1989

Adaptable Recovery Using Dynamic Quorum Assignments

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Shirley Browne

Report Number:
89-886

ADAPTABLE RECOVERY USING DYNAMIC QUORUM ASSIGNMENTS

Bharat Bhargava
Shirley Browne

CSD-TR-886
June 1989

Adaptable Recovery Using Dynamic Quorum Assignments *

Bharat Bhargava and Shirley Browne
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract. This research investigates the problem of how to adapt the changing of quorum assignments for objects in a replicated database to the duration and extent of failures occurring in the underlying communication network. The concept of a view based on a connected component of the network is used to coordinate changes to quorum assignments of different objects. New view formation is used only when needed, and conditions under which quorum assignments may be changed without forming a new view are given. A dynamic view formation protocol is proposed that permits objects to join a new view on demand. A new technique called inheritance enables a new view to acquire quorum assignments from an old view, so that only those objects that were accessed during a failure need to change their quorum assignments back following repair of the failure. Extension of an existing view may be used to incorporate a recovering site into the network without forming a new view, thus localizing the effects of the failure. We have made analytical performance estimates for some sample network configurations and failure situations to show the improvements of our method over previously proposed methods. Following repair of a failure, our method can begin processing transactions almost immediately, but with less extra average overhead than for previous methods. We describe a prototype implementation of our method that will be used for future experimentation.

*This research is sponsored by National Science Foundation Grant IRI-8821398.

1 Introduction

The need for adaptability and reconfigurability in a distributed system has been discussed in [3], [4], and [7]. In [7] a model for adaptability in a distributed database system is proposed and applied to distributed concurrency control and commit protocols, to network partitioning control, and to server relocation. Data replication increases the opportunities for adapting to failures and changing conditions, but adds the problem of maintaining mutual consistency of the replicated copies. Maintaining mutual consistency involves both concurrency control and replication control protocols. The purpose of these protocols is to achieve one-copy serializablity – that is, to ensure that the concurrent execution of transactions on the replicated database has the same effect and appearance as a serial execution on a one-copy database [5].

The use of quorums to deal with site failures and network partitioning was proposed in [11]. A quorum assignment for a replicated object specifies how many or which copies must be accessed to carry out an operation. A quorum method may be either static or dynamic. With a static method, quorum assignments are fixed. A dynamic method allows quorum assignments to be changed in order to increase availability. Methods for coordinating changes to quorum assignments have been proposed in [2], [12], and [13]. Changes to quorum assignments are typically coordinated by means of views. A view of the database is essentially a set of sites that can communicate with each other, together with the copies of objects residing at those sites. Dynamic quorum methods are expensive in processing and communication overhead. The cost is incurred either all at once when a new view of the database is formed, or incrementally as database objects are accessed in the new view. In [2] and [12], all objects in a connected component of the network change their quorum assignments at the same time by means of a single reconfiguration transaction. The same reconfiguration protocol is always invoked, regardless of the extent of the failure. In [13], objects may change quorum assignments one at a time, but a cascade effect causes the quorum assignments for all objects to eventually be changed.

The problem addressed in our research is how to make dynamic quorum methods adaptable to the duration and extent of failures in order to reduce communication costs and overheads during recovery. If possible, we would like to have the cost of adapting to a failure be incurred only during the failure. The cost should be proportional to the severity of the failure, rather than to the size of the database.

Our approach to handling failures is to change the quorum assignment for an object only when that object is accessed by a transaction, as in [13]. By using a new technique called *inheritance*, however, quorum assignments that have not been changed during a network partitioning failure may remain in effect after the failure has been repaired. A new view is formed following repair, but this new view inherits quorum assignments from the old view that existed prior to the failure. Hence, the amount of work required following repair is proportional to the number of quorum assignment changes made during the failure, which depends on the length of the failure. Alternately, to handle repair of a lengthier failure, quorum assignments may be inherited by the new view from the current view for the largest partition of the network. In this way, the cost of repair is localized to objects that were accessed during the failure in smaller partitions. Objects that were accessed in the larger partition need only to extend their quorum assignments to include the copies being merged in. This extension may be carried out by means of an inexpensive *lightweight* protocol that may be integrated with transaction processing with little additional overhead.

To handle a recovery from a brief site failure, inheritance from the old view that existed prior to the site failure may be used. To recover from a lengthier site failure, we propose a new *view extension* technique followed by extension of quorum assignments to include copies at the recovered site. In addition to its usefulness for recovery, our lightweight method of changing quorum assignments may be used to tune performance, or to add or delete copies of an object during non-failure situations.

Because our adaptable dynamic quorum method can change quorum assignments on demand and can re-use unchanged assignments following repair, it adapts to the access pattern of transactions that run during the failure. By localizing the effects of a site failure or of the separation of a small number of sites, our method also adapts to the extent of the failure. Although we were unable to isolate the cost of a failure entirely to the failure period, our adaptability techniques limit the cost that is incurred followed repair.

In this paper, we describe our adaptable dynamic quorum method and briefly describe measurements from a prototype implementation based on a version of the RAID system [8]. These measurements are used to evaluate the estimated performance of our method analytically under various failure situations. A proof that our method ensures one-copy serializability of transaction processing is given in the Appendix. The proof shows that by exchanging appropriate information between sites, the amount of which depends on the length or extent of the failure being repaired, view formation with inheritance satisfies the conditions for one-copy serializability in order of view ids. We also show that our algorithms for lightweight change of quorum assignments and for extending a view preserve one-copy serializability with a view.

