Maintaining Availability in Partitioned Replicated Databases

AMR EL ABBADI and SAM TOUEG
Cornell University

In a replicated database, a data item may have copies residing on 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 new replica control protocol that allows the accessing of data in spite of site failures and network partitioning. This protocol provides the database designer with a large degree of flexibility in deciding the degree of data availability, as well as the cost of accessing data.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; H.2.4 [**Database Management**]: Systems—*concurrency*; *distributed systems*; *transaction processing*

General Terms: Algorithms, Management, Reliability

Additional Key Words and Phrases: Concurrency control, partitioning failures, replica control, replicated databases, serializability theory

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 after 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 be *one-copy equivalent*; that is, it should behave as if each object has only one copy in so far as the user can tell. A *replica control protocol* is one that ensures that the database is one-copy equivalent.

The work of Amr El Abbadi 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. The work of Sam Toueg was supported by the National Science Foundation under grants MCS 83-03135 and DCR-8601864.

Authors' addresses: A. El Abbadi, Department of Computer Science, University of California, Santa Barbara, CA 93106; S. Toueg, Department of Computer Science, Cornell University, Ithaca, NY 14853.

A database system should also ensure *serializability*; that is, 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 [1]. A database system is correct if it is *one-copy serializable*; that is, 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 [21]. 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 [3, 15] 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 [8, 23]. 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 work by El Abbadi, Skeen, and Cristian [12], and is an extension of [11]. In contrast to the method in [12], we do not require a separate protocol to coordinate the views that different sites have of the communication network. This results in a simpler and more efficient protocol. In contrast to [11] and [12], our protocol allows transactions in more than one partition to write the same object. Furthermore, our protocol provides a higher degree of data availability, and greater flexibility in determining the costs of accessing data.

In most previous replica control protocols that tolerate communication failures, the more available an object is to be, the more expensive are operations on it. For example, *quorum-based protocols* [9, 14] 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, as in [11] and [12] 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 high degree of data availability, as well as 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 in Section 4 we prove it correct. In Section 5 we present several optimizations to the protocol.

A comparison with other replica control protocols 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. We assume that a transaction reads and writes an object x at most once, and if t_i reads and writes x, then it reads x before writing x. 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*; that is, before a transaction terminates, it either *commits* or *aborts* all changes it made to the database. Skeen [23] proves that if a network partition occurs during the execution of a transaction, no atomic commit protocol can guarantee the termination of that transaction (as long as the partition persists); that is, partitions may prevent some transactions from terminating. For commit protocols in the presence of partition failures see [5], [6], and [23]. In this paper (as in other concurrency control protocols such as [18] and [25]) we do not address this problem. Rather, our goal is to increase the availability of data in the presence of site and communication failures, including network partitioning, and to consistently restore the database after recovery.

In this section most of the definitions and correctness criteria are drawn from the model developed in [2] and [4]. Consider a set of transactions $T = \{t_1, t_2, \ldots, t_n\}$. We augment T with two special transactions: an initial transaction t_{init} that initializes the database, and a final transaction t_{final} that reads the final state of the database. Formally, $t_{\text{init}}(t_{\text{final}})$ consists solely of write (read) operations, one for each object read or written by a transaction in T. The execution of the transactions in T is modeled by logs. Formally, a *log* L over T is a partial order $(S, <_L)$, where S is the set of all operations executed by transactions in $T \cup \{t_{\text{init}}, t_{\text{final}}\}$, and $<_L$ reflects the order in which the operations were executed. We consider only logs L that start with the write operations of t_{init} followed by all the operations of transactions in T, and end with the read operations of t_{final} [20].

2.1 Correctness Criteria for Nonreplicated Databases

We first consider nonreplicated 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 transactions t_i, t_j (including t_{init} and t_{final}) and any object 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 to be our correctness criterion. The problem of determining whether an arbitrary log is serializable is NP-complete [20]; thus, it is unlikely that there exist 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* [17, 20] 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 $\{t_i \rightarrow t_j \mid \exists\ op_i$ 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 [13, 17].

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* that is initialized by the initial transaction t_{init} (the initial transaction executes for each object x a write operation $w_{\text{init}}[x]$ that results in write operations on all copies of object x). 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 *nonreplicated 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 $w_i[x]$ (other than $w_{\text{init}}[x]$) is executed by

(1) selecting a set of copies of x,
(2) determining *vnmax*, the maximum version number of the selected copies, and
(3) writing all the selected copies with a version number greater than *vnmax*.

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, and
(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 $w_i[x_p], \dots, w_i[x_q]$. For each logical read $r_i[x]$, there is a set $\{x_p, \dots, x_j, \dots, x_q\}$, the set of copies accessed, such that L contains $a_i[x_p] <_L r_i[x_j], \dots, a_i[x_j] <_L r_i[x_j], \dots, a_i[x_q] <_L r_i[x_j]$ (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 (including t_{init} and t_{final}) 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* [2] if it is equivalent to a serial log over the same set of transactions executed over a nonreplicated 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: $\{t_i \longrightarrow t_j \mid \exists\ op_i$ 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 $\Rightarrow_x$ on all transactions that write 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.

Bernstein and Goodman [2] 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 conflict-preserving concurrency control protocol, for example, two-phase locking [13]. Such a protocol ensures that logs are CP-serializable only at the level of *copies* (but not at the object level). In this section we present a replica control protocol that ensures that all logs are one-copy serializable, and hence that transactions are serializable at the object level.

To describe the execution of transactions, we introduce the notion of a view [12]. Each site s maintains a set of sites called its *view*. Each site can independently decide which sites to include in its current view. For example, a site s may choose to include in its view all sites with which s assumes it can communicate. The correctness of our replica control protocol does not depend on this choice, but data availability and operation costs may be affected (in Section 3.3, we describe several possible strategies that sites can use to choose their current view and the trade-offs involved). A user transaction t that is initiated at a site with view v is said to *execute in* v. Informally, view v determines which objects t can read and write, as well as which copies it can access or write. Views are totally ordered according to a unique *view_id* assigned to each view, and two sites are said to have the *same view* if they have identical view-ids.

Our protocol ensures one-copy serializability by (1) ensuring that all transactions executed in one view are one-copy serializable, and (2) ensuring that all transactions executing in a "lower" view are serialized before transactions executing in a "higher" view. Satisfying conditions (1) and (2) enforces a serialization of all transactions executing in all views [10].

3.1 Accessibility Thresholds, Quorums, and User Transactions

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

$$A_r[x] + A_w[x] > n[x]. \tag{1}$$

This relationship ensures 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 $A_r[x]$. (In [11] and [12], the write accessibility threshold has to satisfy the additional requirement that $2A_w[x] > n[x]$, and this requirement prevents transactions in more than one partition to concurrently write 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 physical access and write 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, $n[x, v] = |\, sites[x] \cap v \,|$). For each view v, the quorums of object x must satisfy the following relations:

$$q_r[x, v] + q_w[x, v] > n[x, v], \tag{2}$$

$$2q_w[x, v] > n[x, v], \tag{3}$$

$$1 \le q_r[x, v] \le n[x, v], \tag{4}$$

$$A_w[x] \le q_w[x, v] \le n[x, v]. \tag{5}$$

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 $q_r[x, v]$, $q_w[x, v]$, and $A_r[x]$.

Read operations use the version numbers associated with each copy to identify (and read) the most "up-to-date" copy accessed (as defined in Section 2.2). In our protocol, version numbers consist of two fields $\langle v\_id, k\rangle$. Intuitively, if a copy has version number $\langle v\_id, k\rangle$, then this copy was last written by a transaction t executing in a view v with view-id v_id, and t is the kth transaction to write x in view v. A version number $\langle v_1\_id, k_1\rangle$ is *less than* $\langle v_2\_id, k_2\rangle$, if $v_1\_id < v_2\_id$, or $v_1\_id = v_2\_id$ and $k_1 < k_2$. Initially, sites have a common view v_0 with view-id v_0_id, and all copies have version number $\langle v_0\_id, 0\rangle$. We now describe how user transactions execute read and write operations according to our protocol.

A user transaction t executing in view v can read (write) an object x only if x is read (write) accessible in view v. (Note that a site can determine whether an object is read or write accessible from its local view only, i.e., without accessing any copies.) Furthermore, t can only access, read or write copies of x that reside on sites with view v (this restriction is relaxed in Section 5.1). If object x is read accessible in view v, t executes the logical operation $r[x]$ by

(1) physically accessing $q_r[x, v]$ copies of x residing on sites in v (with view v),
(2) determining *vnmax*, the maximum version number of the selected copies, and
(3) reading the accessed copy with version number *vnmax*.

