbart**
Bell Laboratories
Lucent Technologies Inc.
700 Mountain Avenue
Murray Hill, NJ 07974
yuri@bell-labs.com

**Henry F. Korth**
Bell Laboratories
Lucent Technologies Inc.
700 Mountain Avenue
Murray Hill, NJ 07974
hfk@bell-labs.com

## Abstract

The issue of data replication is considered in the context of a restricted system model motivated by certain distributed data-warehousing applications. A new replica management protocol is defined for this model in which global serializability is ensured, while message overhead and deadlock frequency are less than in previously published work. The advantages of the protocol arise from its use of a lazy approach to update of secondary copies of replicated data and the use of a new concept, *virtual sites*, to reduce the potential for conflict among global transactions.

## 1 Introduction

The problem of consistent access to replicated data has re-emerged as a challenge in recent years [CRR96, GHOS96, HHB96, PL91, SAB+96] with the advent of distributed data warehouses and data marts at the high end, and distributed data in often-disconnected mobile computers at the low end [KI96]. The fundamental problem, as identified by [GHOS96], is that the standard transactional approach to the propagation of updates to replicas is unstable – deadlocks increase as the cube of the number of network sites and as the fourth power of transaction size. This is particularly problematic with relatively long data-mining queries and with mobile transactions. The former access many data items; while the latter effectively live for a long period of time if the mobile computer is disconnected. Thus, deadlock is no longer a rare event with a negligible effect on performance; instead, it is a barrier to the ability of systems to scale up.

To ameliorate this problem, one may dispense with traditional "eager" propagation strategies (see, e.g., [Hol81] for a survey), and employ instead a "lazy" approach. Under lazy propagation, only one replica is updated by the transaction itself. A separate transaction runs on behalf of the original transaction at each site at which update propagation is required. Lazy propagation effectively reduces transaction size but creates the possibility of two or more transactions committing conflicting updates to a data item if they operate on distinct replicas. For example, $T_1$ could update data item $d$ using the replica at site $s_1$ while $T_2$ updates the replica of $d$ at $s_2$. Assume that both transactions commit. Only when updates are propagated is the conflict discovered by the system. Such conflicts require either the use of compensating transactions [KLS90] or update reconciliation. Consistency can be ensured despite lazy propagation by directing all updates to a primary copy (called the lazy-master approach to update regulation in [GHOS96]), and employing an appropriate concurrency-control protocol. "Appropriate" is the key word in the preceding sentence since lazy propagation may cause an update transaction to read "old" replicas of some data, resulting in an execution that generates an inconsistent database state, as the following example illustrates:

**Example 1:** Consider a bank database for checking and savings accounts that is distributed over two sites, $s_1$ and $s_2$. Site $s_1$ contains the primary copy of the checking-account relation and a replica of the savings-account relation. Site $s_2$ contains the primary copy of the savings-account relation and a replica of the checking-account relation. The bank requires only that the sum of a customer's checking and savings accounts be positive.

Suppose that a husband and wife have joint checking and savings accounts and the current balances in these two accounts are \$300 and \$700, respectively. The husband withdraws \$900 from the checking account using an ATM at $s_1$ and, at approximately the same time, the wife withdraws \$900 from the savings account using an ATM at $s_2$. Due to the delay in update propagation resulting from the lazy approach, both transactions may succeed. However, after the updates are propagated, both accounts have a negative balance, violating the bank's constraint that the sum of the balances must be positive. □

To avoid anomalies such as the one illustrated above and nevertheless guarantee global serializability, the


*PODS* '97 Tucson Arizona USA

lazy-master approach must be augmented with one of the following:

- Restrictions on how primary copies of data are selected[CRR96].
- A global concurrency-control mechanism that minimizes coordination among sites.

In this paper, we choose the latter approach and present an alternative approach to lazy propagation that has fewer deadlocks than the approach of [GHOS96], permits local read-only transactions to run without the need to acquire global locks, permits substantial use of the local database system's concurrency control for update transactions (for increased efficiency over global locking), and reduces the distributed transaction management problem to that of maintaining a globally consistent graph (which we call the *replication graph*).

In terms of the classifications defined in [GHOS96], we use master permission regulation and lazy update propagation. Previous protocols for managing updates within these assumptions [GHOS96] either fail to guarantee both consistency and atomicity of the database or are subject to a prohibitive number of deadlocks, or both. We present a protocol based on our replication graph that ensures global serializability.

Site failures in our approach are handled completely by relying on the local database system's recovery manager. Singleton network partitions (as arise from the disconnection of a mobile computer) are easily managed. Only in the case of general partitions do we need to resort to blocking or reconciliation.

## 2 Related Work

Initial work on replicated databases has concentrated on the issues of how to guarantee global serializability and atomicity in the presense of site failures and communication failures[BHG87]. Global serializability can be achieved by using a distributed version of any protocol that guarantees serializability, such as two-phase locking or timestamp protocols[BHG87], in combination with one of the replica update propagation schemes (read-one, write-all or read-any, write-all-available, etc.). To ensure atomicity despite failures, the two-phase commit protocol is used. The various published protocols vary in their degree of central control and in the specific techniques used[Ell77, Tho78, GS78, Min79]. Such approaches guarantee the ACID properties[GR93]. The problem, however, is that they are susceptible to deadlocks, transactions aborts, and site blocking. As the number of sites, data items, and the degree of replication grow, the frequency of these undesirable effects dramatically rises

Recently, Gray et al.[GHOS96] have extended that work by proposing a taxonomy of replication management strategies, based on where an update may be initiated and how an update on one copy is propagated to the other copies. In particular, the paper introduces:

- **Regulation:** *Group* permission, in which any site holding a replica may initiate an update, versus *master* permission, in which only the primary site for the data item may initiate an update to that data item.
- **Propagation:** *Eager* propagation by the update transaction itself, versus *lazy* propagation by a separate asynchronous transaction.

The [GHOS96] protocol is discussed in detail in Sections 5 and 6.

Several authors considered the issues of global serializability and ensuring atomicity without an atomic commit protocol [CRR96, SAB+96]. Their approach is based on either group or master permission and lazy replica propagation. The problems of deadlock, blocking, and transaction aborts are present in this approach as well. More importantly, there are potential data inconsistencies resulting from certain copies of replicated data items holding obsolete data. Thus, there must be a mechanism that ensures data item replica convergence [GHOS96].

## 3 System Model and Assumptions

Our system model is based on a data-warehousing application in which large databases (up to a few terabytes) are connected by a wide-area network. Our motivating application uses much of the network capacity for the actual transfer of updates to sites holding replicas. Thus, network bandwidth is a scarce resource and the round-trip time for a message and acknowledgement is relatively large. Each site runs a database system locally that can ensure the usual ACID properties [GR93] (including serializability).

Transactions that run at only one site are called *local transactions*, while those that run at multiple sites are called *global*. Each transaction is restricted to read data at only one site – the site at which it originates. Thus, all read-only transactions are local transactions. This restriction is a consequence of the assumed network properties. In practice, if a transaction needs to read data not available locally, a local replica can be created outside of the transaction's execution, but we do not concern ourselves with that issue here.