2 Quorum Model and Terminology

A set of copies that suffices to carry out an operation is called a *quorum.* With an appropriate quorum intersection requirement for conflicting operations, the use of quorums, together with distributed concurrency control and atomic commitment protocols, ensures the consistency of replicated data as seen by user transactions. The possible quorums that may be used are listed in a *quorum assignment.* We assume that an object's quorum assignments are stored with each copy of the object. Two types of quorum assignments may be maintained for an object:

1. The *active* quorum assignment is read to determine what copies of the object must be accessed to carry out an operation.

2. The *backup* quorum assignment is used following a site or partitioning failure that renders active quorums unavailable to determine what the new active quorum assignment should be.

A *coquorum* for a set S of quorums in a quorum assignment is a set of copies that intersects every quorum in S.

Quorum intersection rules that must be followed by quorum assignments for a dynamic quorum method have been given in [12, 13] for abstract data types. These rules are stated in the read-write model as follows:

1. An active write quorum must intersect all active read quorums (i.e., an active write quorum must be an active read coquorum).

2. A backup write quorum must intersect all backup read quorums.

3. An active write quorum must intersect all backup read quorums.

Using both types of quorums yields better availability during failures without sacrificing performance in the absence of failures. An object may change its active quorum assignment even when currently active quorums are unavailable, provided backup quorums are available.

To achieve one-copy serializability, objects coordinate changes to their active quorum assignments by means of views. We use a somewhat different notion of a view from that in [9, 2, 12]. In our model, a *view* is a set of copies of objects that agree to use a particular set of quorum assignments that satisfy the quorum assignment rules. Each view has a unique integer view id. The view formation protocol given in section 3 ensures that the view ids for views in which a given copy of an object participates are increasing over time. A view is based on a connected component of the network, that is, a set of sites that can communicate with each other. A view is associated with a *connection vector*, indexed by the sites in the system, with a 1 entry for each site in the view's underlying component and a 0 entry for the other sites. In our model, a view is loosely tied to its corresponding connection vector, however, and the connection vector may be changed without changing the view id.

Whenever a dynamic object is added to the system, either at system startup time or during system operation, a backup quorum assignment must be specified that satisfies quorum intersection rule 2. The active quorum assignment is specified when the object joins a view. An object may join a view with read access if it has a backup read quorum in the view's component. Likewise, it may join with write access if it has a backup write quorum in the component. A copy of an object participates in at most one view. If an object is read- (write-) accessible in a view, then it clearly cannot simultaneously be write- (read-) accessible in any other disjoint view, since a backup read (write) quorum must intersect all backup write (read) quorums.

A transaction must execute entirely within a single view. The quorum intersection rules guarantee that transactions executed in the same view are serializable among themselves. Our proof of correctness, given in the Appendix, uses the group paradigm [1] to show that transactions executed in different views are serializable in order of their view identifiers.

3 Adaptability Techniques

Our dynamic quorum method has the following five components, with the last two specifically designed for adaptability:

- a view formation protocol,
- a protocol for changing quorum assignments that can be used when currently active quorums are unavailable but involves a move to a new view,
- a lightweight protocol for changing quorum assignments that requires a read and a write quorum to be available but does not require new view formation,
- an inheritance mechanism that permits a new view to acquire objects from an old view without changing their quorum assignments,
- a view extension protocol that changes the connection vector for a view without changing the view id.

Our adaptability techniques are an example of state conversion adaptability, as described in [7]. Adaptability in our case is data driven, because only the data structures are changed, with the same transaction processing algorithms used after the conversion as before. More general adaptability techniques that also involve changing to different algorithms are discussed in [7].

Recovery actions that do not require formation of a new view, such as the lightweight and view extension protocols, are cheaper than those that form a new view with a new view id. There are two reasons for the difference in cost. The first is that formation of a new view requires participation of all sites in the new view, whereas a quorum assignment change that does not require a move to a new view need involve only those sites having copies of the object. The second reason is that changing the view id is likely to cause transactions that are in progress concurrently to abort and restart in the new view, because a transaction must execute entirely with a single view.

We assume that the distributed database system runs a correct distributed conflict-preserving concurrency control protocol, as well as conventional disk-based crash recovery algorithms [5]. We also assume the use of transaction commit and termination protocols, as described in [17].

A quorum assignment change is always carried out by means of a two-phase commit protocol. A new version number is issued by the transaction manager coordinating the change and is stored with the quorum assignment. When a transaction sends an operation request to a member of a quorum, it includes the version number it read for the quorum assignment. If the quorum member has a more recent version of the quorum assignment, it informs the transaction coordinator of the new quorum assignment. As long as at least one member of each old quorum has been notified of a quorum assignment change, comparing the versions in this way will ensure that an out-of-date quorum assignment is not used.

3.1 New View Formation

New view formation may be invoked by a transaction manager when failures prevent the use of the quorum assignments associated with the current view, or when repair of a failure makes more copies available for inclusion in active quorum assignments. Following formation of a new view, which is initially empty of any objects, objects are moved to the new view on demand as they are accessed by transactions. Transactions will typically attempt to execute in the most recent view known to have been formed.

