Availability in Partitioned Replicated Databases

(Extended Abstract)

Amr El Abbadi[*]
Sam Toueg[**]
Department of Computer Science
Cornell University
Ithaca, New York 14853

ABSTRACT

In a replicated database, a data item may be distributed at several sites. A replica control protocol is necessary to ensure that data items with several copies behave as if they consist of a single copy, as far as users can tell. We describe a replica control protocol that allows the accessing of data in spite of site failures and network partitioning. The new solution is more efficient and more flexible than previous ones, while providing at least the same degree of data availability.

1. Introduction

The availability of data in distributed databases can be increased by replication. If the data is replicated on several sites, it may still be available following site failures. However, implementing an object with several copies residing on different sites may introduce inconsistencies between copies of the same object. To be consistent, a system should behave as if each object has only one copy in so far as the user can tell, i.e., it should be *one-copy equivalent*. A *replica control protocol* is one that ensures that the database is one-copy equivalent.

A database system should also ensure *serializability*, i.e., if operations of transactions are interleaved, the system behaves as if all the transactions were executed in some serial order. A *concurrency control protocol* is one that ensures serializability. Several such concurrency control protocols are known [BGa]. A database system is correct if it is *one-copy serializable*, i.e., it ensures serializability and is one-copy equivalent.

We consider database systems that are prone to both site and link failures. Sites may fail by crashing or by failing to send or receive messages. Links may fail by crashing, delaying or failing to deliver messages. Several replica control protocols have been proposed that tolerate different types of failures.

An example of a simple replica control protocol is one that requires a *write* operation to write all copies of an object, and a *read* operation to read any one copy. However, such a protocol does not allow write operations to be executed on objects with copies on failed sites. The *Available Copies* protocol [BGc,GSCDFR]

[*] This author was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract DA903-85-C-0124, and by the National Science Foundation under Grant DCR-8412582. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

[**] This author was supported by the National Science Foundation under grant MCS 83-03135.

requires a write operation to write only copies on operational sites. Hence, with this protocol write operations can be executed even when sites fail. However, combinations of site and link failures may partition a database. Sites in a *partition* can communicate with each other, but not with sites in other partitions. If partitioning occurs, the Available Copies protocol may cause inconsistencies in the database.

In this paper, we present a replica control protocol that allows the accessing of data even when the database is partitioned. It can be combined with any available concurrency control protocol to ensure the correctness of a database. Our approach draws on recent work by El Abbadi, Skeen and Cristian in [ASC]. In contrast to the method in [ASC], we do not require a separate protocol to coordinate the views different sites have of the communication network. This results in a simpler and more efficient protocol.

In most previous replica control protocols that tolerate failures, the possibility of executing an operation on an object determines the costs of the operations of that object. For example, *quorum based protocols* [G,ES,Her] require that in order to allow the execution of write operations in the presence of failures, the cost of read operations must be increased (by accessing more copies of the object read). This may preclude their use in applications whose feasibility critically depends on efficient read operations. With our protocol, it is never necessary for a read operation to access more than one copy, even if the database partitions. The cost of a read operation is independent of the level of availability associated with read or write operations. In general, our protocol provides the database designer with a large degree of flexibility in deciding when operations may be executed on objects, as well as in deciding the costs of these operations.

In the next section, we describe the formal database model and our correctness criteria. In Section 3, we propose a replica control protocol, and we outline its proof of correctness in Section 4. A comparison with previous results and a discussion concludes the paper.

2. The Model

We consider a set of *sites* connected by bidirectional *links*. Associated with each site is a unique *site−identifier*. Site identifiers form a total order. A *distributed database* consists of a set of *objects* that may reside at different sites. A *transaction* t_i is a partially ordered set $(S_i, <_i)$, where S_i is the set of all operations executed by t_i, and $<_i$ reflects the order in which they should be executed. Read and write operations executed by transaction t_i on object x are denoted $r_i[x]$ and $w_i[x]$. The site at which a transaction is initiated is called its *initiator*. The execution of a transaction is *atomic*, i.e., before a transaction terminates it either *commits* or *aborts* all changes it made to the database.