If object x is write accessible in view v, with view-id v_id, t executes the logical operation $w[x]$ by

(1) selecting $q_w[x, v]$ copies of x residing on sites in v (with view v),
(2) determining *vnmax*, the maximum version number of the selected copies, and
(3) writing all the selected copies and updating their version numbers to $\langle v\_id, l\rangle$, where $l \geq 1$ is the smallest integer such that $\langle v\_id, l\rangle$ is greater than *vnmax*.

If a user transaction tries to access a copy that resides on a site with a view different from the view of the site where the issuing transaction is initiated, that transaction is aborted.

Quorum relations (2) and (3) ensure that all logically conflicting operations issued by user transactions executing in the same view, also physically conflict. Furthermore, since all transactions use version numbers and a conflict-preserving concurrency control protocol, one can show that all transactions executing in the same view are one-copy serializable.

The use of accessibility thresholds in conjunction with quorums and the fact that each view can independently define its own quorums for each object gives the database designer an unusual degree of flexibility. This can be used to achieve the desired cost/availability trade-off of read and write operations. There are several such trade-offs. For example, one can increase the read availability of an object by decreasing the read accessibility threshold, $A_r[x]$—at the cost of increasing $A_w[x]$—that is, decreasing the write availability of the object. In some applications, read operations on some object x outnumber write operations, and in this case it is advantageous to allow inexpensive read operations for x. By using quorums this can be easily achieved with $q_r[x, v] = 1$, and $q_w[x, v] = n[x, v]$. A more detailed discussion of the possible trade-offs in choosing the accessibility thresholds and quorums is presented in Sections 3.3, 5.1.2, and 5.2.

3.2 Update Transactions

Views change during system execution. For example, a site may want to change its view when it notices a discrepancy between its current view and the sites it can actually communicate with (again, this is not necessary for correctness,

```
/* initiates and installs a view with view-id greater than current_view_id */
initiate_new_view(current_view_id)
new_view := {set of sites}                                  /* choose a new view */
new_view_id := increment (current_view_id)
if install_view(new_view, new_view_id) is aborted           /* install the new view */
  → initiate_new_view(new_view_id)
fi
```

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

but may affect performance). A site s may change its view in two different ways. Site s may decide on the members of a new view v based on its own information, in which case s is called the *initiator* of v. (Policies for determining when to initialize a new view, and which sites to include in the new view are discussed in the next section.) Site s may also decide to adopt a view v initiated by another site, in which case we say s *inherits* view v. 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* that updates the local copies residing on site s. Views are considered objects; therefore, a transaction that executes an operation on a view must follow the concurrency control protocol.

Informally, sites change their views as follows. When a site s decides to initiate a new view, it first assigns to the new view a unique view-id *new_view_id* that is larger than any other the initiator has encountered (uniqueness can be achieved by appending the initiator's site identifier to the view-id). Site s then executes an update transaction to update all its local copies. For each object x that is read accessible in the new view, this update transaction accesses enough copies of x to have at least one copy in common with the copies written by write operations executed in previous views (sites with a view-id greater than *new_view_id* reject these access operations). The update transaction reads the value of the accessed copy of x with the highest version number, and writes it to the local copy x_s. If the update transaction is terminated successfully, then s *installs* the new view, and user transactions may execute in the new view and access or write copies residing on s. All sites successfully accessed by the update transaction either have the same new view or they immediately try to inherit this view by executing an update transaction. We now present in more detail the process of changing views.

Let s be a site whose current view has view-id *current_view_id*. To initiate a new view, s atomically executes the procedure *initiate_new_view* (*current_view_id*) illustrated in Figure 1. Site s chooses a new view, *new_view*, and determines an associated view-id, *new_view_id*, higher than the current view-id. Then s tries to install this new view by executing *install_view*(*new_view*, *new_view_id*). If it fails, it initiates a new view with a higher view-id by calling *initiate_new_view* recursively.

Procedure *install_view* (see Figure 2) executes an update transaction to update local copies of objects. If the update transaction is successful, s *installs* *new_view* by updating *current_view* and *current_view_id* to *new_view* and *new_view_id*, respectively. If the update transaction fails, *install_view* is aborted. Once *new_view* is installed at a site s, user transactions initiated at s are allowed to execute in *new_view*.

```
/* tries to install new_view with new_view_id */
install_view(new_view, new_view_id)
execute update_transaction(new_view, new_view_id)
if update_transaction is not aborted                          /* install new_view */
  → current_view := new_view
    current_view_id := new_view_id
  □ update_transaction is aborted
  → abort install_view
fi
```

Fig. 2. *install_view* procedure executed at site *s* to install a view.

```
/* tries to update the local copies of all read accessible objects in new_view */
update_transaction(new_view, new_view_id)
for all read accessible objects x in new_view with a copy in site s do
  select a set of $A_r[x]$ copies of x including $x_s$
  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 vnmax
      write $x_s$ with the value read and version number ⟨new_view_id, l⟩
        where $l \geq 0$ is the smallest integer such that ⟨new_view_id, l⟩ ≥ vnmax
  □ some access operation aborted
    → abort update transaction
  fi
od
```

Fig. 3. *update_transaction* executed at site *s* to update its local copies according to *new_view*.

Update transactions ensure that local copies of read accessible objects are up-to-date (see Figure 3). *Update_transaction*(*new_view*, *new_view_id*) executed at site *s* updates all the local copies of objects that are read accessible in *new_view*. For each object x that is read accessible in *new_view*, it first accesses $A_r[x]$ copies of x (including x_s) and determines *vnmax*, the maximum version number of the selected copies. It then reads the copy associated with *vnmax* and writes it into x_s with version number ⟨*new_view_id*, l⟩, where $l \geq 0$ is the smallest integer so that ⟨*new_view_id*, l⟩ is greater than or equal to *vnmax*.

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 whose view-id is higher than *new_view_id*. If any access operation aborts, the update transaction is also aborted.

When a site p receives a request to access x_p from s, it can take three possible actions depending on *new_view_id*, the view-id associated with the request, and *current_view_id*, the view-id associated with p's current view (see Figure 4, where each branch of the if statement is atomically executed). If *current_view_id* is less than *new_view_id*, then p executes the access operation, and immediately attempts to inherit *new_view*. No other user transaction can access or write x_p before p terminates the inherit procedure. If *current_view_id* is equal to *new_view_id*, then p executes the access operation. If *current_view_id* is greater than *new_view_id*, then p aborts the access operation.

```
/* process an access operation requested by an update transaction initiated at site $s$ */
$access(x_p, new\_view, new\_view\_id)$
if $current\_view\_id < new\_view\_id \longrightarrow$ access $x_p$ and return version number of $x_p$ to $s$
                                             if $p \neq s \longrightarrow$
                                               execute $inherit\_view(new\_view, new\_view\_id)$
  □ $current\_view\_id = new\_view\_id \longrightarrow$ access $x_p$ and return version number of $x_p$ to $s$
  □ $current\_view\_id > new\_view\_id \longrightarrow$ abort access operation
fi
```

Fig. 4. Response of site p with view *current_view* and view-id *current_view_id* to a request $access(x_p, new\_view, new\_view\_id)$ by site s.

```
/* inherits and installs $view$ or initiates a new view with a higher view-id */
$inherit\_view(view, view\_id)$
if $install\_view(view, view\_id)$ is aborted
  $\longrightarrow initiate\_new\_view(view\_id)$
fi
```

Fig. 5. *inherit_view* procedure executed at site s to inherit a view.

To inherit a view *view* with view-id *view_id*, site s atomically executes the procedure *inherit_view(view, view_id)* illustrated in Figure 5. In this procedure, site s tries to install *view*. If it fails, then it executes *initiate_new_view(view_id)* to initiate a view with a higher view-id than *view_id*. Note that when a site initiates or inherits a view, the corresponding view-id is always greater than any previous view-ids at that site.

3.3 Policies for Changing Views

The protocol described in Figures 1, 2, 3, 4, and 5 includes all the basic steps necessary for ensuring one-copy serializability. However, it does not include several possible optimizations or options for implementation. In the next section, we prove that this basic protocol ensures one-copy serializability, and then in Section 5, we describe several optimizations that can be incorporated into our protocol. As we mentioned before, perfect knowledge of a site's communication capabilities is not necessary for determining which sites are in its view. Furthermore, the protocol's correctness depends neither on when a view is changed nor on which sites to include in a new view. In this section we present an example that illustrates some of the options for changing views, and the trade-offs involved. We then discuss some possible *tracking policies*: These determine when a site should change its view, and which sites to include in the new view.