Each data item has a primary copy located at a particular site, called its *primary site*. Only transactions originating at a data item's primary site may update that data item. A variety of applications that we have studied fit within these restrictions; and, as we shall see, this restriction is key to the power of our replica management protocol. Generally, any application in which each data item has a specific "owner" fits within our restrictions.

A transaction that updates at least one replicated data item is a global transaction. A global transaction is represented by several local sub-transactions – one for the transaction itself running at its origination site, and one for each site that holds replicas of one or more data items updated by the transaction. The sub-transactions running at remote sites on behalf of a global transaction do not begin executing until after the corresponding sub-transaction at the origination site has committed. For convenience of notation and discussion, we refer to all these transactions by the same name (e.g., $T_i$). Because of our lazy approach to update propagation, there is no global atomic commit; once $T_i$ has committed at its origination site, it eventually commits independently at the other sites at which it runs.

# 4 Replication Management Protocol

Since each site's local database system ensures serializability, we could view our problem as one of coordinating executions among these database systems. However, treating the local database systems as monolithic "black boxes" creates an artificially high degree of contention among global transactions [BGMS92]. For this reason, we divide each physical site into a dynamically changing set of *virtual sites* and our replication management protocol provides global transaction management over the set of virtual sites. Local transaction management within each virtual site is provided by the database system running at the physical site containing the virtual site. Because our protocol is part of an integrated system, we are able to use transaction management information from the local transaction managers, unlike the case for multidatabase systems[BGMS92].

## 4.1 Transaction States

In describing our protocol, we consider a transaction $T_i$ to be in one of the following 4 global states at any point in time:

- **aborted**, if $T_i$ has aborted at its origination site;
- **active**, if $T_i$ is active at its origination site;
- **committed**, if $T_i$ is committed at its origination site, but not yet in the **completed** state;
- **completed**, if at every site at which $T_i$ executed $T_i$ has committed and is not preceded (directly or indirectly) in that site's *local* serialization order by a transaction that has not completed.

In practice, there is, of course, a delay between the time at which a state transition occurs and the time remote sites are informed of the transition. While substantial delays of this sort would harm performance, we show in Section 7 that our protocol is robust in the face of arbitrary delays in the communication of state transitions.

If transaction $T_i$ is in the **active** or **aborted** state, then it has not executed any operations on replicated data items at any site except its origination site. From the **active** state, a transaction may transfer either into the **aborted** or **committed** state. Transactions cannot transfer directly into the **completed** state. When a transaction enters into the **aborted** state, it remains there.

If a global transaction is in the **committed** state, then it may have performed some of its operations on secondary copies. From the **committed** state, a transaction can be transferred only into the **completed** state. Observe that during the execution of a **committed** transaction at sites other than its origination site, the subtransaction at that site can be aborted by the local DBMS. However, it would be restarted and reexecuted at the site so that it eventually commits at all sites.

If a transaction $T_i$ has committed at all sites at which it executes, then it does not have any other operations. But it could be that some non-completed transactions precede $T_i$. Consequently, $T_i$ is not necessarily in the **completed** state as the following example demonstrates. In the example, we use the notation $r_i(d)$ to denote an operation in which transaction $T_i$ reads data item $d$. We use the notation $w_i^j(d)$ for a operation in which $T_i$ writes data item $d$ at site $s_j$. We denote that $T_i$ commits at site $s_j$ by $c_i^j$.

**Example 2:** Consider a database consisting of two sites: $s_1$ and $s_2$. Site $s_1$ contains the primary copy of data items $a$, $b$, and $c$. Site $s_2$ contains a secondary copy of $b$ and $c$. Consider the following three transactions:

$T_1$: $r_1(b)$, $w_1(a)$, $w_1(b)$ $\qquad$ $T_3$: $r_3(b)$, $r_3(c)$
$T_2$: $r_2(a)$, $w_2(c)$

Transactions $T_1$ and $T_2$ originate at $s_1$, while $T_3$ originates at $s_2$. Assume that the global execution order of the steps is as follows:

$$r_1(b),\ w_1^1(a),\ w_1^1(b),\ c_1^1,\ r_3(b),\ w_1^2(b),\ c_1^2,$$
$$r_2(a),\ w_2^1(c),\ c_2^1,\ w_2^2(c),\ c_2^2,\ r_3(c),\ c_3^2$$

Then, the following local schedules are generated at each site:

$s_1$: $r_1(b)$, $w_1(a)$, $w_1(b)$, $c_1$, $r_2(a)$, $w_2(c)$, $c_2$
$s_2$: $r_3(b)$, $w_1(b)$, $c_1$, $w_2(c)$, $c_2$, $r_3(c)$, $c_3$

It is simple to see that the above schedule is not globally serializable. $T_1$ precedes $T_2$ at $s_1$, while, at $s_2$, $T_2$ precedes $T_3$ which precedes $T_1$. At the point where $T_1$ has committed everywhere (just after $c_1^2$ in the global execution order), $T_3$ is still active. By our definition, $T_1$ is not in the **completed** state, although it is committed everywhere. If a global concurrency control protocol chose no longer to worry about $T_1$ at this point, it would not be possible to detect the nonserializability of the execution. Thus, our protocol retains such transactions until they enter the **completed** state. □

### 4.2 Virtual-Site Management

Each transaction has a virtual site associated with it at each physical site at which it executes. This virtual site exists from the time the transaction begins until our protocol explicitly removes it from consideration. We denote the virtual site for $T_i$ at physical site $s_j$ by $VS_{ij}$. The set of virtual sites is constructed and maintained based on the three rules below:

- **Locality rule.** We require that each local transaction execute at precisely one virtual site. Thus, local transactions have only one virtual site. A global update transaction, however, has several virtual sites – one at each physical site at which it executes.
- **Union rule.** At every point in time, $VS_{ij}$ must contain the set of data items at physical site $s_j$ that transaction $T_i$ has accessed[1] up to that point. If an access to a data item $d$ by $T_i$ causes a conflict[2] with $T_k$ at physical site $s_j$, then their virtual sites at site $s_j$ must be the same (i.e., $VS_{ij} = VS_{kj}$) and must contain all data at $s_j$ accessed so far by $T_i$ or by $T_k$.

The locality and union rules are requirements for correctness. The next rule is aimed at necessary performance improvements for the protocol to be practical. The power of the protocol arises from keeping virtual sites as small as possible. Thus, when a transaction $T_i$ enters the **aborted** or **completed** state, it is desirable to use this information to split or shrink virtual sites.

- **Split Rule.** When physical site $s_j$ determines that $T_i$ has entered either the **aborted** or **completed** state, any data items accessed exclusively by $T_i$ are removed from $VS_{ij}$ and the replication protocol need no longer consider $T_i$. If there is no $T_k$ distinct from $T_i$ such that $VS_{ij} = VS_{kj}$, this effectively removes $VS_{ij}$. Otherwise, we may recompute the virtual sites at site $s_j$ for all transactions $T_k$ such that $VS_{ij} = VS_{kj}$ using the locality and union rules. This computation can be optimized using transaction conflict information to reduce overhead.

As we shall see once we introduce the replication graph, keeping virtual sites small is critical to our protocol. The overhead of the split rule is entirely that of local processing on a per-site basis. We shall see that it pays significant dividends in terms of global concurrency in the distributed system.