To request that a new view be formed, a transaction manager sends a view formation request to all sites in the new view's component. The request contains the connection vector for the new view. In addition to the connection vectors associated with view, we assume a connection vector is maintained at each site in the form of a hint by the communications subsystem. When a failure occurs, transactions executing at different sites may concurrently request a new view. This contention may be handled by allowing a request from a lower-numbered site to pre-empt a request from a higher-numbered site. A site replies to a request by returning the highest view id it has seen so far. The view formation coordinator chooses a unique new view id greater than any of those received and sends it out in a commit message. Logging the new view id to stable storage is not required. If the view formation is interrupted by a failure, then any site may initiate still another new view formation, regardless of whether the site is able to determine the outcome of the previous attempt.

A separate view formation protocol is not actually required for correctness, as a new view could be formed incrementally in conjunction with moving objects to the new view. Once the network topology stabilizes, the view ids in a connected component of the network would converge to a common value. We include a view formation protocol so that the common view id will be achieved sooner. Because our view formation protocol does not reconfigure any quorum assignments or involve writes to stable storage, it is relatively inexpensive. We conjecture that using a separate view formation protocol will help cut down on the number of transaction aborts that occur following repair of multiple failures.

3.2 Moving an Object to a New View

After a new view has been formed, it will not at first contain any objects. Hence, when a transaction attempts to execute in a newly formed view, it will find that the objects it wants to access are not present in the view. If an object that the transaction wants to access is accessible in the view's component (i.e., the object has a backup quorum in the component), the transaction can attempt to move the object into the new view with a new active quorum assignment. If the object is both read- and write-accessible, then the new active quorum assignment can be any assignment that satisfies the quorum intersection rules given in section 2. If the object is only write-accessible, then the new active assignment is set equal to the backup assignment. If the object is only read-accessible, then the active assignment may instead be set equal to the most recent active assignment, because no other disjoint view can have write access. To move an object into a new view, a transaction manager carries out the following steps:

1. Send view change request messages with the new view id and new active quorum assignment to all copies of the object in the new view. If the object is read accessible in the new view, include a read request for the object to members of a backup read quorum. Each copy checks that its view id is less than the new view id. If so, the copy write locks its quorum assignment, logs the proposed change, and, if requested to do so as a member of a backup read quorum, returns information about the object. Otherwise, if its view id is greater or equal, the copy returns the newer view id and the corresponding active quorum assignment. If some copy returns a newer view id, the view change is aborted. Otherwise, proceed to step 2.

2. If the object is not read-accessible in the new view, go to step 3. Otherwise, send enough information about the object, obtained from a backup read quorum,to a new active write quorum so as to ensure that all members of the new write quorum are up-to-date After replies are received from members of the new write quorum indicating that they have logged any new events or values, proceed to step 3.

3. Send a commit message to all copies in the new view. Upon receiving a commit message, a copy marks itself deleted in the old view (it may retain read access there, however, as long as the old version is kept), adds itself to the new view with the new active quorum assignment, and releases the write lock on the quorum assignment.

Logging is required in step 2 for the same reason that it is required in phase one of a two-phase commit protocol, namely that a site must must be able to redo the updates of a committed·transaction following a failure. Logging the new view id and quorum assignment in step 1 ensures that transactions from different views will be serializable in order of their view ids even if site failures occur. If the view change transaction is interrupted by a failure, then any site may initiate a new view change for the object, even if the site is unable to determine the outcome of the previous attempt.

3.3 Lightweight Quorum Assignment Change

If both a read quorum and a write quorum for an object are available, then its quorum assignment may be changed without forming a new view. Note that a read quorum is a write coquorum (i.e., a read quorum intersects every write quorum) and a write quorum is a read coquorum. Hence, notifying both a read quorum and a write quorum of the change ensures that at least one member of every old quorum has been notified. This rule applies equally well to both active and backup quorum assignments. Either type of quorum assignment may be changed without changing the other, provided the resulting active and backup quorum assignments satisfy the quorum intersection rules given in section 2.

To change a quorum assignment for an object, a transaction manager carries out the following steps:

1. Send a request for a quorum assignment change containing the old quorum assignment version number to all available old quorum members and to all new quorum members.

If the change is for an active quorum assignment, include a read request for the object to members of an old read quorum.

2. Upon receiving a request for quorum assignment change, a quorum member write locks the quorum assignment and logs the new assignment to stable storage. It also checks if its quorum assignment version number agrees with the one in the request. If its own quorum assignment is more recent, it sends back its own quorum assignment. If requested to do so as a member of a read quorum, it sends back information about the object. After replies are received from all new quorum members and from at least one old read quorum and one old write quorum, proceed to step 3. (If a more recent quorum assignment was received from some site, restart the protocol with step 1).