In this section most of the definitions and correctness criteria are drawn from the model of Bernstein and Goodman in [BGb]. The execution of operations in the system is modeled by logs. A *log* L over a set of transactions $T = \{t_1, t_2, \ldots, t_n\}$ is a partial order $(S, <_L)$ where S is the set of all operations executed by transactions in T, and $<_L$ reflects the order in which the operations were executed. We assume that an initial transaction t_{init} is executed to write all objects in the database before any other transaction.

2.1. Correctness Criteria for Non-replicated Databases

We first consider non-replicated databases where each object x resides on a single site. Let L be a log over a set of transactions T. Transaction t_j *reads−x−from* transaction t_i in L if:

1- $w_i[x]$ and $r_j[x]$ are in L.

2- $w_i[x] <_L r_j[x]$.

3- There is no $w_k[x]$ such that $w_i[x] <_L w_k[x] <_L r_j[x]$.

Two logs L_1 and L_2 are *equivalent* if for all t_i, t_j and x, t_j *reads−x−from* t_i in L_1 if and only if t_j *reads−x−from* t_i in L_2. A *serial* log is a totally ordered log such that for every pair of transactions t_i and t_j either all operations executed by t_i precede all operations executed by t_j or vice versa. A log is *serializable* if it is equivalent to a serial log over the same set of transactions. Since a serial execution of transactions preserves the consistency of the database, we consider serializability as our correctness criterion. The problem of determining whether an arbitrary log is serializable is NP-complete [P], thus it is unlikely that there exists efficient concurrency control protocols that allow all serializable logs. We therefore consider a class of concurrency control protocols that allows a subset of the class of serializable logs, the *CP-serializable* logs as defined below.

Two operations *conflict* if they both operate on the same object, and one of them is a write. A log L is *CP−serializable* [Had,P] if there exists some serial log L_s over the same set of transactions, such that if op_1 and op_2 conflict and $op_1 <_L op_2$ then $op_1 <_{L_s} op_2$. Note that L is equivalent to L_s, and since L_s is serial, L is serializable.

A *serialization graph* $SG(L)$ of a log L is a directed graph whose nodes are transactions and whose edges are: $\{t_i \rightarrow t_j | \exists\ op_i$ executed by t_i and op_j executed by t_j such that op_i conflicts with op_j and $op_i <_L op_j\ \}$. L is CP-serializable if and only if $SG(L)$ is acyclic [EGTL,Had].

2.2. Correctness Criteria for Replicated Databases

We now consider replicated databases where each object is implemented by a set of copies that reside on different sites. The copy of object x that resides on site p is denoted by x_p. Each copy x_p has a *version number* which is initialized to 1 by t_{init}. The set of all copies of object x is denoted *copies*$[x]$, and the set of sites on which x resides is denoted *sites*$[x]$. The number of copies of x in the system is $n[x]$ ($n[x]=|copies[x]|$). In a *non-replicated database* all objects are implemented by a single copy.

An operation issued by a transaction on an object is called a *logical operation*. Such an operation is executed by a set of *physical operations* on the copies of the object. A logical write $w_i[x]$ is executed by

1- selecting a set of copies of x,

2- determining *vnmax*, the maximum version number of the selected copies,

3- writing all the selected copies and updating their version numbers to $vnmax + 1$.

A logical read $r_i[x]$ is executed by

1- selecting a set of copies of x,

2- *accessing* all the selected copies to find the one with the highest version number,

3- reading this copy.

The read, access and write operations that are executed on a copy x_p by transaction t_i are denoted $r_i[x_p]$, $a_i[x_p]$ and $w_i[x_p]$.