Consider a database with four sites s_1, s_2, s_3, and s_4, where object x has three copies residing on sites s_1, s_2, and s_3, and object y has three copies residing on sites s_2, s_3, and s_4. Assume that all accessibility thresholds are equal to 2; that is, $A_r[x] = A_w[x] = A_r[y] = A_w[y] = 2$. Initially all sites have the same view $v_0 = \{s_1, s_2, s_3, s_4\}$ with read quorums $q_r[x, v_0] = q_r[y, v_0] = 1$ and write quorums $q_w[x, v_0] = q_w[y, v_0] = 3$.

Assume that the network partitions into $P_1 = \{s_1, s_2\}$, and $P_2 = \{s_3, s_4\}$. Sites s_1 and s_2 in partition P_1 have two options. If they install a *new* view $v_1 = \{s_1, s_2\}$ (with read quorum $q_r[x, v_1] = 1$, and $q_w[x, v_1] = 2$) to reflect the partitioning of the network, then x will remain read and write accessible. But since $A_r[y] = A_w[y] = 2$ and there is only one copy of y in v_1, y will not be read nor write accessible in this new view. If s_1 and s_2 keep their *old* view $v_0 = \{s_1, s_2, s_3, s_4\}$ and their old quorums (even though they cannot communicate with s_3 and s_4), then transactions initiated in P_1 can still read both x and y. Note that even though x and y are also write accessible in v_0 (by definition), no transaction initiated in partition P_1 will be able to write x or y since $q_w[x, v_0] = q_w[y, v_0] = 3$ and this requires the writing of three copies of x or y. In summary, if sites in P_1 install the new view (in accordance with their new communication capabilities), they will retain the ability to read and write x, but will lose the ability to read or write y. If they keep their old view, however, they will retain the ability to read x and y, but will lose the ability to write x.

Hence, the database designer has several tracking strategies to choose from. Views could track changes in the network topology as closely as possible, thus reducing the risk of aborting transactions that read or write objects that are (by definition) accessible in the current view. This strategy was called *aggressive tracking* in [7]. (In our example above, this strategy would lead sites s_1 and s_2 to install the new view v_1.) Another approach is to change a view only if some high-priority objects are read accessible in the new view. A variation of this strategy is called *lazy tracking* in [7]. Our protocol with lazy tracking ensures *optimal availability* according to an availability measure proposed in [7]. A third approach is to change a view only when some high-priority transactions can no longer execute in the old view, but would be able to execute in the new view. We call this approach *demand tracking.* Thus, our protocol accommodates several strategies for changing views, and the database designer may dynamically choose the strategy used according to the immediate objectives and needs of the specific database being considered.

4. PROOF OF CORRECTNESS

In this section we prove that the basic protocol described in the previous section ensures one-copy serializability. Before proceeding to prove the correctness of the protocol, we extend the standard serializability theory to include both user and update transactions.

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 *directly 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 *directly reads_x_from* t_i, then t_u reads the value of x written by t_i and writes it into x_s. Now, if t_j *directly 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}, $t_{u_2}, \ldots, t_{u_n}$ be a sequence of update transactions and t_i and t_j be two user transactions. We say t_j *indirectly reads_x_from* t_i if t_{u_1} *directly reads_x_from* t_i, t_{u_2} *directly reads_x_from* $t_{u_1}, \ldots,$ and t_j *directly reads_x_from* t_{u_n}. We henceforth refer to both *directly* and *indirectly reads_x_from* relations simply as *reads_x_from*.

The serialization graph $SG[L]$ for the log L is a directed graph whose nodes are all user and update transactions and where edges capture physical conflicts between transactions. Formally, the edges of $SG[L]$ are $\{t_i \rightarrow t_j \mid \exists\ op_i$ executed by t_i and op_j executed by t_j such that op_i *physically* conflicts with op_j and $op_i <_L op_j\}$. We redefine $1{-}SG[L]$ to ensure that it has a path between any two transactions issuing logically conflicting operations. $1{-}SG[L]$ must have enough edges so that

(1) For each object x, $1{-}SG[L]$ embodies a total order $\Rightarrow_x$ on all user transactions that write x.
(2) For any two user transactions t_i and t_j, if t_j *reads_x_from* t_i (directly or indirectly), then $1{-}SG[L]$ has a path from t_i to t_j.
(3) For any three user transactions t_i, t_j, and t_k, and t_j *reads_x_from* t_i (directly or indirectly) and $t_i \Rightarrow_x t_k$, then $1{-}SG[L]$ has a path from t_j to t_k.

The proof of Theorem 2 in [2] still holds 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 at the level of copies. Let $SG[L]$ be the corresponding serialization graph. Note that CP-serializability ensures the acyclicity of $SG[L]$ [13, 17]. We now show how to extend $SG[L]$ into a $1{-}SG[L]$ by adding enough edges to satisfy the three requirements described in Section 4.1. The next lemma proves that $SG[L]$ already satisfies the second of these three requirements.

LEMMA 1. *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. If t_j *directly reads_x_from* t_i, then (by definition) there is a copy x_s that t_i writes and is subsequently read by t_j. Hence, $SG[L]$ has an edge from t_i to t_j. If t_j *indirectly reads_x_from* t_i, then there is a sequence of update transactions

$t_{u_1}, t_{u_2}, \ldots, t_{u_n}$ such that t_{u_1} *directly reads_x_from* t_i, t_{u_2} *directly reads_x_from* $t_{u_1}, \ldots, t_j$ *directly reads_x_from* t_{u_n}. Therefore, $SG[L]$ has an edge from t_i to t_{u_1}, from t_{u_1} to $t_{u_2}, \ldots$, and from t_{u_n} to t_j. □

We now define an order $\Rightarrow_x$ on all transactions that write x as follows:

$t_i \Rightarrow_x t_j$ where t_i and t_j write x, if and only if the version number t_i assigns to x is less than the one t_j assigns to x.

To show that $\Rightarrow_x$ defines a *total* order on all user transactions that write x we need the following technical lemma.

LEMMA 2. *The version number of a copy never decreases.*

PROOF. The version number of a copy x_s is changed only by a user transaction or an update transaction. In the first case, our write rule for user transactions ensures that the version number of x_s increases. In the second case, our rule for update transactions ensures that the version number of x_s does not decrease. □

LEMMA 3. *If t_i and t_j are distinct user transactions that write x, then t_i assigns to x a different version number than t_j (and thus, either $t_i \Rightarrow_x t_j$ or $t_j \Rightarrow_x t_i$).*

PROOF. There are two cases to consider. If t_i and t_j execute in the same view v, then both t_i and t_j write $q_w[x, v]$ copies of x in v. Since $2q_w[x, v] > n[x, v]$, there must be at least one copy x_s in v that both t_i and t_j write. Without loss of generality assume t_i writes x_s before t_j does. By Lemma 2, and our write rule, t_j must write x with a higher version number than t_i. If t_i and t_j execute in different views, then by definition the first field of the version numbers they assign to x must be different. □

Lemma 1 shows that $SG[L]$ already satisfies the second requirement of a $1-SG[L]$. We now prove that $SG[L]$ partially satisfies the first requirement as well, with respect to the total order $\Rightarrow_x$ defined above.

LEMMA 4. *If $t_i \Rightarrow_x t_j$, and t_i and t_j are two user transactions* executing in the same view, *then $SG[L]$ has an edge from t_i to t_j.*

PROOF. Suppose t_i and t_j execute in view v. Thus, both t_i and t_j write $q_w[x, v]$ copies of x in v. Since $2q_w[x, v] > n[x, v]$, there must be at least one copy x_s in v that both t_i and t_j write. Since $t_i \Rightarrow_x t_j$, t_i writes x_s with a smaller version number than t_j. Thus, from our rules for write operations it is clear that t_i writes x_s before t_j does. Hence, by definition $SG[L]$ contains an edge from t_i to t_j. □

To prove that $SG[L]$ partially satisfies the third requirement of $1-SG[L]$, we first need the following technical lemma. To simplify the presentation, we define $v[t]$ as follows. If t is a user transaction, then $v[t]$ is the view in which t was executed. 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 $v[t]$.