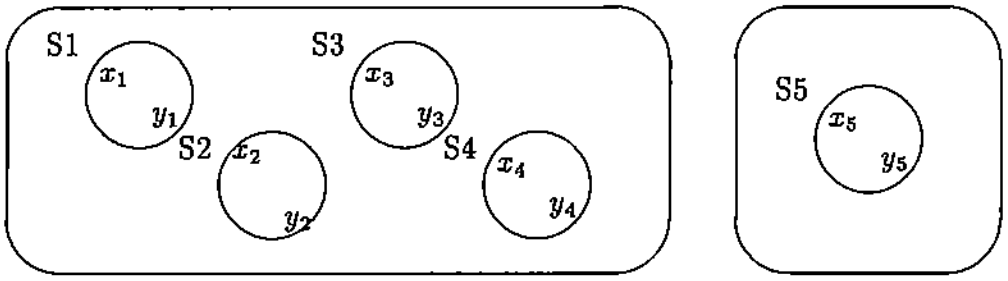
3. If the change is only for a backup quorum assignment, proceed to step 4. Otherwise (the active quorum assignment is being changed), send enough information about the object (obtained from the old read quorum) to a new write quorum to ensure that all members of the new write quorum are up-to-date. After replies are received from members of the new write quorum indicating they have logged any new values or events, proceed to step 4.

4. Send a commit message to all new quorum members and to all available old quorum members. Upon receiving a commit message, a quorum member updates the quorum assignment and releases the write lock on the quorum assignment.

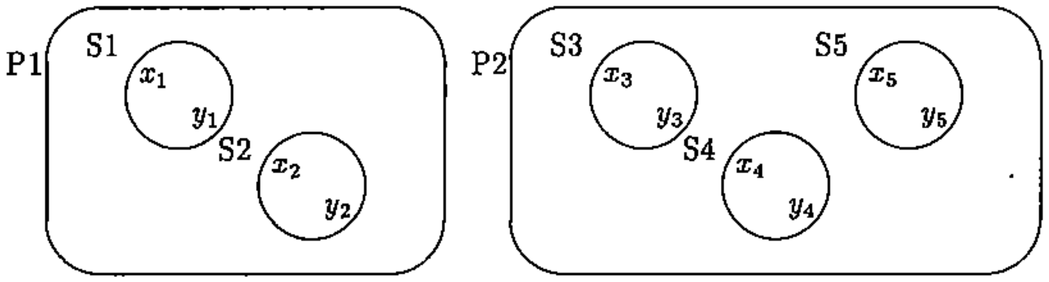
Logging is required in step 3 because a site must be able to redo the updates of a committed transaction following a failure. Logging of the new quorum assignment is required in step 2 to ensure that knowledge of the quorum assignment change at sites in the old quorums survives site failures. The reason logging of the new quorum assignment is required in step 2 is illustrated by Example 3.1.

Example 3.1 Suppose sites S1, S2, S3, S4, and S5 all have copies of both X and Y. Let the backup quorum assignment for X be read-three/write-three, and the backup assignment for Y be read-four/write-two. Let S5 be partitioned from the other sites as shown below. A lightweight quorum assignment change transaction coordinated at S1 is attempting to change the active assignment for

X from read-three/write-three to read-two/write-four and the active assignment for Y from read-three/write-three to read-four/write-two.



Sites S3 and S4 fail following the change and come back up connected to S5 but partitioned from S1 and S2. Transaction T_1 coordinated at S1 attempts to run in P1, and T_2 coordinated at S5 attempts to run in P2.



$T_1 :\ r[X]\ w[Y]$ $\qquad\qquad$ $T_2 :\ r[Y]\ w[X]$

If the new active assignments are used in P1, but the old assignments are used in P2, the incorrect execution will be allowed. If the recovery of S3 and S4 forces a new view formation, however, Y will not be read-accessible in P2 and T_2 will not be able to execute.

Example 3.1 shows why an old quorum member must log the change to stable storage in phase 1 if site recoveries need not force new view formation. Suppose sites S3 and S4 fail during their periods of uncertainty (i.e., they have replied to the request but have not received the commit message) and have not logged the proposed change to stable storage. If the recoveries of S3 and S4 do not force a new view to be formed, T_2 will be able to execute using the old assignments. T_1 will be able to execute in P1 using the new assignments. It is undesirable to force new view formation every time a site recovers. Requiring a new view to be formed would result in greater communication overhead if a site failure is of short enough duration that a new view excluding the failed site has not been formed. Moreover, it would rule out the view extension technique described in section 3.5 that can be used to make recovery from long site failures more efficient.

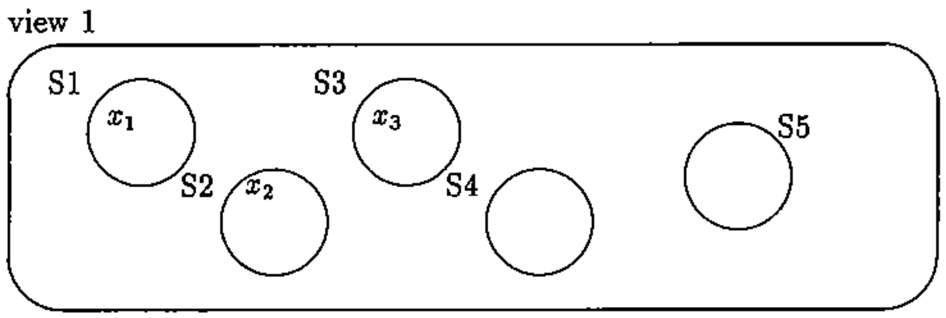
3.4 Inheritance of Quorum Assignments by Views