A *replicated database log* L contains physical operations on the copies of objects. For each logical write $w_i[x]$, there is a set $\{x_p, , \ldots, x_q\}$, the set of copies written, such that L contains

$w_i[x_p], \ldots, w_i[x_q]$. For each logical read $r_i[x]$, there is a set $\{x_p, \ldots, x_j, \ldots, x_q\}$, the set of copies accessed, such that L contains $a_i[x_p] <_L r_i[x_j]$, ... , $a_i[x_j] <_L r_i[x_j]$, ... , $a_i[x_q] <_L r_i[x_j]$ (where x_j is the value read). Transaction t_j *reads−x−from* t_i if there is a copy x_p of object x such that:

1- $w_i[x_p]$ and $r_j[x_p]$ are in L.

2- $w_i[x_p] <_L r_j[x_p]$.

3- There is no $w_k[x_p]$ such that $w_i[x_p] <_L w_k[x_p] <_L r_j[x_p]$.

Two logs L_1 and L_2 are *equivalent* if for all t_i, t_j and x, t_j *reads−x−from* t_i in L_1 if and only if t_j *reads−x−from* t_i in L_2. A log is *one−copy serializable* [BGb] if it is equivalent to a serial log over the same set of transactions executed over a non-replicated database.

We extend the definition of conflict to both logical and physical operations. Two logical (physical) operations *logically* (*physically*) *conflict* if they both operate on the same object (copy) and one of them is a write.

The serialization graph $SG(L)$ of a replicated database log L is a directed graph whose nodes represent transactions, and whose edges are: $\{t_i \rightarrow t_j | \exists\ op_i$ executed by t_i and op_j executed by t_j such that op_i *physically* conflicts with op_j and $op_i <_L op_j\}$. The graph $SG(L)$ orders transactions that issue physically conflicting operations in L. However, two transactions may issue two logically conflicting operations that may not physically conflict, and hence are not ordered by $SG(L)$. To order transactions that issue logically conflicting operations, we extend $SG(L)$ into a *one-copy serialization graph*, $1-SG(L)$ by adding enough edges such that:

1- For each object x, $1-SG(L)$ embodies a total order over all transactions that write x. This total order is called a *write order* for x, and is denoted $\Rightarrow_x$.

2- For each object x and transactions t_i, t_j, t_k such that t_j *reads−x−from* t_i and $t_i \Rightarrow_x t_k$, $1-SG(L)$ contains a path from t_j to t_k. This path is called a *reads−before* path.

In [BGb], Bernstein and Goodman prove that a log L is one-copy serializable if L has an acyclic $1-SG(L)$ graph.

3. A Replica Control Protocol

Our replica control protocol assumes two types of transactions: *User transactions*, issued by the users of the database, and *update transactions*, issued by the protocol. We assume that all transactions follow a correct concurrency control protocol, e.g., two phase locking [EGLT], timestamp ordering [BSR], etc. Such a protocol ensures logs are CP-serializable. In this section we present a replica control protocol that ensures logs are one-copy serializable. To describe the execution of transactions, we first have to define the notions of a view, and of read and write accessibility of objects in a view [ASC].

3.1. Views, Accessibility and Quorums

Each site s maintains a set called its *view*, the set of sites s assumes it can communicate with. Associated with each view is a *view−id*, and two sites are said to have the *same* view, if their views have identical view-ids (we describe how view-ids are assigned later).

With each object x, we associate *read* and *write accessibility thresholds*, $A_r[x]$ and $A_w[x]$, respectively. An object x is *read* (*write*) *accessible* in a view only if $A_r[x]$ ($A_w[x]$) copies reside on sites in that view. The accessibility thresholds $A_r[x]$ and $A_w[x]$ must satisfy:

$$A_r[x] + A_w[x] > n[x] \qquad (1)$$

$$2A_w[x] > n[x]$$

These relations ensure that a set of copies of x of size $A_w[x]$ has at least one copy in common with any set of copies of x of size $A_r[x]$ or $A_w[x]$.