LEMMA 5. *If an update transaction t_u reads_x_from t_i and $t_i \Rightarrow_x t_k$, where t_k is a user transaction, then $t_u \Rightarrow_x t_k$.*

PROOF. There are two cases to consider:

(1) Suppose that t_u *directly reads_x_from* t_i. We first show that there is a copy x_s that is first accessed by t_u and later written by t_k. Update transaction t_u accesses $A_r[x]$ copies of x and user transaction t_k writes $q_w[x, v] \geq A_w[x]$ copies of x (for some v). Since $A_r[x] + A_w[x] > n[x]$ (the total number of copies of x), there must be at least one common copy x_s that t_u accesses and t_k writes. Since $t_i \Rightarrow_x t_k$, then, by the definition of $\Rightarrow_x$, the copies of x that t_i writes have a smaller version number than the copies written by t_k. By Lemma 2, the version number of a copy never decreases. Since t_u *directly reads_x_from* t_i and not from t_k, it must be that t_u accesses x_s before t_k writes it.

    We claim that t_k assigns a greater version number to x than t_u does (hence, $t_u \Rightarrow_x t_k$). Since t_u directly reads the value of x written by t_i, then t_u reads a copy with version number $vnmax = \langle v\_id[t_i], k\rangle$, where $v\_id[t_i] \leq v\_id[t_u]$ and $k \geq 0$. Suppose that $v\_id[t_u] = v\_id[t_i]$. In this case, the update transaction t_u writes x with version number $\langle v\_id[t_i], k\rangle$, the same version number that t_i assigned to x. Thus, since $t_i \Rightarrow_x t_k$, we also have $t_u \Rightarrow_x t_k$. Now suppose that $v\_id[t_i] < v\_id[t_u]$. In this case, t_u writes x with version number $\langle v\_id[t_u], 0\rangle$. We now show that t_k writes x with version number $\langle v\_id[t_k], l\rangle$, where $v\_id[t_k] \geq v\_id[t_u]$ and $l \geq 1$. Recall that there is a copy x_s that t_u accesses before t_k writes it. When t_u accesses x_s, site s must have a view whose view-id is less than or equal to $v\_id[t_u]$. In the former case, s must immediately execute *inherit_view*$(v[t_u], v\_id[t_u])$. Thus, in both cases, by the time t_k writes x_s, site s must have a view with view-id $v\_id[t_k] \geq v\_id[t_u]$. Thus, t_k writes x with version number $\langle v\_id[t_k], l\rangle$ for some $l \geq 1$, which is greater than $\langle v\_id[t_u], 0\rangle$, the version number assigned by t_u. Hence, $t_u \Rightarrow_x t_k$.

(2) Suppose that t_u *indirectly reads_x_from* t_i. There must be a sequence of update transactions $t_{u_1}, t_{u_2}, \ldots, t_{u_{n-1}}$ such that t_u *directly reads_x_from* $t_{u_{n-1}}, \ldots, t_{u_2}$ *directly reads_x_from* t_{u_1}, t_{u_1} *directly reads_x_from* t_i. Since $t_i \Rightarrow_x t_k$, by Case (1), we have $t_{u_1} \Rightarrow_x t_k$. Since $t_{u_1} \Rightarrow_x t_k$, by Case (1) again, we have $t_{u_2} \Rightarrow_x t_k$. It is now clear that a simple induction shows that $t_u \Rightarrow_x t_k$. □

We can now prove that $SG[L]$ partially satisfies the third requirement of $1\text{-}SG[L]$.

LEMMA 6. *If* t_j *reads_x_from* t_i, $t_i \Rightarrow_x t_k$, *and* t_j *and* t_k *are user transactions* executing in the same view, *then* $SG[L]$ *has an edge from* t_j *to* t_k.

PROOF. t_j reads x and t_k writes x in the same view v. By the quorum relation $q_r[x, v] + q_w[x, v] > n[x, v]$, there must be a copy x_s that t_j accesses and t_k writes. There are two cases to consider.

(1) Suppose that t_j *directly reads_x_from* t_i. Since $t_i \Rightarrow_x t_k$, then by the definition of $\Rightarrow_x$, t_k writes x_s with a greater version number than the one t_i assigns to the copies of x. By Lemma 2, the version number of a copy never decreases. Since t_j directly reads the value of x written by t_i and not t_k, then it must be that t_j accesses x_s before t_k writes it. Hence, by definition, $SG[L]$ has an edge from t_j to t_k.

(2) Suppose that t_j *indirectly reads_x_from* t_i. Then there must be an update transaction t_u such that t_j *directly reads_x_from* t_u and t_u *reads_x_from* t_i (directly or indirectly). Since $t_i \Rightarrow_x t_k$, then, by Lemma 5, $t_u \Rightarrow_x t_k$. Since t_j *directly reads_x_from* t_u, then, by Case (1), $SG[L]$ has an edge from t_j to t_k. □

Lemmas 1, 4, and 6 show that we can extend $SG[L]$ into $1{-}SG[L]$ by adding only the following edges:

(1) If $t_i \Rightarrow_x t_j$, and t_i and t_j are user transactions executing in *different* views, then add an edge from t_i to t_j. These are *write* edges.

(2) If t_j *reads_x_from* t_i, $t_i \Rightarrow_x t_k$, and t_j and t_k are user transactions executing in *different* views, 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 *write* edges and *reads-before* edges between user transactions executing in different views.

Informally, the edges of $SG[L]$ capture a serialization order between any two user transactions that issue physically conflicting operations. However, with our threshold and quorum assignments, a *read* x and a *write* x executing in different views do not necessarily physically conflict. The same holds for two *write* x operations. Thus, we extend $SG[L]$ into $1{-}SG[L]$, a graph that orders all the user transactions that execute logically conflicting operations, by adding *write* and *reads-before* edges between user transactions executing in different views.

In the next section we prove that transactions executing in different views are serialized according to the view-ids of those views; that is, if $v\_id[t] < v\_id[t']$, then t is serialized before t' in the global serialization order.

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. We first prove that if $SG[L]$ has an edge from transaction t_i to transaction t_j, then $v\_id[t_i] \le v\_id[t_j]$. We then extend this result to the edges of $1{-}SG[L]$.

LEMMA 7. *If $SG[L]$ has an edge from t_i to t_j, then $v\_id[t_i] \le v\_id[t_j]$.*

PROOF. The edges of $SG[L]$ are between transactions that execute physically conflicting operations. Let $op_i[x_s]$ and $op_j[x_s]$ be the two physically conflicting operations executed by t_i and t_j. Since $SG[L]$ has an edge from t_i to t_j, then $op_i[x_s] <_L op_j[x_s]$. Denote by v_i the view of s when $op_i[x_s]$ is executed, and by v_i_id the view-id of v_i. There are two cases to consider depending on whether t_i is a user or update transaction:

(1) t_i is a user transaction. Therefore, when $op_i[x_s]$ is executed, s must have a view $v_i = v[t_i]$ with $v_i\_id = v\_id[t_i]$. When $op_j[x_s]$ is later executed, s must have a view v_j with view-id v_j_id such that $v_j\_id \le v\_id[t_j]$. Since $op_i[x_s] <_L op_j[x_s]$, v_i is installed at s before v_j. But a site installs views with increasing view-ids; therefore, $v_i\_id \le v_j\_id$. Hence, $v\_id[t_i] \le v\_id[t_j]$.

(2) t_i is an update transaction. From Figure 4, it is clear that when $op_i[x_s]$ is executed, s must have a view v_i with $v_i\_id \le v\_id[t_i]$. If $v_i\_id = v\_id[t_i]$, then the proof of Case (1) can be applied to show that $v\_id[t_i] \le v\_id[t_j]$. If $v_i\_id < v\_id[t_i]$, then, immediately after the execution of $op_i[x_s]$, s executes

inherit_view$(v[t_i], v\_id[t_i])$. Thus, s installs a view v'_i with view-id $v'_i\_id \geq v\_id[t_i]$. Since $op_i[x_s] <_L op_j[x_s]$, t_j is either the update transaction triggered by the installation of v'_i by s, or a user transaction or an update transaction that operates on s after the installation of v'_i. In the first case, $v[t_j] = v'_i$ and $v\_id[t_j] = v'_i\_id \geq v\_id[t_i]$. Proof of the second case is similar to the proof of Case (1). When $op_j[x_s]$ is executed, s must have a view v_j with view-id $v_j\_id \leq v\_id[t_j]$ and $v_j\_id \geq v'_i\_id$. Combining inequalities results in $v\_id[t_i] \leq v'_i\_id \leq v_j\_id \leq v\_id[t_j]$. □

We now extend the previous lemma to edges in $1{-}SG[L]$.