Inheritance essentially permits re-use of an old view that is still largely intact. When coordinating the formation of a new view, a site may check whether some old view has the same connection vector as the new one. If so, and if a large number of dynamic objects still reside in the old view, it may choose to have the new view inherit these objects from the old view. If it decides on inheritance, it sends the view id for the old view with the view formation request. Upon receiving the request, each site returns the names of any objects that it has deleted from the old view. The coordinator sends the names of all deleted objects with the commit message. Upon receiving the commit message, each site checks that all named objects have been deleted and switches the view id of the old view to that of the new one. If each object stores a pointer to the location of the view id, rather than the view id itself, then the view id for all the objects can be changed with a single write operation. Alternatively, the site can change the view ids of all the objects.

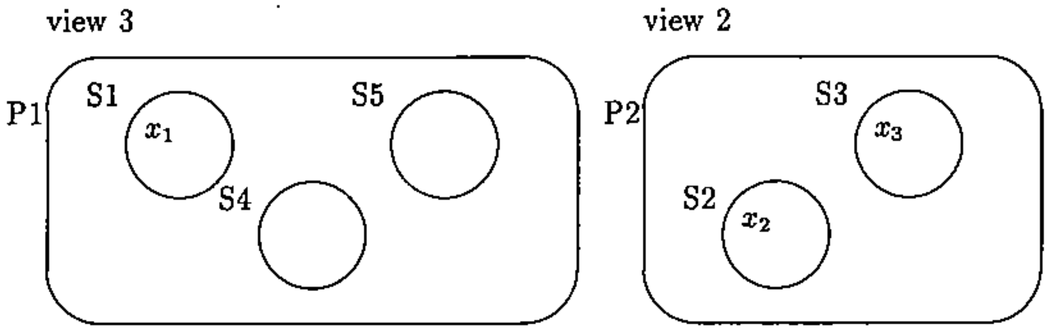
The idea behind our proof of correctness for inheritance is that at least one member of any old quorum must know about the deletion of an object. This knowledge prevents an old quorum assignment from being used for a deleted object. Propagating the names of deleted objects to all sites in the new view ensures that this condition holds for any object that was accessible in the old view.

It is not essential that the new connection vector be exactly the same as the new one. If the sites in the new view's component are a superset of those in the old one, then inheritance may still be used. Quorum assignments can then be extended to the new sites using the lightweight quorum change protocol. Thus, inheritance may be used to merge a small partition into the main network. If there are sites in the connection vector for the old view that are not in the new one, however, then using inheritance could result in transactions not being serializable in order of their view ids, as illustrated by Example 3.2. Transactions should be serializable in the order of their view ids because our proof of correctness depends on this fact.

Example 3.2 Suppose five sites have object X replicated as shown, with an active quorum assignment for X of read-any/write-all and a backup quorum assignment for X of read-two/write-two.



Suppose a partitioning occurs that separates S1, S4, and S5 in partition P1 that forms view 3 from S2 and S3 in partition P2 that forms view 2, and that transaction T_a attempts to run in view 3 and transactions T_b in view 2.



$T_a : \ldots r[X] \ldots$ $\qquad\qquad$ $T_b : \ldots w[X] \ldots$

If view 3 is allowed to inherit the quorum assignment for X from view 1, and if T_b moves X to view 2, then both transactions will be able to execute. T_a must be serialized before T_b, since T_a does not read the value written by T_b, but this order is opposite of the order of the view ids.

The problem in Example 3.2 is caused because the view for a proper subset of the sites in view 1 is allowed to inherit from view 1. Although X is deleted from view 1 by T_b, no site in the component for view 3 knows about this deletion. Such a problem will not occur if inheritance by a proper subset is disallowed.

Note that the serializability problem cannot be prevented by allowing inheritance from a proper subset and having the coordinating site run through its list of objects and backup quorum assignments for those objects and verify that all are accessible in the new view. With partial replication, the coordinating site will not necessarily have copies of all the objects. Furthermore, backup quorum assignments may have been changed and to find out about such changes, a backup quorum must be accessed for each object. Checking for changes to backup quorum assignments would thus involve additional communication and would make the cost of inheritance dependent on the size of the database rather than on the number of deleted objects. Hence, we disallow inheritance from a proper subset.

3.5 Extending a View

View extension can be used to avoid forming a new view when a site recovers from a failure. If a recovering site has been excluded from the view currently in use by the operational sites and it is desired to include copies of objects at that site in read and write quorums, a new view can be formed that includes the recovered site. Unless the new view is able to use inheritance, however, its formation will require all objects subsequently accessed to move to the new view, including objects that do not have copies at the recovered site. An alternative to forming a new view is to extend the connection vector for the current view to include the recovered site. Then the lightweight protocol described in section 3.3 may be used to change the quorum assignments for those objects that have a copy at the recovered site.

To extend the current view, a recovering site carries out the following steps:

1. Copy the connection vector and view id for the current view from an operational site.

2. Send a join view request containing the view id to all sites in the view. If the current view id at a site receiving a request message agrees with the view id in the request, the site modifies its connection vector for the view and sends a positive acknowledgment. Otherwise, it returns the newer view id and corresponding connection vector.