In each view v, every object x is assigned a *read* and *write quorum*, $q_r[x,v]$ and $q_w[x,v]$: these specify how many read and write physical operations are needed to read and write an object x in view v. Let $n[x,v]$ be the number of copies of x that reside on sites in view v (formally, $n[x,v] = |sites[x] \bigcap v|$). For each view v, the quorums of object x must satisfy the following relations:

$$q_r[x,v] + q_w[x,v] > n[x,v] \quad (3)$$

$$1 \leq q_r[x,v] \leq n[x,v] \quad (4)$$

$$A_w[x] \leq q_w[x,v] \leq n[x,v] \quad (5)$$

These relations ensure that, in a view v, a set of copies of x of size $q_w[x,v]$ has at least one copy in common with any set of copies of x of size $q_r[x,v]$ as well as with any set of size $q_w[x,v']$ for any v'. Furthermore, a set of size $q_w[x,v]$ has at least one copy in common with any set of size $A_r[x]$.

3.2. User Transactions

A user transaction *executes in* view v if it is initialized at a site with view v. A transaction t_i that executes in view v can read (write) an object x only if x is read (write) accessible in view v. If x is read accessible in view v, t_i executes the logical operation $r_i[x]$ by physically accessing $q_r[x,v]$ copies of x in v. Similarly, if x is write accessible in view v, t_i executes the logical operation $w_i[x]$ by physically writing $q_w[x,v]$ copies of x in v. A user transaction executing in view v is aborted if any of the copies it accesses or writes resides on a site with a view different from v, or on a site with which s cannot communicate.

The use of accessibility thresholds in conjunction with quorums, and the fact that each view can independently define its own quorums gives the database designer an unusual degree of flexibility. This can be used to achieve the desired cost/availability tradeoff of read and write operations. There are several such tradeoffs. For example, one can increase the availability of an object for reading by decreasing the read accessibility threshold, $A_r[x]$—at the cost of increasing $A_w[x]$, i.e., decreasing the availability of write operations on the object. In some applications read operations outnumber write operations, and in this case it is advantageous to allow inexpensive read operations. Using quorums, this can be easily achieved with $q_r[x,v]=1$, and $q_w[x,v]=n[x,v]$.

3.3. Update Transactions

Views change during system execution. A site changes its view whenever it notices a discrepancy between its current view and the sites it can actually communicate with. A site s may change its view in two different ways. Site s may decide on the members of a view v based on its own information, in which case s is called the *initiator* of v. Site s may also decide to adopt a view v initiated by another site, in which case we say s *inherits* view v. Views are considered objects, therefore a transaction that executes an operation on a view must follow the concurrency control protocol.

The initiator of a view associates a *view−id* with the view. The view-id consists of two fields $<no, id>$ where *no* is an integer greater than any other *no* the initiator has encountered, and *id* is the unique site identifier

of the initiator. We say $<no_1, id_1>$ is *less than* $<no_2, id_2>$, if $no_1 < no_2$, or $no_1 = no_2$ and $id_1 < id_2$.

Whenever a site s changes its view to a new view v, either by initiating that view or by inheriting it, s must execute an *update transaction*. Site s executes an update transaction to update its local copies: for each object x, the local copy x_s is updated relative to the other copies of x that reside in the new view v.

Let s be a site whose current view has view-id *view_id*. To initiate a new view, s atomically executes the procedure *initiate_new_view*(*view_id*) illustrated in Figure 1 (where *view_id.no* refers to the first field in *view_id*). Site s first determines the new view, *new_view* and its associated view-id, *new_view_id*, and then executes a update transaction. If the update transaction is successful, s *installs* *new_view* as its view. If the update transaction is aborted, s initiates a new view with view-id higher than *new_view_id*.

The execution of the update transaction ensures that all local copies of objects are up to date with respect to all other copies in *new−view* (see Figure 2). Let $R_a[new\_view]$ be the set of objects that are read accessible in *new_view*. The update transaction updates all copies of objects in $R_a[new\_view]$ that reside on s. For each object $x \in R_a[new\_view]$, the update transaction first executes access operations on $A_r[x]$ copies of x that reside on sites in *new_view*. It then reads the accessed copy with the highest version number, and writes its value and its version number into x_s.

We associate with each access operation *new_view* and *new_view_id*. The access of copy x_p is aborted if s cannot communicate with p, or if p has a view-id higher than *new_view_id*. If any access operation aborts, the update transaction is aborted.