LEMMA 8. *If* $1{-}SG[L]$ *has an edge from* t_i *to* t_j *then* $v\_id[t_i] \leq v\_id[t_j]$.

PROOF. If the edge $\langle t_i, t_j\rangle$ is in $SG[L]$, Lemma 7 implies that $v\_id[t_i] \leq v\_id[t_j]$. If the edge $\langle t_i, t_j\rangle$ is not in $SG[L]$, then it is either a *write* edge or a *reads-before* edge between user transactions that execute in different views. If $\langle t_i, t_j\rangle$ is a *write* edge, then by definition $t_i \Rightarrow_x t_j$. Thus, the version number assigned by t_i, $\langle v\_id[t_i], k_i\rangle$, is less than the one assigned by t_j, $\langle v\_id[t_j], k_j\rangle$, and therefore $v\_id[t_i] \leq v\_id[t_j]$.

Hence, we only have to consider the case where $\langle t_i, t_j\rangle$ is a *reads-before* edge between two user transactions. In this case, there must be a transaction t_h such that t_i *reads_x_from* t_h and $t_h \Rightarrow_x t_j$. There are two possible cases:

(1) t_i and t_h execute in the same view; that is, $v\_id[t_i] = v\_id[t_h]$. Since $t_h \Rightarrow_x t_j$, then $v\_id[t_h] \leq v\_id[t_j]$. Therefore, $v\_id[t_i] \leq v\_id[t_j]$.

(2) t_i and t_h execute in different views. Thus, t_i *indirectly reads_x_from* t_h; that is, there must be an update transaction t_u such that t_i *directly reads_x_from* t_u and t_u *reads_x_from* t_h. Since user transaction t_i *directly reads_x_from* t_u, we have $v\_id[t_u] = v\_id[t_i]$. Since update transaction t_u *reads_x_from* t_h and $t_h \Rightarrow_x t_j$, then by Lemma 5, $t_u \Rightarrow_x t_j$. Thus, $v\_id[t_u] \leq v\_id[t_j]$ and $v\_id[t_i] \leq v\_id[t_j]$. □

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 that $1{-}SG[L]$ has a cycle. From Lemma 8, it is clear that for any two transactions t_i and t_j in this cycle, $v\_id[t_i] = v\_id[t_j]$, and hence $v[t_i] = v[t_j]$. That is, all transactions in this cycle execute in the same view. Since $SG[L]$ is acyclic, the cycle in $1{-}SG[L]$ has at least one edge that is not in $SG[L]$. Thus, it has at least one *reads-before* or *write* edge between two transactions executing in different views, a contradiction. Hence, $1{-}SG[L]$ is acyclic. □

5. OPTIMIZATIONS

The basic protocol, presented in Section 4, is sufficient to ensure one-copy serializability. However, its implementation can incorporate several optimizations to increase the protocol's efficiency. In this section we describe two types of optimizations: those that increase data availability and those that reduce the costs of update transactions.

5.1 Increasing Data Availability

In the basic protocol, the availability of an object x is limited by the fact that a user transaction executing in a view v is only allowed to touch (access, read, or write) copies that reside on sites with the *same* view v. Such a transaction is aborted if it cannot touch a quorum of copies residing on sites with this view. In this section we propose two approaches to relax this restriction. These optimizations are easy to integrate with the basic protocol.

5.1.1 *Relaxing the Read and Write Rules.* With the basic protocol, a user transaction t executing in some view can only access and write copies residing on sites with the same view. To increase data availability, we first relax this rule for write operations, while maintaining the restriction on read operations. We then discuss relaxing the restriction on read operations.

Relaxed Write Rule. A user transaction t executing in view v writes an object x by writing $q_w[x, v]$ copies of x residing on sites in v and with a view whose view-id is *less than or equal to* $v\_id[t]$. Once a copy x_s is written by t, it rejects *all* operations issued by user transactions executing in views with view-ids less than $v\_id[t]$. Furthermore, later views installed at site s must have a view-id greater than or equal to $v\_id[t]$.

A simple way to enforce this rule is to require a site with view v that processes a write operation of transaction t, where $v \neq v[t]$, to immediately execute *install_view*$(v[t], v\_id[t])$. Note that a *read* operation of object x executed by t must still access $q_r[x, v]$ copies residing on sites in v, with view v. The Relaxed Write rule still ensures one-copy serializability. The proof of correctness follows the one in Section 4, with some minor modifications. In particular, the proofs of Lemmas 1–4 do not change. The proof of Lemma 5 needs a slight modification that we leave to the reader. The proof of Lemma 6 does not change, and the proof of Lemma 7 is slightly modified as follows.

LEMMA 7′. *If $SG[L]$ has an edge from t_i to t_j, then $v\_id[t_i] \leq v\_id[t_j]$.*

PROOF. The edges of $SG[L]$ are between transactions that execute physically conflicting operations. Let $op_i[x_s]$ and $op_j[x_s]$ be the two physically conflicting operations executed by t_i and t_j. Denote by v_i the view of s when $op_i[x_s]$ is executed, and by v_i_id the view-id of v_i (v_j and v_j_id are similarly defined). Since $SG[L]$ has an edge from t_i to t_j, then $op_i[x_s] <_L op_j[x_s]$. But a site installs views with increasing view-ids; therefore, $v_i\_id \leq v_j\_id$. Irrespective of whether t_i is a user or update transaction, our new rule ensures that when $op_i[x_s]$ ($op_j[x_s]$) is executed, $v_i\_id \leq v\_id[t_i]$ ($v_j\_id \leq v\_id[t_j]$). If $v_i\_id = v\_id[t_i]$, then $v\_id[t_i] \leq v\_id[t_j]$.

Now suppose that $v_i\_id < v\_id[t_i]$. In this case, either $op_i[x_s]$ is a write operation by a user transaction or $op_i[x_s]$ is issued by an update transaction. In both cases, after the execution of $op_i[x_s]$, no operation is executed until s installs a view v' with view-id v'_id such that $v\_id[t_i] \leq v'\_id$. Since t_j is either the update transaction triggered by the installation of v' by s, or a user transaction or an update transaction that operates on s after the installation of v', then $v'\_id \leq v_j\_id$. Hence, $v\_id[t_i] \leq v\_id[t_j]$. □

Finally the proofs of Lemma 8 and Theorem 1 do not change.

Relaxing the read rule introduces a trade-off between availability and cost. To increase data availability, we could allow a user transaction executing in view v to read an object x by accessing copies that reside on sites with views other than v, even if this object is *not* read accessible in v. However, each *read* x must now access $A_r[x]$ copies of x, which possibly reside on sites with views other than v (instead of $q_r[x, v]$ copies that reside on sites in v with view v).

Alternative Read Rule. A user transaction t reads an object x by accessing $A_r[x]$ copies of x residing on sites with a view whose view-id is *less than or equal to* $v\_id[t]$. Once a copy x_s is accessed by t, it rejects all *write* operations issued by user transactions executing in views with view-ids less than $v\_id[t]$. Furthermore, later views installed at site s must have a view-id greater than or equal to $v\_id[t]$.

Either or both of the Relaxed Write or the Alternative Read rules may be used in a given system since they are independent of each other. With the Alternative Read rule, the proofs of Lemmas 1–6 essentially remain the same. A minor modification is needed to the last paragraph of the proof of Lemma 7′ above. If $op_i[x_s]$ is an access operation, then $op_j[x_s]$ must be a write operation (since $op_i[x_s]$ and $op_j[x_s]$ are conflicting operations). By our Alternative Read rule, $op_j[x_s]$ cannot be executed until s installs view v', where $v\_id[t_i] \leq v'\_id$; hence, $v\_id[t_i] \leq v\_id[t_j]$.

The proof of Lemma 8 has to be modified as follows:

LEMMA 8′. *If* $1\text{–}SG[L]$ *has an edge from* t_i *to* t_j *then* $v\_id[t_i] \leq v\_id[t_j]$.