3. If no site returns a newer view id, the view extension is completed. Otherwise, repeat step 2 using the newest view id received.

Because correctness of our method does not depend on the accuracy of a view's connection vector, the connection vector may be treated as a hint. Rather than using a separate protocol, we could even propagate changes in the connection vector along with lightweight quorum change requests. The one-phase protocol is not expensive, however, and should cut down on the disparity between the connection vectors at different sites. Using a similar protocol to delete one or more sites from a view's connection vector will be less useful, however, since quorum assignments will most likely include the deleted sites. When active quorums cannot be obtained because of the failure, a new view will need to be formed anyway. Furthermore, if inheritance is used for subsequent new view formation, then deleting one or more sites from a view's component can lead to execution histories that are not serializable in order of the view ids of the transactions. If we allow sites to be deleted from a view's connection vector, then the deleted site(s) may be the only one(s) that know about the deletion of some objects from the old view. Hence, a new view could incorrectly inherit the deleted objects.

4 Analytical Performance Estimates

In this section, we evaluate different ways of handling new view formation and/or view extension following failures and subsequent repairs. We investigate how the relative performance of different techniques depends on the length of time between failure and repair. We have claimed that our techniques allow a replicated database system to adapt to the duration of a failure. In particular, we hypothesize that inheritance from the old view that existed prior to a failure will give good results if the failure is of short duration. For a longer failure, we expect that inheritance from the view for the larger component of the partitioned network (in the case of network partitioning) or extension of the current view (in the case of site recovery) will be better. We wished to determine whether a clear cut crossover point for the preferred method exists. If such a crossover point exists, the recovery mechanism may be able to determine whether or not this point has been reached and make decisions based on this knowledge.

For our analytical model, we have made the following assumptions:

1. Copies of relations are distributed uniformly among the sites.

2. The access pattern of relations by transactions has a random uniform distribution. This assumption makes the analytical model tractable.

3. Each transaction reads and writes a single relation out of a database of 100 relations.

4. Moves to a new view and lightweight quorum assignment changes are integrated with transaction processing.

5. Transaction throughput is limited by the rate at which servers can handle requests, rather than by network bandwidth. This assumption is based on the fact that high-bandwidth networks are becoming readily available.

Our analytical model has the following parameters (Let T_{Norm} denote a normal transaction, T_{NV} a transaction that moves an object to a new view, and T_{LW} a transaction that carries out a lightweight quorum assignment change):

NVtoNormResp – the ratio of the response time for T_{NV} to the response time for T_{Norm}.

LWtoNormResp – the ratio of the response time for T_{LW} to the response time for T_{Norm}.

NVtoNormLoad – the ratio of the load on the Transaction Manager servers for T_{NV} to the load for T_{Norm}.

LWtoNormLoad – the ratio of the load on the Transaction Manager servers for T_{LW} to the load for T_{Norm}.

Estimated values for these parameters for our prototype implementation are given in section 5.

Our calculations are based on the solution of a system of differential equations that describe how various quantities in the system change over time. These equations represent a continuous approximation to the discrete system. Experimental results in [6] give credence to a similar differential equation model for analysis of the fail-lock technique for doing site recovery. One of the experiments studied the rate at which fail-locks for out-of-date copies are cleared by transaction processing. The experimental data reported in [6] closely matches the solution of a differential equation that relates the rate at which fail-locks are cleared to the percentage of data items fail-locked.

The quantities solved for are the following:

move(t) – the number of relations remaining to be moved to a new view at time t following new view formation.

ltwt(t) – the number of relations still needing lightweight quorum assignment change at time t following new view formation or view extension.

Resp(t) – the ratio of average response time at time t following new view formation to normal average response time.

Th(t) – the ratio of throughput at time t following new view formation to normal throughput.

The quantities $Resp(t)$ and $Th(t)$ depend on one or both of $move(t)$ and $ltwt(t)$, depending on the type of view formation used. The equations and projected performance for different cases of failure and recovery are detailed in the following subsections.

4.1 Network partitioning failure and repair

Following a simple network partitioning, the network is divided into two components, and a new view has been formed for each component. If new view formation with reconfiguration has been used, all quorum assignments have been re-adjusted by the view formation transaction. If empty new view formation has been used, quorum assignments will be adjusted incrementally as relations are moved to the new view on demand by the transactions that access them.

Following empty view formation, the quantity $move_0(t)$ is the number of relations that remain to be moved to the new view at time t following the partitioning. Transactions will be committing at a rate $\lambda(t)$ which depends on the value of $move_0(t)$, since the additional overhead of moving relations to the new view will slow down the transaction processing rate. Given our assumption that relations are equally likely to be accessed, the transaction processing rate should be scaled by the fraction of relations that remain to be moved to obtain the rate of change of $move_0(t)$. Thus, a continuous approximation to the integer-valued quantity $move_0(t)$ is given by the solution to the following differential equation:

$$\frac{d\ move_0(t)}{dt} = \frac{-\lambda(t) move_0(t)}{dbsize} \tag{1}$$

where

$$\lambda(t) = Norm\lambda * Th(t)$$

is the transaction processing rate of the system at time t and

$$Th_0(t) = \frac{1}{(1 - \frac{move_0(t)}{dbsize}) + NVtoNormLoad * \frac{move_0(t)}{dbsize}} \tag{2}$$

is the ratio of throughput at time t to normal throughput. The initial condition for equation 1 is

$$move_0(0) = dbsize.$$

The ratio of average response time at time t following new view formation to normal response time is given by the following equation:

$$Resp_0(t) = (1 - \frac{move_0(t)}{dbsize}) + NVtoNormResp * \frac{move_0(t)}{dbsize}. \tag{3}$$