### 4.3 Replication Graph

We associate a *replication graph* with an execution to represent conflicts arising from updates to replicated data. There is a single, global replication graph for the entire distributed system. For now, we assume perfect global knowledge of this graph, and relax this assumption later. A replication graph is an undirected bipartite graph $RG =< T \cup V, E >$, where $T$ is a set of transactions and $V$ is the set of all virtual sites for transactions in $T$. Edge $< VS_{ij}, T_k >$ belongs to $E$ if and only if $T_k$ performs a write operation on a replicated data item that is in $VS_{ij}$.

We say that $RG =< T \cup V, E >$ is a replication graph for a schedule $S$ if $T$ is the set of all transactions in $S$ and $V$ is constructed in compliance with the locality and union rules. We do not require that $E$ be empty initially. Thus, replication graph for a global schedule is not necessarily unique, as is shown by the following example:

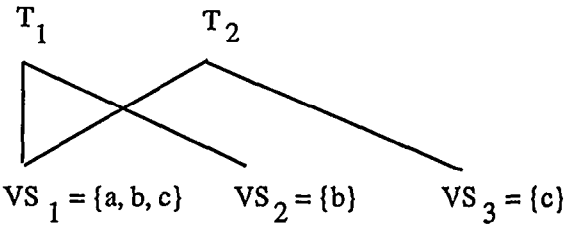


Figure 1: Acyclic Replication Graph for Example 3

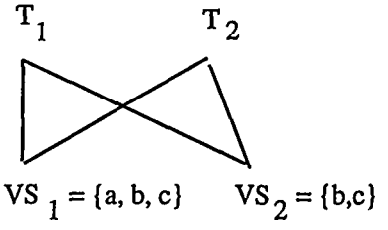


Figure 2: Cyclic Replication Graph for Example 3

**Example 3:** Consider a database consisting of two sites: $s_1$ and $s_2$. Site $s_1$ contains the primary copy of $a$, $b$, and $c$. Site $s_2$ contains a secondary copy of $b$ and $c$. Consider the following three transactions:

$T_1$: $r_1(b)$, $w_1(a)$, $w_1(b)$ $\qquad$ $T_3$: $r_3(c)$
$T_2$: $r_2(a)$, $w_2(c)$

Transactions $T_1$ and $T_2$ originate at $s_1$, while $T_3$ originates at $s_2$.

Then the following local schedules may be generated at sites $s_1$ and $s_2$:

$s_1$: $r_1(b)$, $w_1^1(a)$, $w_1^1(b)$, $r_2(a)$, $w_2^1(c)$
$s_2$: $w_2^2(c)$, $r_3(c)$, $w_1^2(b)$

A replication graph for global schedule given above is shown in Figure 1. An alternative replication graph is shown in Figure 2. The first of these graphs is acyclic while the second one is cyclic. □

[1] A transaction is said to access a data item $d$ at site $s$ if it has executed a read of $d$ at $s$ or has execute a write of any replica of $d$ regardless of site.

[2] We assume the usual notion of conflict among reads and writes.[BHG87]

**Theorem 1** *Let $S$ be a global schedule. If there is an acyclic replication graph for $S$, then $S$ is globally serializable.*

**Proof Sketch:** If $S$ is not globally serializable, then the union of the local serialization graphs must contain a cycle. Consider one such cycle, $T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow T_0$. Since each local site ensures serializability, there must be two or more global transactions in the cycle (otherwise the cycle is wholely contained within one local site). Let $T_i$ and $T_j$ be global transactions such that $i < j$, $T_i \rightarrow T_{i+1} \rightarrow \ldots \rightarrow T_{j-1} \rightarrow T_j$, and $\{T_{i+1}, T_{i+2}, \ldots, T_{j-1}\}$ are local transactions. Then $T_{i+1}, T_{i+2}, \ldots, T_{j-1}$ all execute at the same local site, which we denote $s_k$. Since $T_i$ and $T_{i+1}$ conflict, and $T_{i+1}$ is local to $s_k$, $T_i$ and $T_{i+1}$ conflict at $s_k$. Likewise, $T_j$ and $T_{j-1}$ conflict at $s_k$. Therefore, $VS_{ik} = VS_{(i+1),k} = \ldots = VS_{(j-1),k} = VS_{jk}$ and there exist edges in the $RG$ between this virtual site and both $T_i$ and $T_j$. A simple induction on the number of global transactions in the cycle (with 2 as a basis) proves that a cycle exists in any $RG$ for $S$. □

The above theorem provides a sufficient though not necessary condition for serializability. Non-necessity is shown by the example below:

**Example 4:** Consider a database consisting of two sites: $s_1$ and $s_2$. Site $s_1$ contains the primary copy of data item $a$. Site $s_2$ contains a secondary copy of $a$. Consider the following two transactions:

$$T_1\text{: } w_1(a)$$
$$T_2\text{: } w_2(a)$$

Transactions $T_1$ and $T_2$ originate at $s_1$. Assume that the following schedules are generated at each site:

$$s_1\text{: } w_1(a), w_2(a), c_1, c_2$$
$$s_2\text{: } w_1(a), w_2(a), c_1, c_2$$

Clearly, the execution is globally serializable. However, just prior to the commit of $T_1$ at $s_1$, the replication graph is cyclic. □

As Theorem 1 suggests, our interest in the replication graph will be in checking for cycles as our protocol generates schedule $S$. Thus, in practice, we do not need to keep an entire replication graph for $S$, but rather it suffices to maintain a dynamic replication graph with the property that avoiding a cycle in this graph suffices to ensure serializability.

From the replication graph definition, it follows that only global update transactions need be present among transactions nodes of the graph, since no edges could be incident upon other transaction nodes. Furthermore, it suffices to maintain the graph only over transactions that are in the **active** or **committed** states; that is, transactions can be ignored once they make a transition to either the **aborted** or **completed** states. Clearly, keeping virtual sites small and splitting them when possible reduces the frequency of cycles in the replication graph. We summarize these ideas in the statement of our Global Serializability (GS) Protocol below.

### 4.4 The Protocol and Its Properties

In this section, we state the Global Serializability (GS) Protocol using the above-defined notions of virtual sites and replication graphs. Following that, we state some results pertaining to the protocol. Our presentation is informal for the sake of brevity and intuitiveness.

---

**Protocol GS**

We begin by defining a test, which we call **RGtest**, that is applied by GS when a transaction $T_i$ submits an operation at its origination site. The test consists of tentatively applying the locality and union rules to virtual sites in the replication graph and tentatively adding any edges that would be mandated by the definition of the replication graph. If no cycle results, the test succeeds and the tentative changes to the replication graph are applied.