PROOF. The original proof holds except in the case where $\langle t_i, t_j \rangle$ is a *reads-before* edge and t_i uses the Alternative Read rule to read a value written by some t_h, where $t_h \Rightarrow_x t_j$. There are two cases to consider:

(1) Suppose that t_i *directly reads_x_from* t_h. User transaction t_i uses the Alternative Read rule and accesses $A_r[x]$ copies of x, and user transaction t_j writes $q_w[x, v] \geq A_w[x]$ copies of x (for some v). Since $A_r[x] + A_w[x] > n[x]$ (the total number of copies of x), there must be at least one common copy x_s that t_i accesses and t_j writes. Since $t_h \Rightarrow_x t_j$ then, by the definition of $\Rightarrow_x$, the copies of x that t_h writes have a smaller version number than the copies written by t_j. By Lemma 2, the version number of a copy never decreases. Since t_i *directly reads_x_from* t_h and not from t_j, it must be that t_i accesses x_s before t_j writes it. But once t_i accesses x_s using the Alternative Read rule, site s rejects all *write* operations issued by user transactions executing in views with view-ids less than $v\_id[t_i]$. Hence, $v\_id[t_i] \leq v\_id[t_j]$.

(2) Suppose that t_i *indirectly reads_x_from* t_h. Then there must be an update transaction t_u such that t_i *directly reads_x_from* t_u and t_u *reads_x_from* t_h (directly or indirectly). Since $t_h \Rightarrow_x t_j$, then, by Lemma 5, $t_u \Rightarrow_x t_j$. Since t_i *directly reads_x_from* t_u using the Alternative Read rule, and $t_u \Rightarrow_x t_j$, then, by Case (1), $v\_id[t_i] \leq v\_id[t_j]$. □

5.1.2 *Using Multiversions.* Multiversion databases are widely known to increase data availability [18, 22]. We can integrate multiversions into our protocol by simply associating with each copy a sequence of versions, each corresponding to a different view (and view-id). The underlying concurrency control protocol that we now assume ensures CP-serializability at the level of the versions only (not at the level of copies or objects). The versions of the copy are ordered by their associated view-ids. Furthermore, each site maintains a sequence of views installed at that site, with their view-ids and corresponding quorums. For a user transaction t to execute, it first chooses a view v, and we say *t executes in v* (note that v does not have to be the most recent view installed at the initiating site). The latest view installed at site s is considered to be the *view* of s.

With multiversions, an update transaction t_u installing a new view $v[t_u]$, with view-id $v\_id[t_u]$ is executed as follows:

Update Transaction Rule. For each object x that is read accessible in $v[t_u]$:

(1) t_u accesses $A_r[x]$ copies of x; each accessed site rejects any *write* x by user transactions executing in a view with a view-id less than $v\_id[t_u]$;
(2) for each accessed copy, t_u accesses the version with the highest view-id less than or equal to $v\_id[t_u]$;
(3) t_u determines *vnmax*, the maximum version number of the accessed versions, and reads the accessed version associated with *vnmax*;
(4) t_u writes the value read into the $v_id[t_u]$th version of the local copy of x. The associated version number is $\langle v\_id[t_u], l\rangle$, where $l \geq 0$ is the smallest integer so that $\langle v\_id[t_u], l\rangle$ is greater than or equal to *vnmax*.

Note that, unlike the single version case, some sites accessed by t_u may have a view whose view-id is greater than $v\_id[t_u]$.

With multiversions, a user transaction t executing in view v observes the following write rule.

Multiversion Write Rule. If object x is write accessible in view v,

(1) t accesses $q_w[x, v]$ copies of x residing on sites in v;
(2) for each such copy, t accesses the version with the highest view-id less than or equal to $v\_id[t]$;
(3) t determines *vnmax*, the maximum version number of the accessed versions;
(4) t writes the $v_id[t]$th version of all the selected copies and updates their version numbers to $\langle v\_id[t], l\rangle$, where $l \geq 1$ is the smallest integer such that $\langle v\_id[t], l\rangle$ is greater than *vnmax*.

A user transaction t executing in a view v must observe one of the following read rules, which represent another trade-off between cost and availability.

Multiversion Read Rule 1. If object x is read accessible in view v,

(1) t accesses $q_r[x, v]$ copies of x residing on sites in v and with a view whose view-id is *greater than or equal to* $v\_id[t]$;
(2) for each such copy, t accesses the $v_id[t]$th version;
(3) t determines *vnmax*, the maximum version number of the accessed versions, and reads the accessed version associated with *vnmax*.

With the second read rule, a user transaction t executing in view v can read x even if x is not read accessible in v.

Multiversion Read Rule 2. t reads x by executing the first three steps of an update transaction.

The Multiversion Read rules represent a trade-off between availability and cost. Using Multiversion Read rule 1, a transaction t executing in v can read x by accessing $q_r[x, v]$ copies of x that reside on sites that have installed view v (or a view with a higher view-id). In contrast, if t reads x using Multiversion Read rule 2, then it can access copies of x residing on any sites regardless of their views; however, t must access $A_r[x]$ copies (note that $A_r[x] \geq q_r[x, v]$).

Multiversions increase data availability. To illustrate this increase in availability (with respect to single version databases), we reconsider the example discussed in Section 3. Recall that object x has copies residing on sites s_1, s_2, and s_3, and object y has copies residing on sites s_2, s_3, and s_4. Furthermore, $A_r[x] = A_w[x] = A_r[y] = A_w[y] = 2$, and initially in view $v_0 = \{s_1, s_2, s_3, s_4\}$, $q_r[x, v_0] = q_r[y, v_0] = 1$ and $q_w[x, v_0] = q_w[y, v_0] = 3$.

Recall that in the single-version case, after the network partitions into $P_1 = \{s_1, s_2\}$, and $P_2 = \{s_3, s_4\}$, the sites in P_1 have two options: (1) they either keep the old view $v_0 = \{s_1, s_2, s_3, s_4\}$, thus retaining the ability to read x and y, but losing the ability to write x, or (2) they install a new view $v_1 = \{s_1, s_2\}$ (with read quorum $q_r[x, v_1] = 1$, and $q_w[x, v_1] = 2$), thus retaining the ability to read and write x, but losing the ability to read y. In contrast, with multiversions, the sites in P_1 can retain both the ability to execute transactions that read and write x, as well as transactions that read y. They install the new view $v_1 = \{s_1, s_2\}$, and execute transactions as follows. A *read* x or *write* x transaction issued in partition P_1 can be executed in the *new* view v_1 (e.g., *write* x writes the v_1_*id*th version of $q_w[x, v_1]$ copies of x in v_1). A *read* y transaction issued in partition P_1 can be executed in the *old* view v_0 (by reading the v_0_*id*th version of $q_r[y, v_0]$ copies of y in v_0). Note that with multiversions, in P_1, a transaction can read and write x, by executing in new view v_1, while another transaction can read x and y, by executing in view v_0. At the same time, symmetrically, P_2 can execute transactions that read and write y, and others that read x and y. In a single version database, no accessibility thresholds and quorum assignments allow both P_1 and P_2 to execute these transactions.

Multiversions increase data availability, but storing all the versions of each copy can be expensive. To reduce costs, the database may store only the most recent versions of each copy (more than one version but not necessarily all previous versions). In this case, some user transactions may need to access versions that were not saved, and thus be aborted. This intermediate scheme achieves lower availability than multiversions, at a lower cost. Wright studied the degree of availability provided by the class conflict protocol [24, 25] when two or more versions of a copy are stored, and showed an increase in data availability over the single-version case, without incurring the expensive overhead of storing all versions of a multiversion database. In ISIS [19], each copy keeps the two most recent versions for recovery purposes; by using our protocol, an increase in data availability can be achieved when these two versions are used.

5.2 Reducing the Cost of Update Transactions

Various optimizations can be used to reduce the costs associated with update transactions. In this section, we present three such optimizations. We first eliminate the redundancy of read operations executed by update transactions that install the same view at different sites. Second, we reduce the granularity of update transactions by associating views with copies instead of with sites. We also show that this optimization results in an increase in data availability. Finally, we discuss methods for reducing the communication costs of access operations by storing copies in the form of logs of operations.

5.2.1 *Eliminating Redundant Update Transactions.* Our first optimization eliminates the redundancy of read operations executed by update transactions installing the *same* view at different sites. Consider a set of sites installing the same new view. One site initiates the new view, and several others inherit it. For each read accessible object x, we previously required that *all* sites installing this new view to *independently* access $A_r[x]$ copies of x to update their local copy of x. This is clearly redundant and expensive: A single site could execute the update transaction and access $A_r[x]$ copies to update its local copy of x, and then propagate this updated copy to all other sites that want to install the same view. If the new view to be installed contains l copies of x, this optimization reduces by a factor of l the "read cost" of updating all the copies of x during installation of this view.