Although response time and throughput are normal following view formation with reconfiguration, successful transaction processing cannot resume until after the time required for reconfiguration. The time required for empty view formation is insignificant, and transaction processing can resume almost immediately following the failure. Then as relations are moved to the new view, response time and throughput gradually return to normal.

Repair of network partitioning. We compared the following ways of handling the repair of a network partitioning:

1. new view formation with reconfiguration,
2. empty new view formation,
3. view formation with inheritance from the old view that existed prior to the partitioning,

4. view formation with inheritance from the current view for the larger component.

For methods 3. and 4., we assumed that empty view formation was used following the network partitioning failure. Following 3., relations that were deleted from the old view while the network was partitioned need to be moved back. The number of such relations increases with the duration of the partitioning. Following 4., relations that have not yet been moved to the view for the larger component still need to be moved. Relations that have been moved to the view for that component but that have copies in the smaller component need to have their quorum assignments extended using lightweight quorum assignment change.

In the analysis that follows, we denote the average response time ratio for method i at time t by $Resp_i(t)$ and the throughput ratio for method i at time t by $Th_i(t)$. We let t_r denote the time of the repair.

4.1.1 New view formation with reconfiguration

$Resp_1(t)$ and $Th_1(t)$ for $t \geq t_r$ will both be 1.

4.1.2 Empty new view formation

$Resp_2(t)$ and $Th_2(t)$, for $t \geq t_r$, will be given by equations (3) and (2), respectively, with zero subscripts replaced by two's, but with the initial condition for equation (1) given by

$$move_2(t_r) = dbsize.$$

4.1.3 Inheritance from the old view

The throughput and response time ratios at time t, for $t \geq t_r$, are calculated as follows:

$$Th_3(t) = \frac{1}{\frac{1-move_3(t)}{dbsize} + NVtoNormLoad * \frac{move_3(t)}{dbsize})} \tag{4}$$

$$Resp_3(t) = (1 - \frac{move_3(t)}{dbsize}) + NVtoNormResp * \frac{move_3(t)}{dbsize}. \tag{5}$$

where $move_3(t)$ is the number of relations remaining to be moved to the new view. This quantity is initially equal to the number of relations that have been moved out of the old view by the time repair occurs. Hence, $move_3(t)$, for $t \geq t_r$, is given by the following differential equation:

$$\frac{d\ move_3(t)}{dt} = \frac{-\lambda(t)\ move_3(t)}{dbsize} \tag{6}$$

with the initial condition

$$move_3(t_r) = deleted(t_r) = dbsize - move_0(t_r).$$

4.1.4 Inheritance from the current view for the larger component

The throughput and response time ratios at time t following inheritance from the current view for the larger component are calculated as follows:

$$Th_4(t) = \frac{1}{(1 - \frac{move_4(t)+ltwt_4(t)}{dbsize}) + LWtoNormLoad * \frac{ltwt_4(t)}{dbsize} + NVtoNormLoad * \frac{move_4(t)}{dbsize}} \tag{7}$$

$$Resp_4(t) = (1-\frac{move_4(t) + ltwt_4(t)}{dbsize})+LWtoNormLoad*\frac{ltwt_4(t)}{dbsize}+NVtoNormLoad*\frac{move_4(t)}{dbsize} \tag{8}$$

where $move_4(t)$ is the number of relations remaining to be moved to the new view and $ltwt_4(t)$ is the number of relations still needing lightweight quorum assignment change at time t. The differential equations and initial conditions for $move_4(t)$ and $ltwt_4(t)$ for $t \geq t_r$, are as follows:

$$\frac{d\ move_4(t)}{dt} = \frac{-\lambda(t)\ move_4(t)}{dbsize} \tag{9}$$

$$\frac{d\ ltwt_4(t)}{dt} = \frac{-\lambda(t)\ ltwt_4(t)}{dbsize} \tag{10}$$

Based on the assumption of a uniform distribution of relations among the sites, we calculated values for the initial conditions $move_4(t_r)$ and $ltwt_4(t_r)$. Relations needing to be moved to the new view at time t_r (i.e., $move_4(t_r)$) are those that were not accessed in the larger component of the partitioned network (i.e., that were write-accessible in the smaller component or that were accessible in the larger component but had not yet been moved to the view for that component). Relations requiring lightweight quorum assignment change at time t_r (i.e., $ltwt_4(t_r)$) are those that were accessed in the larger component during the partitioning and have copies at sites in the smaller component. Details of how $move_4(t_r)$ and $ltwt_4(t_r)$ were calculated may be found in [10].