The protocol rules are as follows:

1. If $T_i$ submits a read or write operation at its origination site:
   - If **RGtest** succeeds, allow the operation to execute.
   - If **RGtest** fails and $T_i$ is local, $T_i$ submits the abort operation.
   - If **RGtest** fails and $T_i$ is global, test the tentative replication graph to see if any cycle includes a transaction in the **committed** state. If so, $T_i$ submits the abort operation, else the local subtransaction of $T_i$ waits.
2. If $T_i$ submits a write operation at a site other than its origination site, allow the operation to proceed.
3. If $T_i$ submits the commit operation, proceed with execution. If $T_i$ is in the **completed** state, remove it by deleting it from the replication graph (if it was present) and applying the split rule. Check whether any waiting transactions can be activated or aborted as a result of rule 1.
4. If $T_i$ submits the abort operation at its origination site, remove all edges incident on $T_i$ from the replication graph, and remove subtransactions of $T_i$ from any waiting queues in which they appear. Apply the split rule. Check whether any waiting transactions can be activated.

---

Protocol GS prevents the problem we illustrated in Example 1 by causing a cycle in the replication graph

at the point that the latter of the two transactions is attempted. To illustrate our protocol further, consider the following example:

**Example 5:** Consider a database consisting of two sites: $s_1$ and $s_2$. Site $s_1$ contains the primary copy of data items $a$, $e$, and $f$. Site $s_2$ contains the primary copy of data item $c$, and a secondary copy of data items $e$ and $f$. Consider the following five transactions:

$T_1$: $r_1(a)$, $w_1(f)$ $\quad$ $T_4$: $r_4(e)$, $w_4(c)$
$T_2$: $w_2(a)$, $r_2(e)$ $\quad$ $T_5$: $r_5(c)$, $r_5(f)$
$T_3$: $w_3(e)$

Transactions $T_1$, $T_2$, and $T_3$ originate at site $s_1$, while transactions $T_4$ and $T_5$ originate at site $s_2$. Note that $T_1$ and $T_3$ update replicated data whereas $T_2$ and $T_4$ are local update transactions (that is, they do not update replicated data). $T_5$ is a read-only transaction. Suppose that execution has proceeded at $s_1$ and $s_2$ as indicated below:

$s_1$: $r_1(a)$, $w_1(f)$, $c_1$, $w_2(a)$, $r_2(e)$, $c_2$, $w_3(e)$, $c_3$
$s_2$: $w_3(e)$, $c_3$, $r_4(e)$, $w_4(c)$, $c_4$, $r_5(c)$

The replication graph at this point is shown in Figure 3. In the figure, $VS_1 = VS_{1,1} = VS_{2,1} = VS_{3,1}$, and $VS_2 = VS_{3,2} = VS_{4,2} = VS_{5,2}$, and $VS_3 = VS_{1,2}$.

Observe that after $T_3$ has committed at both sites, it is not removed immediately from the graph. The reason is that $T_3$ is preceded by $T_1$ at site $s_1$ ($T_1 \rightarrow T_2 \rightarrow T_3$) and $T_1$ is not yet committed at both sites. Data item $e$, which is accessed exclusively by local transaction $T_2$ remains in virtual site $VS_1$ even after $T_2$ has committed. The reason is that $T_2$ is preceded at site $s_1$ by $T_1$ and $T_1$ is not yet committed at all sites. Likewise, data item $c$ remains in virtual site $VS_2$ after local transaction $T_4$ commits.

Suppose that $T_5$ submits its last operation, $r_5(f)$. In processing this operation, **RGtest** merges $VS_2$ with $VS_3$, thus creating a cycle. Therefore, **RGtest** fails and $T_5$ is aborted. Observe that the cycle included $T_1$, which had committed at $s_2$. Subsequently, $T_1$ may execute its replica update for $f$ ($w_1^2(f)$) at site $s_2$, all active transactions commit and complete, and $T_5$ can be restarted. □

A transaction cannot be removed from the replication graph until it enters the **completed** state, even if it has committed at all sites. Recall that we have shown the necessity of this in Example 2. In fact, while local transactions are never nodes of the replication graph, they play a role in deciding whether a global transaction can be removed from the graph. The next two examples illustrate how local transactions affect the execution of global transactions by merging virtual sites. The first example also demonstrates that the chain of everywhere-committed and non-completed transactions in the replication graph can be arbitrary long.

**Example 6:** Consider a distributed database located at $k$ sites: $s_1$, $s_2, \ldots,$ $s_k$. Site $s_i$ ($i = 2, 3, \ldots, k$) contains the primary copy of data item $a_i$ and a secondary copy of the data item $a_{i-1}$. Site $s_1$ contains the primary copy of $a_1$ and a secondary copy of $a_k$.

Consider the following set of $2k$ transactions:

$T_i$: $\quad w_i(a_i)$ $(i = 1, 2, \ldots, k)$
$T_{k+1}$: $r_{k+1}(a_1), r_{k+1}(a_k)$
$T_{k+i}$: $r_{k+i}(a_i), r_{k+i}(a_{i-1})$ $(i = 2, 3, \ldots, k)$

Transactions $T_i$ and $T_{k+i}$ originate at site $s_i$ ($i = 1, 2, \ldots, k$). Assume that the executions at sites $s_1$ and $s_2$ proceed as follows:

$s_1$: $w_1(a_1), c_1$
$s_2$: $w_2(a_2), r_{k+2}(a_2), r_{k+2}(a_1), c_{k+2}, w_1(a_1), c_1$

Assume that, at this point, transaction $T_1$ is removed from the replication graph. Observe that $T_1$ is not completed yet, since it is preceded in the serialization order at site $s_2$ by transaction $T_2$, which has not completed.

Suppose that execution proceeds and the following local schedules are generated at sites $s_1, s_2, \ldots, s_k$, respectively:

$s_1$: $w_1(a_1)$, $c_1$, $r_{k+1}(a_1), r_{k+1}(a_k)$, $c_{k+1}$, $w_k(a_k)$, $c_{k+1}$
$s_2$: $w_2(a_2)$, $c_2$, $r_{k+2}(a_2), r_{k+2}(a_1)$, $c_{k+2}$, $w_1(a_1)$, $c_1$
$\vdots$
$s_k$: $w_k(a_k)$, $c_k$, $r_{2k}(a_k), r_{2k}(a_{k-1})$, $c_{2k}$, $w_{k-1}(a_{k-1})$, $c_{k-1}$

The global schedule is not serializable, and would not have been generated under protocol GS. Under protocol GS, none of the transactions $T_1, T_2, \ldots, T_k$ could have been removed after they had committed, since none of them would be in the **completed** state. In this situation, when the operation $r_{2k}(a_{k-1})$ was submitted, protocol GS would abort $T_{2k}$ and the resulting schedule would be globally serializable. □

Observe that only global transactions can be present as nodes in the replication graph. However, read-only and local update transactions can delay removal of a global transaction from the replication graph for an arbitrarily long time, as demonstrated by the following example.

**Example 7:** Let $k > 0$ be an even number. Consider a database consisting of $(k+2)/2$ sites. Site $s_1$ contains a secondary copy of data items $a_0$, $a_1$, $\ldots$, $a_{k-1}$. Site $s_2$ contains the primary copy of data items $a_0$ and $a_{k-1}$. Any other site $s_i$ ($i = 3, 4, \ldots, (k+2)/2$) contains the primary copy of $a_{2(i-2)-1}$ and $a_{2(i-2)}$.