5.2.2 *Reducing the Granularity of Update Transactions.* The second optimization reduces the granularity of update transactions. Thus far, we associated a view with each site. Therefore, to change the view of a site s, the update transaction initiated at s must update all the local copies of read accessible objects in the new view. As Herlihy pointed out, reducing the granularity of update transactions may have several advantages, and the *quorum consensus protocol* that he describes [18] performs updates on a per object basis, rather than per site. We can achieve the same goal simply by associating a view with each copy of an object instead of with each site. (This is equivalent to considering each copy as residing on a virtual site of its own.) Now each copy x_s has its own view, consisting of a set of copies rather than a set of sites. To change the view of a copy x_s, the update transaction initiated at s must only update the value of x_s.

In addition to reducing the granularity of update transactions, assigning views to copies instead of sites also increases availability. This is illustrated by the example we considered in Section 3. Recall that x has three copies residing on sites s_1, s_2, and s_3, and y has three copies residing on s_2, s_3, and s_4. Furthermore, $A_r[x] = A_w[x] = A_r[y] = A_w[y] = 2$. Initially, all copies of x and y have the same initial view $v_0 = \{x_{s_1}, x_{s_2}, x_{s_3}, y_{s_2}, y_{s_3}, y_{s_4}\}$. After the network partitions into $P_1 = \{s_1, s_2\}$, and $P_2 = \{s_3, s_4\}$, instead of requiring all copies in P_1 to either keep their old view v_0 (thus retaining the ability to read x and y, but losing the ability to write x), or to form a new view $v_1 = \{x_{s_1}, x_{s_2}, y_{s_2}\}$ containing all copies residing on sites in P_1 (thus retaining the ability to read and write x, but losing the ability to read y), reducing the granularity gives us a third alternative.

Copy y_{s_2} may keep its old view v_0, while copies x_{s_1} and x_{s_2} install the new view v_1. Thus, a *read* x or *write* x transaction issued in partition P_1 can be executed in the new view v_1 (e.g., *write* x writes $q_w[x, v_1]$ copies of x). A *read* y transaction issued in partition P_1 can be executed in the old view v_0, by reading $q_r[y, v_0]$ copies of y. Hence, if we associate views with copies, instead of with sites, we allow *read* x, *write* x, and *read* y transactions to execute in P_1. However, we note that a transaction that reads both x and y would not be able to execute in P_1, since this transaction must choose to execute in v_0 or in v_1 (but cannot execute in both).

5.2.3 *Reducing Communication Costs of Update Transactions.* The third optimization may be applied to reduce the communication costs of access operations. This reduces the cost of those update transactions that access many copies. A copy can either be stored explicitly as its complete current state, or as a log of operations executed on that copy [16]. If objects are "large," that is, the complete description of their state is large, the communication costs incurred in transferring the whole state of a copy can be expensive. To reduce this communication cost, a sending site can send a log of all the operations on that object that the receiving site has missed. Suppose that a site s installs a view with view-id v_id, and must update copy x_s, last written by transaction t. Then site s needs to receive only the sequence of *write* x operations executed by transactions in views with view-ids greater than $v\_id[t]$, but less than v_id.

6. COMPARISON WITH OTHER WORK

The first protocol to use the concept of "views" to decrease the costs of read operations was presented in [12]. In that protocol, whenever the communication topology changes, a two-phase protocol is first executed to ensure the consistency of views. Then, an update operation is initiated that updates the different copies to ensure the consistency of data. The cost of a separate view management protocol is a 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.

In [11] and [12], the write accessibility thresholds must satisfy the additional requirement that $2A_w[x] > n[x]$ (in addition to threshold requirement (1)). With this extra requirement, when the network partitions at most one partition can write each object x. Furthermore, in [12], the read and write quorums are fixed to $q_r[x, v] = 1$ and $q_w[x, v] = n[x, v]$ for all views v. Our new protocol allows more flexibility in the choice of quorums: each view may have different quorum assignments for the same object x, as long as they satisfy quorum relations (2), (3), (4), and (5). This, in addition to the greater flexibility in choosing the accessibility thresholds $A_r[x]$ and $A_w[x]$, and the optimizations outlined in Section 5, provides the database designer with a larger degree of freedom in deciding the cost of operations, and the degree of data availability.

The protocol presented in this paper also provides a greater degree of flexibility than the protocols proposed in [9] and [14]. With each object x, Gifford's protocol associates static read and write quorums $Q_r[x]$ and $Q_w[x]$ such that $Q_r[x] + Q_w[x] > n[x]$ and $2Q_w[x] > n[x]$. This is a special case of our protocol where all the sites have the same view v that includes all the sites of the database, and where $q_r[x, v] = A_r[x] = Q_r[x]$ and $q_w[x, v] = A_w[x] = Q_w[x]$.

Eager and Sevick [9] introduced the *Missing Writes Protocol,* which is an extension of the Gifford protocol. Initially, transactions (called *normal mode transactions*) use a read-one, write-all protocol. When a transaction cannot execute due to a failure, it switches to *failure mode,* and for each object it uses read and write quorums $Q_r[x]$ and $Q_w[x]$, as in Gifford's protocol. To ensure serializability, all failure mode transactions leave a *missing write token* at all copies they access or write. When a normal mode transaction encounters a missing write token, it switches to failure mode. This protocol is also a special case of our protocol with the thresholds set to $A_r[x] = Q_r[x]$ and $A_w[x] = Q_w[x]$. Initially, all transactions execute in the *normal view* v_n that includes all the sites of the database and has quorums $q_r[x, v_n] = 1$ and $q_w[x, v_n] = n[x]$. When a site detects a failure, it installs a *failure view* v_f that include all the sites that it can communicate with, and with quorums $q_r[x, v_f] = Q_r[x]$ and $q_w[x, v_f] = Q_w[x]$. As in [11] and [12], when the network partitions, the protocols in [9] and [14] allow at most one partition to write each object x.

The protocol presented in this paper may also reduce the number of physical operations executed, compared to the protocols in [9] and [14]. 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 [9] and [14]. To achieve the same level of availability as Gifford's protocol does, we set the thresholds and quorums of our protocol to $A_r[x] = Q_r[x]$, $A_w[x] = Q_w[x]$, $q_r[x, v] = A_r[x]$, and $q_w[x, v] = n[x, v] + 1 - q_r[x, v]$, for all v. With these threshold and quorum assignments, any read or write operation on x that is executable using Gifford's protocol 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 protocol in [14] requires writing $Q_w[x] = n[x] + 1 - Q_r[x]$ copies, compared to writing $q_w[x, v]$ copies with our protocol. Note that $q_w[x, v] \leq Q_w[x]$, since $n[x, v] \leq n[x]$ and $q_r[x, v] = Q_r[x]$.

To achieve the same level of availability as the Missing Writes protocol does, we set the thresholds and the write quorum as above. To set the read quorum, we consider the views v_n and v_f that correspond to the normal and failure modes of the Missing Writes protocol. We set the read quorum of the normal view to $q_r[x, v_n] = 1$. For a failure view v_f, we set $q_r[x, v_f] = A_r[x]$. This special case of our protocol will mimic the behavior of the Missing Writes protocol by switching to some failure view whenever the Missing Writes protocol switches to failure mode. In the normal mode, the cost of a write operation is the same for both protocols (both protocols write all copies of an object). In the failure mode, write operations may be less expensive with our protocol: It is easy to see that $q_w[x, v_f] \leq Q_w[x]$ since $n[x, v_f] \leq n[x]$ (v_f is a subset of all sites).

As in [11] and [12], our protocol never requires a read operation to physically access more than one copy, not even if the database partitions. This is in contrast to the protocols in [9] and [14], where each read is required to access more than one copy, 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 [14] requires that read operations always physically access at least $f + 1$ copies. This is improved upon in [9] where these expensive read operations are necessary only when failures occur.

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 accessibility thresholds to $A_r[x] = 5$ and $A_w[x] = 4$ (thus satisfying threshold relation (1)), x is read and write accessible in P. The database designer has the following three possible choices for the read and write quorums of x in partition P:[1]

Quorums	Choice I	Choice II	Choice III
$q_r[x, v]$	1	2	3
$q_w[x, v]$	6	5	4

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

With the protocols in [9] and [14], to allow the writing of x in P one can set either $Q_w[x] = 6$, in which case a read x has to access 3 copies (and a write x writes 6 copies), or $Q_w[x] = 5$, in which case a read x has to access 4 copies (and a write x writes 5 copies). Note that with these protocols writing only four copies of x (as in our Choice III) is not allowed since four copies are not a majority of copies of x. With the protocol in [12], only Choice I is allowed. With the one in [11], only Choices I and II are allowed.