When a site p receives an access request from s it can take three possible actions depending on *new_view_id*, the view-id associated with the request, and *view_id*, the view-id associated with p's current view (see Figure 3 where the select statement is executed atomically). If *view_id* is less than *new_view_id* then p

---

initiate_new_view(*view_id*)

new_view := {p| s assumes it can communicate with p}
new_view_id := $<$*view_id.no* + 1, $s>$
execute update transaction(*new_view*, *new_view_id*)
if update transaction is not aborted /* install *new_view* */
→ *view* := *new_view*;
view_id := *new_view_id*
[] update transaction is aborted
→ *initiate_new_view*(*new_view_id*)
fi

Figure 1. *initiate_new_view* procedure executed at site s to initiate a new view.

---

update transaction(new_view, new_view_id)

For all *read accessible* objects x in *new_view* **do**

Select $A_r[x]$ copies of x in *new_view*;

For all selected copies x_p **execute** *access*(x_p, *new_view*, *new_view_id*);

if no access operation aborted

→read accessed copy of x with highest version number

write x_s with the value read

[] some access operation aborted

→**abort** update transaction

fi

od

Figure 2. *update transaction* executed at site s to update its local copies according to *new_view*.

access(x_p, *new_view*, *new_view_id*)

select from

view_id < *new_view_id* → **execute** $a[x_p]$ and return x_p to s
execute *inherit_new_view*(*new_view*, *new_view_id*)

[] *view_id* = *new_view_id* → **execute** $a_r[x_p]$ and return x_p to s

[] *view_id* > *new_view_id* → **abort** access operation

selectend

Figure 3. Response of site p to a request *access*(x_p, *new_view*, *new_view_id*) by site s.

executes the access operation, and immediately attempts to inherit *new_view*. If *view_id* is equal to *new_view_id* then p executes the access operation. If *view_id* is greater than *new_view_id* then p aborts the access operation.

To inherit a view *new_view* with view-id *new_view_id*, site s atomically executes the procedure *inherit_new_view*(*new_view*, *new_view_id*) illustrated in Figure 4. Site s simply executes an update transaction. If the update transaction is committed, s *installs* *new_view*. Otherwise, s initiates a new view with view-id higher than *new_view_id*. Note that when a site initiates or inherits a new view, the new view-id is always greater than any previous view-ids at that site.

4. Proof of Correctness

In this section we prove that the protocol presented in the previous section ensures one-copy serializability. Before proceeding to prove the correctness of the protocol, we extend the standard serialization theory to allow for operations executed by both user and update transac-

```
inherit_new_view(new_view, new_view_id)

execute update transaction(new_view, new_view_id)
if update transaction is not aborted      /* install new_view */
        → view := new_view;
          view_id := new_view_id
[] update transaction is aborted
        → initiate_new_view(new_view_id)
fi
```

Figure 4. *inherit_new_view* procedure executed at site s to inherit a new view *new_view*.

tions.

4.1. Extensions to the Standard Serializability Theory

The standard serializability theory presented in Section 2 assumed that only user transactions were executed in the system. In this section, we extend the theory to include both update and user transactions. We redefine *reads−x−from* relations, serialization graphs and one-copy serialization graphs.

We first extend the *reads−x−from* relation to include update transactions. Let L be a log over a set of transactions T. For any two (user or update) transactions t_i and t_j and object x, t_j *direct reads−x−from* t_i in L if there is a copy x_p such that:

1- $w_i[x_p]$ and $r_j[x_p]$ are in L.

2- $w_i[x_p] <_L r_j[x_p]$.

3- There is no $w_k[x_p]$ such that $w_i[x_p] <_L w_k[x_p] <_L r_j[x_p]$.

Let t_u be an update transaction, executed by site s, that updates the values of a copy x_s. If t_u *direct reads−x−from* t_i, then t_u reads the value of x written by t_i (and its associated version number) and writes it into x_s. Now, if t_j *direct reads−x−from* t_u then t_j reads the value of x written by t_u. Since this value was originally written by t_i, t_j indirectly reads the value written by t_i.