The following transactions originate at site $s_1$:

$$T_i\text{: } r_i(a_{i-1}), r_i(a_i) \ (i = 1, 3, \ldots, k-1)$$

The following transactions originate at site $s_2$:

$T_0$: $w_0(a_0)$
$T_k$: $w_k(a_{k-1})$
$T_{k+1}$: $r_{k+1}(a_0)r_{k+1}(a_{k-1})$

Note that $T_0$ and $T_k$ are global. The following transactions originate at site $s_j$ for $j = 3, 4, \ldots, (k+2)/2$:

$T_i$: $w_i(a_{i-1}), w_i(a_i)$ $(i = 2, 4, \ldots, k)$

Suppose that the following schedules at sites $s_1$, $s_2$, ... $s_{k-1}$ were generated just before the operation $r_{k+1}(a_{k-1})$ was submitted for execution (to simplify the presentation, we assume in this example that the transaction commits at the site as soon as it submits its last operation at the site):

$s_1$: $r_1(a_0), w_0(a_0), w_2(a_1), r_1(a_1), r_3(a_2), w_2(a_2), w_4(a_3), r_3(a_3), \ldots, r_{k-1}(a_{k-2}), w_{k-2}(a_{k-2}), w_k(a_{k-1})$
$s_2$: $w_0(a_0), r_{k+1}(a_0), r_{k+1}(a_{k-1}), w_k(a_{k-1})$
$s_j$: $w_i(a_{i-1}), w_i(a_i)$
$(i = 2, 4, \ldots, k$ and $j = 3, 4, \ldots, (k+2)/2)$

None of the global update transactions have been removed by Protocol GS, since each time the global transaction is committed at all sites, there is either a read-only transaction that precedes it and is not completed, or there is another non-completed global transaction that precedes it. After operation $r_{k+1}(a_{k-1})$ is submitted, Protocol GS discovers a cycle in the replication graph and aborts $T_{k+1}$. After that all transactions in the graph will be removed at the same time. □

**Theorem 2** *Protocol GS guarantees global serializability.*

**Proof Sketch:** Let $S$ be a schedule generated by protocol GS that is not globally serializable. Then the serialization graph of $S$ contains a cycle which we denote $T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow T_0$. This results (by the technique used in the proof of Theorem 1) to a cycle in any replication graph for $S$ of the form $G_0 - VS_0 - G_1 - \ldots G_{m-1} - VS_{m-1} - G_0$, where $\{G_0, G_1, \ldots, G_{m-1}\}$ is the set of global transactions in $\{T_0, T_1, \ldots, T_{n-1}\}$. This cycle must have gone undetected by protocol GS due to one or more instances of

- removal of a global transaction in the cycle, resulting in the removal of edges and applications of the split rule
- removal of a local transaction in the cycle, resulting in an application of the split rule

prior to the actual creation of the cycle.

Let us consider the state of the system immediately before the first such transaction removal. Let $T_i$ denote that first transaction. $T_i$ must have been in the completed state, and thus not preceded (directly or indirectly) by an active transaction in any local serialization order. Since, in the serialization graph, $T_{i-1 \pmod n} \rightarrow T_i$, $T_{i-1 \pmod n}$ must have executed at some site in common with $T_i$. Therefore $T_{i-1 \pmod n}$ must have committed. If $T_{i-1 \pmod n}$ has committed, then $T_{i-2 \pmod n}$ must have started so that it could precede directly $T_{i-1 \pmod n}$. But if $T_{i-2 \pmod n}$ started and cannot be active (due to $T_i$ being completed), it, too, must have committed. By a simple induction, all of $\{T_0, T_1, \ldots, T_{n-1}\}$ have committed. But then the cycle $G_0 - VS_0 - G_1 - \ldots G_{m-1} - VS_{m-1} - G_0$ must have existed prior to any transaction removals by protocol GS, contradicting Theorem 1. □

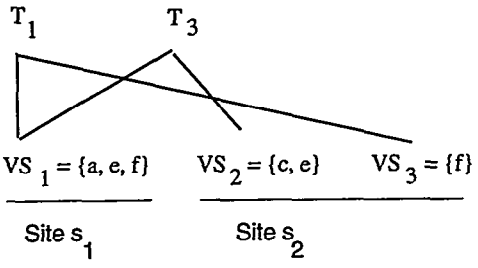


Figure 3: Replication Graph for Example 5.

## 5 Protocol GHOS

In this section, we compare protocol GS with the protocol given in [GHOS96] (we term it protocol GHOS) that uses the lazy-master replication approach (as does protocol GS). In protocol GHOS, each transaction must request a read lock from the primary site of each data item that it reads. Transactions must submit update operations to the primary site of the data item being updated. Thus, read and update operations conflict at the primary site of the data item. Until an update is completed for all replicas of data item $a$, no other transaction can read $a$. Write operations on secondary copies are synchronized using the Thomas Write Rule (TWR) [BHG87]. Thus, in the terminology of [BHG87] the concurrency control mechanism uses the Thomas write rule to synchronize *ww* conflicts and a form of two-phase locking to synchronize *rw* and *wr* conflicts.

The specific form of two-phase locking, which we term *two-phase locking with respect to reads*, or 2PL-R, is as follows. A transaction must hold a shared lock on any data item it reads. A transaction may not request a shared lock if it has already released a *shared* lock. A transaction must obtain an exclusive lock on a data item prior to writing it. An exclusive lock obtained on a data item must be retained until all writes to all replicas are completed, at which time the lock is released. Note the two-phase requirement applies only to shared locks. There is no two-phase requirement pertaining to exclusive locks.

In order to guarantee global serializability, protocol

GHOS must use the strict 2PL-R protocol. That is, shared locks must be held until the end of the transaction. Otherwise, the global serializability may be violated as the following example demonstrates:

**Example 8:** Consider a database consisting of two sites: $s_1$ and $s_2$. Site $s_1$ contains the primary copy of data items $a$ and $d$, and a secondary copy of data item $b$. Site $s_2$ contains the primary copy of data items $b$ and $c$, and a secondary copy of data item $d$. Let $T_1$, $T_2$, $T_3$, and $T_4$ be transactions defined as follows:

$T_1$: $r_1(a)$, $w_1(d)$ $\quad$ $T_3$: $r_3(c)$, $w_3(b)$
$T_2$: $r_2(b)$, $w_2(a)$ $\quad$ $T_4$: $r_4(d)$, $w_4(c)$

Transactions $T_1$ and $T_2$ originate at site $s_1$ and transactions $T_3$ and $T_4$ originate at site $s_2$. Observe that transactions $T_2$ and $T_4$ are local update transactions and transactions $T_1$ and $T_3$ are global update transactions.

Suppose that transaction operations were submitted and executed in the following global order:

$$r_3^2(c),\ r_4^2(d),\ w_4^2(c),\ c_4^2,\ r_1^1(a),\ r_2^1(b),\ w_2^1(a),$$
$$c_2^1,\ w_3^2(b),\ c_3^2,\ w_3^1(b),\ c_3^1,\ w_1^1(d),\ c_1^1,\ w_1^2(d),\ c_1^2$$

The resulting schedule is not globally serializable. However, the above schedule can be generated by non-strict 2PL-R. Including the strictness requirement suffices to rule out the above schedule.