Note that whenever views change, our approach requires the execution of update transactions. With Gifford's protocol there are no update transactions, but, as illustrated above, user transactions can be more expensive than with our method. We assume that the network topology does not change often, and hence update transactions are rare with respect to user transactions. This justifies a higher cost for update transactions, and a lower cost for user transactions.

For multiversion databases, Herlihy [18] recently presented a generalization of Gifford's quorum protocol called the *quorum consensus protocol.* Each copy of an object has several versions ordered by *levels*. Each object has a *quorum assignment table* with a sequence of quorum assignments. Each quorum assignment corresponds to a level. For a transaction to execute, it chooses a level, and uses the quorum assignments associated with that level to execute operations. The quorum assignments are restricted to satisfy the *quorum intersection invariant*: "Each write quorum associated with level l must intersect with each read quorum associated with a level greater than or equal to l."

One of the main advantages of the quorum consensus protocol is that it adjusts "lazily" to changes in the network on a per object basis. As we saw in Sections 3.3 and 5.2, our protocol can achieve the same goals by using "on-demand" view tracking; that is, views are changed only when some "high-priority" transactions can no longer execute in the previous view, and associating views with copies

[1] This choice can be made dynamically by the initiator of the view v according to the current state and requirements of the database. The quorums chosen are then imposed on the other sites that inherit this view (by passing the chosen quorums via the update transaction executed by the initiator).

thus allows update transactions to be executed on a per object basis, only when necessary.

There are trade-offs between our protocol and the quorum consensus protocol in terms of costs. The quorum consensus protocol is designed for multiversion databases. It must maintain a quorum assignment table and ensure that this table always satisfies the quorum intersection invariant. This overhead allows transactions to run at increasingly higher levels (by a process called *inflation*) without incurring update costs. However, to satisfy the quorum intersection invariant, read quorums must monotonically increase with respect to level number, thus making read operations more expensive at higher levels. Hence, objects eventually need to reduce the read quorums assigned to their higher levels. For this purpose, objects execute a process called *deflation*, which is similar to our update transaction. One advantage of our protocol is that there is no need to maintain a quorum assignment table. Furthermore, when a site decides to install a new view (to execute transactions at a "higher level" in Herlihy's terminology [18]), it can freely choose the read and write quorums associated with the new view, without being restricted by an assignment that must satisfy the quorum intersection invariant for a given quorum assignment table.

7. CONCLUSION

In this paper we presented a new replica control protocol for reading and writing replicated data in spite of site and communication failures. The protocol uses views to ensure one-copy serializability as follows. First the protocol ensures that all transactions executing in each view are one-copy serializable (this is achieved by using intersecting read and write quorums). Then it ensures that all transactions executing in one view are serialized after all transactions executing in all views with lower view-ids; that is, transactions executing in different views are serialized according to view-ids (this is achieved by using the proper accessibility thresholds, and update transactions). In [10], we show that this method for serializing transactions in replicated databases is an instance of a class of protocols that are defined by a general paradigm. We show that [9], [14], and [18] are also instances of the paradigm, and prove that any instance of the paradigm ensures one-copy serializability.

The choice of accessibility thresholds, quorums, and views gives a large degree of flexibility in determining the availability of objects and the costs of executing read and write operations. First, one can choose the read and write accessibility thresholds for each object. These thresholds determine the read and write availability of each object in all views (depending on the number of copies in those views). To increase the read availability of an object, the read accessibility threshold is decreased, and vice versa. Second, for each new view installed during execution, one can choose a read and a write quorum for each object (that is read or write accessible in the new view). These quorums determine the costs of executing read and write operations for each object in each new view. Finally, the database designer may choose among several policies for deciding when to change views (following changes in the network topology), and which sites to include in the new views. These view-changing policies also determine the availability of different objects, and the cost of executing operations.

ACKNOWLEDGMENTS

We would like to thank Ken Birman, Maurice Herlihy, the three anonymous referees, and the students in the distributed systems group at Cornell for their helpful comments on earlier drafts of this paper.

REFERENCES

1. BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.
2. BERNSTEIN, P. A., AND GOODMAN, N. The failure and recovery problem for replicated databases. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, pp. 114–122.
3. BERNSTEIN, P. A., AND GOODMAN, N. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 596–615.
4. BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.
5. CHEUNG, D., AND KAMEDA, T. Optimal decentralized termination protocols for partition failures. LCCR TR 86-1, Laboratory for Computer and Communication Research, Simon Fraser Univ., Burnaby, B.C., Canada, Jan. 1986.
6. CHUNG-KUO, C., AND GOUDA, M. Independent recovery. In *Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems* (Williamsburg, Va., Mar. 1987). IEEE, New York, 1987, pp. 93–104.
7. COAN, B., OKI, B., AND KOLODNER, E. Limitations on database availability when networks partitions. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing* (Calgary, Alberta, Canada, Aug. 1986). ACM, New York, pp. 63–72.
8. DAVIDSON, S. B., GARCIA-MOLINA, H., AND SKEEN, D. Consistency in partitioned networks. *ACM Comput. Surv. 17*, 3 (Sept. 1985), 341–370.
9. EAGER, D., AND SEVCIK, K. Achieving robustness in distributed database systems. *ACM Trans. Database Syst. 8*, 3 (Sept. 1983), 354–381.
10. EL ABBADI, A. A paradigm for concurrency control protocols for distributed databases. Ph.D. thesis, TR 87-853, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y. (Aug. 1987).
11. EL ABBADI, A., AND TOUEG, S. Availability in partitioned replicated databases. In *Proceedings of the 5th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems* (Cambridge, Mass., Mar. 24–26, 1986). ACM, New York, pp. 240–251.
12. EL ABBADI, A., SKEEN, D., AND CRISTIAN, F. An efficient fault-tolerant protocol for replicated data management. In *Proceedings of the 4th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems* (Portland, Ore., Mar. 25–27, 1985). ACM, New York, pp. 215–229.
13. ESWARAN, K., GRAY, J., LORIE, R., AND TRAIGER, I. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
14. GIFFORD, D. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 10–12, 1979). ACM, New York 1979, pp. 150–159.
15. GOODMAN, N., SKEEN, D., CHAN, A., DAYAL, U., FOX, S., AND RIES, D. A recovery algorithm for a distributed database system. In *Proceedings of the 2nd ACM Symposium on Principles of Database Systems* (Atlanta, Ga., Mar. 21–23, 1983). ACM, New York, pp. 8–15.
16. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager in the System R database manager. *ACM Comput. Surv. 13*, 2 (June 1981), 223–242.
17. HADZILACOS, V. Issues of fault tolerance in concurrent computations. Tech. Rep. 11-84, Center for Research in Computing Technology, Harvard Univ., Cambridge, Mass. (June 1984).
18. HERLIHY, M. Dynamic quorum adjustments for partitioned data. *ACM Trans. Database Syst. 12*, 2 (June 1987), 170–194.
19. JOSEPH, T., AND BIRMAN, K. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Trans. Comput. Syst. 4*, 1 (Feb. 1986), 54–70.
20. PAPADIMITRIOU, C. H. Serializability of concurrent database updates. *J. ACM 24*, 4 (Oct. 1979), 631–653.

21. PERRY, K., AND TOUEG, S. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Softw. Eng. SE-12*, 3 (Mar. 1986), 477–482.
22. REED, D. P. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 3–23.
23. SKEEN, D. Crash recovery in a distributed database management system. Ph.D. thesis, Memo. UCB/ERL M82/45, Electronics Research Lab., Univ. California, Berkeley, 1982.
24. SKEEN, D., AND WRIGHT, D. Increasing availability in partitioned networks. In *Proceedings of the 3rd ACM SIGACT/SIGMOD Symposium on Principles of Database Systems* (Waterloo, Ontario, Canada, Apr. 2–4, 1984). ACM, New York, 1984, pp. 290–299.
25. WRIGHT, D. Managing distributed databases in partitioned networks. TR83-572, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Sept. 1983.

Received May 1986; revised August 1987; accepted November 1987