We formalize this concept by extending the definition of *reads−from*. Let $t_{u_1}, t_{u_2}, \cdots, t_{u_n}$ be a sequence of update transactions and t_i and t_j be two user transactions. We say t_j *indirect reads−x−from* t_i if t_{u_1} *direct reads−x−from* t_i, t_{u_2} *direct reads−x−from* t_{u_1}, . . . , and t_j *direct reads−x−from* t_{u_n}. We henceforth refer to both *direct* and *indirect reads−x−from* relations simply as *reads−x−from*.

The serialization graph $SG(L)$ for a log L is a directed graph whose nodes are all user and update transactions. $SG(L)$ has an edge between any two transactions that issue conflicting physical operations. We redefine $1\text{−}SG(L)$ to ensure that it has a path between any two transactions issuing logically conflicting operations. $1\text{−}SG(L)$ must have enough edges so that:

1- For each object x, $1\text{−}SG(L)$ embodies a total order $\Rightarrow_x$ on all user transactions that write x.

2- For any two user transactions t_i and t_j, such that t_j *reads−x−from* t_i (direct or

indirect), $1-SG(L)$ has a path from t_i to t_j. Such a path is called a *reads−from* path.

3- For any three user transactions t_i, t_j and t_k such that t_j *reads−x−from* t_i (direct or indirect) and $t_i \Rightarrow_x t_k$, $1-SG(L)$ has a path from t_j to t_k. Such a path is called a *reads−before* path.

The proof of Theorem 2 in [BGb] still hold with the extensions we have made: if the graph $1-SG(L)$ is acyclic, the log L is *one-copy serializable.*

4.2. The Proof

Given a log L of transactions executed using our replica control protocol, we first show how to construct a corresponding $1-SG(L)$ graph. Then we show that $1-SG(L)$ is acyclic, and hence L is one-copy serializable.

4.2.1. Construction of $1-SG(L)$

Let L be a log over a set of user and update transactions executed using our replica control protocol in conjunction with a protocol ensuring *CP−serializability*. Let $SG(L)$ be the corresponding serialization graph. Note that *CP−serializability* ensures the acyclicity of $SG(L)$. We now show how to extend $SG(L)$ into an $1-SG(L)$ by adding enough edges to satisfy the three requirements described in Section 4.1.

The next two lemmas prove that $SG(L)$ already satisfies the first two of these three requirements.

Lemma 1: For each object x, $SG(L)$ embodies a total order $\Rightarrow_x$ on all user transactions that write x.

Proof: Left to the full paper [AT]. □

Lemma 2: For any two user transactions t_i and t_j if t_j *reads−x−from* t_i then $SG(L)$ has a path from t_i to t_j.

Proof: Left to the full paper [AT]. □

Lemmas 1 and 2 show that $SG(L)$ already satisfies the first two requirements of an $1-SG(L)$. We now prove that $SG(L)$ partially satisfies the third requirement as well. We start with a technical lemma.

Lemma 3: The index of a user transaction t_i in $\Rightarrow_x$ is the version number that it assigns to each copy of x that it writes.

Proof: Left to the full paper [AT]. □

Lemma 4: Let t_i, t_j and t_k be three user transactions where t_j and t_k execute in the same view. If t_j *reads−x−from* t_i and $t_i \Rightarrow_x t_k$, then $SG(L)$ has an edge from t_j to t_k.

Proof: Left to the full paper [AT]. □

Lemmas 1,2 and 4 show that we can extend $SG(L)$ into $1-SG(L)$ by adding only the following edges:

> For any three user transactions t_i, t_j and t_k, where t_j and t_k execute in *different* views, if t_j *reads−x−from* t_i and $t_i \Rightarrow_x t_k$, then add an edge from t_j to t_k. These are *reads−before* edges.

So $SG(L)$ is extended into $1-SG(L)$ by adding only *reads−before* edges between transactions in different views.