It is interesting to note that the above global schedule results in local schedules that are *locally* feasible under standard two-phase locking. We show these schedules below:

$s_1$: $r_1(a)$, $r_2(b)$, $w_2(a)$, $c_2$, $w_3(b)$, $c_3$, $w_1(d)$, $c_1$
$s_2$: $r_3(c)$, $r_4(d)$, $w_4(c)$, $c_4$, $w_3(b)$, $c_3$, $w_1(d)$, $c_1$

□

**Theorem 3** *Protocol GHOS guarantees global serializability*

**Proof Sketch:** Assume there exists a nonserializable schedule generated by protocol GHOS. Then, there exists a cycle in the serialization graph of the form:

$$T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow T_0$$

Consider $T_0$ and $T_1$. Since $T_0 \rightarrow T_1$, $T_0$ and $T_1$ must conflict on some data item, $d$. We consider the three possible types of conflicts:

- *rw*: If $T_0$ reads $d$ before $T_1$ writes it, then $T_1$ cannot have accessed $d$ until $T_0$ releases its shared lock on $d$ when it commits. Therefore, $T_0$ commits before $T_1$ commits anywhere.
- *wr*: If $T_1$ reads the value of $d$ written by $T_0$, then it must wait until $T_0$ releases its exclusive lock on $d$. This cannot happen until the write has completed everywhere. Because of lazy propagation, this cannot occur until $T_0$ commits at its origination site. Therefore, $T_0$ commits at its origination site before $T_1$ does.
- *ww*: Write-write conflicts are handled in protocol GHOS by the Thomas Write Rule (TWR). The timestamps assigned by the TWR are based on commit time at the transaction's origination site. If $T_0 \rightarrow T_1$, then $T_0$ must have done its write before $T_1$ (logically, at least, even though late writes are ignored under TWR). Thus, $T_0$ must have a lower timestamp than $T_1$ and $T_0$ must have committed at its origination site before $T_1$.

We thus conclude that $T_0$ must have committed at its origination site before $T_1$. A simple induction generates the contradiction that completes the proof. □

## 6 Deadlocks

The deadlock phenomenon in our protocol differs from traditional deadlock. Waits in protocol GS are induced by an operation that would cause a cycle in the replication graph. Such waiting transactions are not waiting for a specific transaction to "go away," but rather they are waiting for *any* transaction in the cycle to be removed or any virtual site to be split in a way that breaks the cycle. This motivates a more general definition of deadlock that applies both to protocol GS and to the standard notion of deadlock as it exists in lock-based protocols in the local database systems.[3]

### 6.1 Definition of Deadlock

A set $D$ of transactions is said to be in *deadlock* if every transaction in $D$ has submitted an operation that either

1. waits for another member of $D$ within the concurrency control of some local database system,
2. waits under protocol GS due to **RGtest** generating a cycle involving only transactions in $D$ (and their associated virtual sites).

For simplicity in our discussion, we shall always assume that a deadlock set is minimal. The following example illustrates a deadlock caused by protocol GS:

**Example 9:** Consider a database located at three sites. Site $s_1$ contains the primary copy of data items $a$ and $c$, and a secondary copy of data items $d$ and $e$. Site $s_2$ contains the primary copy of data items $b$ and $d$ and a secondary copy of data item $a$. Site $s_3$ contains the primary copy of data item $e$, and a secondary copy of data items $b$ and $c$.

Let $T_1$, $T_2$, and $T_3$ be transactions originating at sites $s_1$, $s_2$, and $s_3$, respectively and defined as follows:

[3] We assume that all local waits arise from data-item conflicts. That is, if $T_1$ waits for $T_2$, then they conflict on some data item and $T_2$ accessed that data item first.

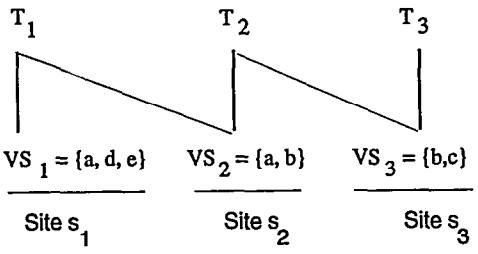


Figure 4: Replication Graph for Example 9

$T_1$: $r_1(d)$, $r_1(e)$, $w_1(a)$, $w_1(c)$
$T_2$: $r_2(a)$, $w_2(b)$, $w_2(d)$
$T_3$: $r_3(b)$, $r_3(c)$, $w_3(e)$

Suppose that, so far, transaction operations $r_1(d)$, $r_1(e)$, $w_1(a)$, $r_2(a)$, $w_2(b)$, $r_3(b)$ and $r_3(c)$ were submitted and executed at sites of their origination. This execution generates the local schedules shown below:

$s_1$: $r_1(d)$, $r_1(e)$, $w_1(a)$
$s_2$: $r_2(a)$, $w_2(b)$
$s_3$: $r_3(b)$, $r_3(c)$

The replication graph at the point is as shown in Figure 4. When $T_1$ submits $w_1(c)$ at site $s_1$, protocol GS will make $T_1$ wait, since otherwise the cycle $T_1 - vs_3 - T_2 - vs_2 - T_1$ would occur in the replication graph. When $T_2$ submits $w_2(d)$, it will have to wait too, since otherwise the cycle $T_1 - vs_1 - T_2 - vs_2 - T_1$ would occur. Finally, when $T_3$ submits $w_3(e)$, it also will have to wait since otherwise, the cycle $T_1 - vs_1 - T_3 - vs_3 - T_2 - vs_2 - T_1$ would occur. Consequently, none of these transactions can proceed, and a deadlock occurs. □

Due to our model's restrictions on the types of transactions that may execute, any distributed deadlock – that is, a deadlock involving more than one site – must involve at least one global transaction. As a consequence of our use of lazy update propagation, a global transaction executes at only its origination site until it has committed there. Subsequently, it runs independent subtransactions at each local site at which it must propagate updates. Thus, no wait involving a global transaction spans more than one local database system unless the wait is due to protocol GS. Thus, our definition of deadlock takes into account all possible waits, and, therefore, all possible deadlocks. Another fortunate consequence of the above argument is that no distributed deadlock may consist entirely of committed transactions.

We shall use the term local deadlock to refer to non-distributed deadlock.

### 6.2 Managing Deadlocks

Deadlocks involving only waits within one local database system are managed by that DBMS. We shall not concern ourselves with the specific manner in which these deadlocks are managed.

Deadlocks that involve waits generated by protocol GS are particularly difficult to detect. In a standard wait-for graph, a straightforward cycle detection algorithm can be executed. However, waits generated by part 2 of our definition of deadlock are waits for *any* member of the cycle found by **RGtest**, not all of them and not any specific one. To detect such deadlocks algorithmically, we thus need to create an **and/or** graph of waits. Deadlock detection in such graphs has been studied previously [CMH83, KKNR83]. In practice, we would avoid this complexity by implementing a timeout-based scheme to abort transactions that have been waiting "too long," and are therefore likely to be in deadlock. Such an approach to deadlock management is acceptable only if those deadlocks that are not exclusive to local database systems occur very infrequently. As we shall see below, this is indeed the case – global deadlock under our protocol is much less likely asymptotically than deadlock within a local database system.