4.2.2. Acyclicity of $1-SG(L)$

Since the log L is *CP−serializable*, $SG(L)$ is acyclic. In this section we show that its extension $1-SG(L)$ is also acyclic. To simplify the presentation, we define $v[t]$ as follows. If t is a user transaction, then $v[t]$ is the view t executed in. If t is an update transaction, then $v[t]$ is the view whose installation caused the execution of t. Let $v\_id[t]$ be the view-id of $v[t]$. We first prove that if $SG(L)$ has an edge from transaction t_i to transaction t_j then $v\_id[t_i] \leq v\_id[t_j]$. We then extend this result to edges of $1-SG(L)$.

Lemma 5: If $SG(L)$ has an edge from t_i to t_j then $v\_id[t_i] \leq v\_id[t_j]$.

Proof: Left to the full paper [AT]. □

We now extend the previous lemma to edges in $1-SG(L)$.

Lemma 6: If $1-SG(L)$ has an edge from t_i to t_j then $v\_id[t_i] \leq v\_id[t_j]$.

Proof: Left to the full paper [AT]. □

We can now show that $1-SG(L)$ is acyclic, and therefore L is one-copy serializable.

Theorem 1: $1-SG(L)$ is acyclic.

Proof: For contradiction, suppose $1-SG(L)$ has a cycle. By Lemma 6, it is clear that all transactions in this cycle must have the same view-id, and hence execute in the same view. Since $SG(L)$ is acyclic, the cycle in $1-SG(L)$ has at least one *reads−before* edge between two transactions executing in different views, a contradiction. Hence, $1-SG(L)$ is acyclic. □

5. Discussion

We presented a replica control protocol that increases the availability of databases in the presence of site failures and partitioning. Our protocol is an improvement over previous work in several ways. In [ASC], whenever the communication topology changes, a two-phase protocol is executed to ensure consistency of views. Then, an update operation is initiated that updates the different copies to ensure the consistency of data. The cost of the view management protocol is the main disadvantage of this method. We observe that while the update protocol is necessary to ensure consistency of data, the two-phase view management protocol is redundant. The approach taken in this paper eliminates the need for a separate view management protocol.

The protocol presented in this paper achieves the same degree of data availability as the protocols proposed in [ASC,G,ES,Her], while providing for a greater degree of flexibility. In [ASC], the quorums are fixed to $q_r[x,v] = 1$ and $q_w[x,v] = n[x,v]$ for all views v. The algorithms presented in [G,ES,Her] are also special cases of our protocol where quorums and thresholds are uniformly fixed to $q_r[x,v]=A_r[x]$, and $q_w[x,v]=A_w[x]$ for all views v. With our protocol each view may have different quorum assignments for the same object x, as long as they satisfy quorum relations (3), (4) and (5). This, in addition to the flexibility in choosing the accessibility thresholds $A_r[x]$ and $A_w[x]$ (according to relations (1) and (2)), provide the database designer with a large degree of freedom in deciding the cost of operations, as well as their level of availability.

Furthermore, our protocol reduces the number of physical operations executed. For example, consider an object x with $n[x]$ copies. Let $Q_r[x]$ and $Q_w[x]$ be the quorums associated with x in any of the protocols in [G,ES,Her]. These quorums must satisfy $Q_r[x] + Q_w[x] > n[x]$. To achieve the same level of availability as these protocols do, we set the thresholds and quorums of our protocol to $A_r[x]=q_r[x,v]=Q_r[x]$ for all v, and $A_w[x]=Q_w[x]$. With these threshold and quorum assignments, any read or write operation on x that is executable using quorum protocols is also executable with our protocol. Let P be a partition where all sites have view $v = P$. To write an object x in P, the protocols in [G,ES,Her] require writing $Q_w[x]>n[x] - Q_r[x]$ copies, compared to writing $q_w[x,v]>n[x,v]-q_r[x,v]$ copies with our protocol. Note that $n[x,v] \leq n[x]$ and $q_r[x,v] = Q_r[x]$.