As we have noted earlier, sub-transactions that update secondary copies of data may be aborted and restarted as needed. Thus, local deadlocks involving sub-transactions of committed global transactions present no difficulties.

### 6.3 Probability of Deadlock in Protocol GS

We begin our consideration of deadlock probability by showing that deadlock sets must have cardinality 3 or greater unless the deadlock is local.

**Theorem 4** *Assume that in any local database system, waits result only from data-item conflicts (as in locking). Let $T_1, T_2, \ldots, T_t$ be a set of global transactions that are involved into a distributed deadlock. Then $t > 2$.*

**Proof Sketch:** Consider a deadlock set $D$. If $D$ contains only one global transaction, then the deadlock must be contained within one local database system and not involve protocol GS. Such a deadlock is not a distributed deadlock.

Next consider the case that $D$ contains exactly two global transactions: $T_1$ and $T_2$.

First consider the case where $T_1$ and $T_2$ originate at the same site. If $T_1$ waits for $T_2$ at site $s_k$, and the conflict includes a replicated data item, then $T_1$ and $T_2$ are in a cycle in the replication graph. Since such cycles are forbidden by protocol GS, any deadlock must be local.

Next, assume $T_1$ and $T_2$ originate at distinct sites, $s_1$ and $s_2$, respectively. Then, they cannot update any data items in common (since only primary copies can be updated directly). If they conflict directly or indirectly at $s_1$, their virtual sites must be the same ($VS_1 = VS_{1,1} = VS_{2,1}$) and must include a replicated data item. Thus, there must be a path $T_1 - VS_1 - T_2$ in the replication graph. A similar situation at $s_2$ would cause a cycle in the graph and be forbidden. Thus any deadlock must either be local or include some virtual

site at a third physical site $s_k$. But then any conflict at $s_k$ involving $T_1$ and $T_2$ must include a replicated data item (since neither transaction originated at $s_k$), and such conflicts create a path $T_1 - VS_2 - T_2$ in the replication graph (where $(VS_2 = VS_{1,k} = VS_{2,k})$. Therefore, any distributed deadlock must be preceded by a cycle in the replication graph, and protocol GS disallows this. □

We now consider the probability of deadlocks that involve protocol GS. In order to simplify our analysis, we make several assumptions:

- There are $n$ transactions that are not yet completed, and all these transactions are global.
- Each transaction accesses $r$ data items.
- There are $m$ data items, all of which are fully replicated at each physical site.
- Data accesses are uniformly distributed, and all accesses are writes.
- Each transaction is half-executed and, thus, has accessed $r/2$ data items.

The uniformity assumptions ignore the possibility of hot spots. One may view the $m$ data items as being the hot spots and, likewise, the $r$ accesses by a transaction as being its accesses to hot spots. Overall, the above assumptions will cause us to overstate the probability of deadlock since, in practice, there would be read operations, not all data would be replicated, and data that is replicated would not all be fully replicated. Our assumptions, though pessimistic, correspond to those of [GHOS96, GHKO81].

Because of our assumption of update-only transactions, there is a path in the replication graph between a pair of transactions if and only if they conflict directly or indirectly on data accesses. The probability that an operation submitted by $T_i$ conflicts with some already-submitted operation of $T_j$ is $\frac{r}{2m}$. The probability that any remaining unsubmitted operation of $T_i$ conflicts with some already-submitted operation of $T_j$ is slightly less that $\frac{r^2}{4m}$ (and equal if we ignore the chance that two transactions may have more than one operation in conflict).

Now consider a chain $T_0 - T_1 - \ldots - T_{t-1}$ such that $T_i$ conflicts with $T_{i+1}$ for $i = 0, 1, \ldots, t-2$. The probability of such a chain is

$$\left(\frac{r^2}{4m}\right)^{t-1}$$

Now consider a set $D = \{T_0, T_1, \ldots, T_{t-1}\}$ and consider the probability that this set is a deadlock set. Each transaction in $D$ must have submitted an operation that would cause a cycle in the replication graph. Thus, there must be an ordering of $D$ such that for $i = 0, 1, \ldots, t-2$, $T_i$ and $T_{i+1}$ share a virtual site and, therefore, conflict. Without loss of generality, we assume that the transactions in $D$ are so numbered.

The probability that an operation submitted by a transaction in $D$ conflicts with another transaction in $D$ is bounded above by $\frac{(t-2)r}{2m}$ for $T_2, T_3, \ldots, T_{t-2}$, and by $\frac{(t-1)r}{2m}$ for $T_1$ and $T_{t-1}$. The $(t-2)$ factor arises from Theorem 4, which implies that in a conflict chain $T_0 - T_1 - \ldots - T_{t-1}$, if $T_i$ has submitted an operation on replicated data conflicting with $T_{i+1}$, then $T_{i+1}$ cannot have caused a deadlock by submitting an operation conflicting with $T_i$.

Thus, the probability that $D$ is a deadlock set is the probability that there is a chain of conflicts $T_0 - T_1 - \ldots - T_{t-1}$ times the probability that each $T_i$ submits an operation conflicting with some other transaction $T_j$. This probability is bounded above by

$$\left(\frac{r^2}{4m}\right)^{t-1}\left(\frac{(t-1)r}{2m}\right)^{2}\left(\frac{(t-2)r}{2m}\right)^{t-2}$$

Observe that, as expected from Theorem 4, this probability is zero for $t = 1$ and $t = 2$.

For simplicity, we weaken our upper bound as follows:

$$\left(\frac{r^2}{4m}\right)^{t-1}\left(\frac{tr}{2m}\right)^{2}\left(\frac{tr}{2m}\right)^{t-2}$$

Combining terms, we get:

$$\frac{r^{3t-2}t^t}{2^{3t-2}m^{2t-1}}$$

To obtain the overall probability of deadlock, we need to consider not only one set $D$ of $t$ transactions, but rather, all ways that such a set of $t$ transactions can be chosen. Since there are $n$ transactions, for each value of $t$ there are $\binom{n}{t}$ sets to consider. This gives

$$\sum_{t=3}^{n}\binom{n}{t}\frac{r^{3t-2}t^t}{2^{3t-2}m^{2t-1}}$$

The summations starts at 3 because we already know that the probability is zero for $t = 1$ and $t = 2$. Assuming $nr << m$ (a reasonable assumption)[4] the $t = 3$ term dominates in the sum. We thus conclude that the probability that the system is in distributed deadlock is

$$PD_{gs} = O\left(\frac{n^3r^7}{m^5}\right)$$

The probability that a given transaction deadlocks is $PD/n$.

[4] Without this assumption the overall contention rate is so high that one would impose admission control on transactions, and, as a result ensure that $nr << m$.

### 6.4 Comparison With Protocol GHOS

To put this result in perspective, we compare it to that of [GHOS96], which relies on global strict locking (2PL-R). In [GHOS96] the probability that the system is in deadlock is (using our notation)[5]:

$$PD_{ghos} = O\left(\frac{n^2r^4}{m^2}\right)$$

The ratio of the probability of distributed deadlocks in our protocols to that of [GHOS96] is:

$$\frac{PD_{gs}}{PD_{ghos}} = \frac{n^3r^7/m^5}{n^2r^4/m^2} = \frac{nr^3}{m^3}$$

This difference is significant since $nr << m$.

Unlike our protocol, [GHOS96] relies on global two-phase locking, and thus, the asymptotic probability of deadlock is the same as in any lock-based system (e.g. [GHKO81]). In our protocol, the probability of purely local deadlocks is $O\left(\frac{n^2r^4}{m^2}\right)$ as in [GHOS96], but the probability of distributed deadlock is much less than in [GHOS96]. The source of this difference is our use of the replication graph and virtual sites to eliminate distributed deadlock cycles of length 2, the likeliest kind.

We show below that if protocol GS deadlocks then so does protocol GHOS.

**Theorem 5** *If both protocols GHOS and GS have executed a prefix of a schedule for transaction set $T = \{T_1, T_2, \ldots, T_k\}$ and all members of $T$ are active, then if protocol GS deadlocks at this point, then so does protocol GHOS.*

**Proof Sketch:** Consider an execution of transactions in the two protocols. Assume that so far the execution order is the same for both algorithms. $T_1, T_2, \ldots, T_k$ active transctions that are in a deadlock as a result of protocol GS. This means that the operation submitted by each member of $T$ creates a cycle in the replication graph. A cycle in the replication graph can be created for one of the two reasons:

1. a new edge is introduced that creates a cycle;
2. two virtual sites are merged as a result of the union rule

In the first case, the introduced edge indicates that there is a conflict between active transactions at some virtual site. Since transactions are active, the lazy updates at secondary sites have not yet begun and thus exclusive locks taken by protocol GHOS could not have been released yet. Therefore, protocol GHOS causes a wait in this case as well. In the second case, consider the transaction $T_i$ waiting due to a virtual-site merger that lead to a cycle in the tentative replication graph of **RGtest**. There must be a conflict (directly or indirectly) between $T_i$ and the transaction with whose virtual site a merger is being attempted. Thus, under protocol GHOS, $T_i$ would wait as well. □

It is easy to design an example of a schedule where protocol GHOS would deadlock while protocol GS would not.

[5] Unlike [GHOS96], we include all transactions in $n$; and thus, we do not have a separate variable for the number of sites.

## 7 Fault Tolerance

Our protocol is robust in the presence of site failures, though it cannot handle network partitions. If a site fails, we assume that the local database system recovers correctly. Thus, no committed local transactions are lost. Any sub-transactions that were performing updates to secondary copies and that were active during a failure can be resubmitted.

Since a failed site cannot do any work before it recovers, it cannot take any action to change the replication graph while it is down. Thus, other sites may proceed. Of course, global transactions that cannot complete due to a site failure must remain in the replication graph, thus causing blocking.

In the event of a network partition, each partition may modify the replication graph, resulting in inconsistency.

Finally, we note that our reliance on local DBMSs for recovery simplifies the problem of recovery at the global level. A site needs to maintain an accurate view of the replication graph only as it pertains to its virtual sites and to transactions originating at that site in order to enable recovery. Suppose that a site $s_1$ crashes. Upon recovery, any transaction that was active at $s_1$ but did not originate at $s_1$ must be completed by accessing the primary copy of the data the failed transaction was updating.

## 8 Performance Issues

The key determinant of the practicality of our protocol is the overhead of maintaining the replication graph. If we assume centralized graph maintenance and compare this with centralized global locking, we can show that graph maintenance requires fewer messages since

1. only global writes generate new edges, and
2. updates to $VS_{ij}$ can be generated only by site $s_j$, so a site need perform graph maintenance globally only if it determines locally that virtual sites must be merged. Furthermore, by propagating only paths between transactions globally, local sites can avoid the need to distribute the actual set of data items in each virtual site.

The split rule can be applied lazily to conserve network resources.

Clearly, centralized graph maintenance may present problems of performance and recoverability. To enhance performance, we may use an optimistic approach in which updates to the replication graph are distributed to the physical sites on a best-effort basis. Prior to transaction commit, a validation test is run and, if the transaction is in a cycle, it is aborted.

Performance studies to test tradeoffs between optimism and pessimism are underway. We are also studying the performance of a 2-version scheme which ensures that read-only transactions are never delayed.

## 9 Conclusion

We have presented a protocol for managing replicated data that offers lower message overhead and asymptotically fewer deadlocks than previous results. Our protocol ensures serializability and is robust in the presence of site failures. Two concepts are key to our protocol. One is the use of virtual sites instead of physical sites and the dynamic management of virtual sites to keep them small. The second is the replication graph, which, in effect, reduces the problem of replica management over a large set of data items to the problem of managing a global graph whose size is on the order of the number of global transactions executing at a given time.

## Acknowledgements

The authors wish to thank Rick Stellwagen of NCR for helping us understand the role of replication in data warehouses, and Phil Gibbons, Yossi Matias, Rajeev Rastogi, and Avi Silberschatz of Bell Labs for helpful comments during this work.

## References

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2), 1992.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[CMH83] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[CRR96] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proceedings of the Twelveth International Conference on Data Engineering, New Orleans, Louisiana*, 1996.

[Ell77] C. A. Ellis. Consistency and correctness of duplicate database systems. *Operating Systems Review*, 11(5), November 1977.

[GHKO81] J. Gray, P. Homan, H. F. Korth, and R. Obermarck. A strawman analysis of the probability of wait and deadlock. Technical Report RJ2131, IBM San Jose Research Laboratory, 1981.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of ACM-SIGMOD 1996 International Conference on Management of Data, Montreal, Quebec*, pages 173–182, 1996.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann, San Mateo, CA, 1993.

[GS78] E. Gelenbe and K. Sevcik. Analysis of update synchronization for multiple copy data-bases. In *Proceedings of the Third Berkeley Workshop on Distributed Databases and Computer Networks*, pages 69–90, August 1978.

[HHB96] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. *Replication Techniques in Distributed Systems.* Kluwer Academic Publishers, 1996.

[Hol81] E. Holler. Multiple copy update. In *Lecture Notes in Computer Science, Distributed Systems — Architecture and Implementation: An Advanced Course.* Springer-Verlag, Berlin, 1981.

[KI96] H. F. Korth and T. I. Imielinski. Introduction to mobile computing. In *Mobile Computing*, pages 1–39. Kluwer Academic Publishers, 1996.

[KKNR83] H. F. Korth, R. Krishnamurthy, A. Nigam, and J. T. Robinson. A framework for understanding distributed (deadlock detection) algorithms. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta*, pages 192–201, 1983.

[KLS90] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane*, pages 95–106, August 1990.

[Min79] T. Minoura. A new concurrency control algorithm for distributed database systems. In *Proceedings of the Fourth Berkeley Workshop on Distributed Databases and Computer Networks*, pages 221–234, August 1979.

[PL91] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data, Denver, Colorado*, pages 377–386, May 1991.

[SAB+96] J. Sidell, P. M. Aoki, S. Barr, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proceedings of the Twelveth International Conference on Data Engineering, New Orleans, Louisiana*, 1996.

[Tho78] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *CompCon78*, 1978.