With our protocol, it is never necessary for a read operation to physically access more than

one copy, not even if the database partitions. This is in contrast to the protocols in [G,ES,Her] where expensive read operations are required in order to execute write operations in the presence of failures. A write operation on an object x cannot require the physical writing of more than $n[x]-f$ copies, if it is supposed to execute despite the inaccessibility of f copies. For this case, the protocol in [G] requires that read operations always physically access at least $f+1$ copies. This is improved upon in [Es,Her] where these expensive read operations are necessary only when failures occur. With our protocol, it is never necessary to incur this high cost, even if partitioning occurs.

To illustrate the possible choices of thresholds and quorums with our protocol, consider an object x with $n[x]=8$. Let P be a partition where all sites have view $v=P$, and $n[x,v]=6$. By setting the accessability thresholds to $A_r[x]=4$ and $A_w[x]=5$, x is read and write accessible in P. The database designer has the following two possible choices for the read and write quorums of partition P:

Quorums	Choice I	Choice II
$q_{r[x,v]}$	1	2
$q_{w[x,v]}$	6	5

Both choices satisfy the quorum relations (3), (4) and (5). With choices I and II, a read operation on x physically accesses 1 or 2 copies, respectively.

With a quorum-based protocol [G,ES,Her], to allow the writing of x in P one can set either $Q_w[x]=6$, in which case a read operation has to access three copies, or $Q_w[x]=5$, in which case a read operation has to access four copies.

6. Acknowledgements

We would like to thank Ken Birman, Tommy Joseph, Gayatri Kapur, Richard Koo, Gil Neiger, Thomas Räuchle and T.K. Srikanth for their helpful comments.

7. References

[ASC] El Abbadi, A., Skeen, D. and Cristian, F., "An Efficient Fault-Tolerant Protocol for Replicated Data Management", *Proc. 4th ACM Symp. on Princ. of Database Systems*, Portland, Oregon, March 1985, 215-229.

[AT] El Abbadi, A., Toueg, S., "Availability in Partitioned Replicated Databases", Tech. Report, Cornell University (December 85).

[BGa] Bernstein, P. and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys 13*, 2, (June 1981) 185-222.

[BGb] Bernstein, P., and Goodman, N., "The Failure and Recovery Problem for Replicated Databases," *Proc. 2nd ACM Symp. on Princ. of Distributed Computing*, Montreal, Quebec, August 1983, 114-122.

[BGc] Bernstein, P., and Goodman, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases" *ACM Transactions on Database Systems 9*, 4 (December 1984), 596-615.

[BSR] Bernstein, P., Shipman, D., and Rothnie, Jr., J., "Concurrency Control in a System

for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems 5*, 1 (March 1980), 18-51.

[EGLT] Eswaran, K., Gray, J., Lorie, R., and Traiger, I., "The Notions of Consistency and Predicate Locks in a Database System," *Comm. of the ACM 19*, 11 (November 1976), 624-633.

[ES] Eager, D., and Sevcik, K., "Achieving Robustness in Distributed Database Systems," *Transactions on Database Systems 8*, 3 (September 83), 354-381.

[GSCDFR] Goodman, N., Skeen, D., Chan, A., Dayal, U., Fox, S., and Ries, D., "A Recovery Algorithm for a Distributed Database System," *Proc. 2nd ACM Symp. on Princ. of Database Systems*, Atlanta, Georgia, March 1983, 8-15.

[G] Gifford, D., "Weighted Voting for Replicated Data," *Proc. of the 7th Symposium on Operating Systems Principles*, Dec. 1979.

[Had] Hadzilacos, V., "Issues of Fault Tolerance in Concurrent Computations," Tech. Report 11-84, Harvard University, Center for Research in Computing Technology, Cambridge, Massachusetts (June 1984).

[Her] Herlihy, M., "Replication Methods for Abstract Data Types", MIT/LCS/TR-319, Massachusetts Institute of Technology, Cambridge, Massachusetts (May 1984).

[P] Papadimitriou, C.H., "Serializability of Concurrent Database Updates," *Journal of the ACM 24*, 4, (October 1979) 631-653.