4.1.5 Comparison of the different methods

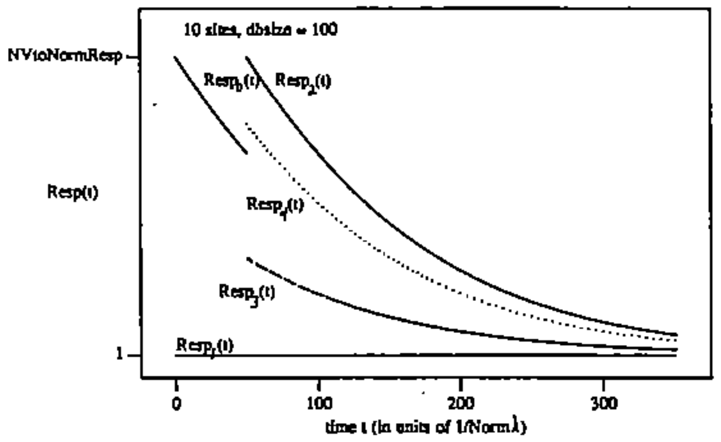
We solved the above equations numerically with $dbsize = 100$ and $nsites = 10$ to obtain the curves in Figures 1 and 2 that show the calculated average response time and throughput ratios following the different methods of handling repair of the partitioning for different durations of the partitioning. For the 10-site case shown, the sizes of the two partitioned components were 4 and 6 sites. We used values for $NVtoNormResp$, $LWtoNormResp$, $NVtoNormLoad$, and $LWtoNormLoad$ as suggested by the preliminary experimental data reported in section 5.

The curves for view formation with reconfiguration (labeled $Resp_1(t)$ in Figure 1 and $Th_1(t)$ in Figure 2) illustrate the performance achieved by the methods in [2] and [12]. The curves for empty view formation (labeled $Resp_2(t)$ in Figure 1 and $Th_2(t)$ in Figure 2) illustrate the performance achieved by the method in [13]. The curves for both of our inheritance methods show improvement on the performance of method 2. The curves for our inheritance methods indicate crossover points. If repair occurs before the crossover point, transaction processing performance following repair is better if the repair is handled by using inheritance from the old view. If repair occurs after this point, performance is better if inheritance is from the current view for the larger component. Although the crossover points do not necessarily occur at the same place for response time and throughput, they will be close, since the two measures depend on the same quantities in similar ways. For different values of $dbsize$, or of the number of sites, or of the parameters $NVtoNormResp$, $LWtoNormResp$, $NVtoNormLoad$, and $LWtoNormLoad$, we obtain curves with the same basic shapes.

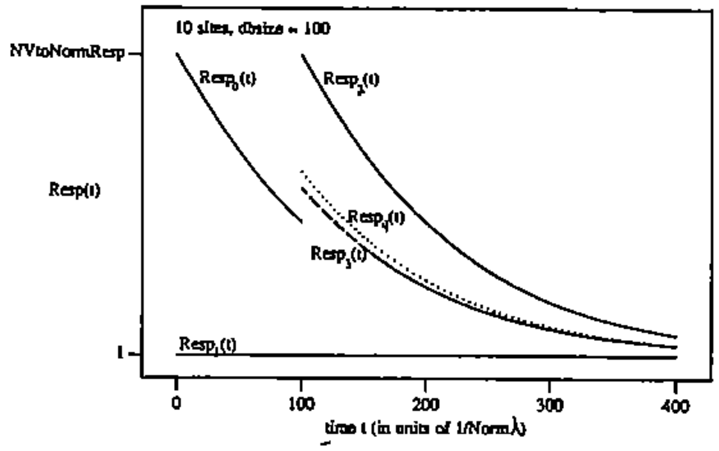
4.2 Site recovery

We compared the following alternatives for handling the site recovery process:

LEGEND:	
$Norm\lambda$	normal transaction processing rate
	Response time ratios
$Resp_0(t)$	following network partitioning and empty new view formation
$Resp_1(t)$	with complete reconfiguration
$Resp_2(t)$	following repair of partitioning and empty new view formation
$Resp_3(t)$	following repair of partitioning and inheritance from pre-failure view
$Resp_4(t)$	following repair of partitioning and inheritance from larger [illegible] failure view



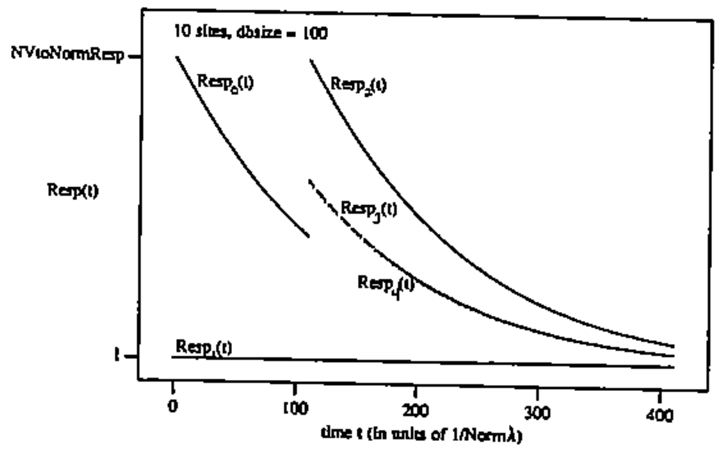
(a) Repair after 50*(1/Normλ) seconds



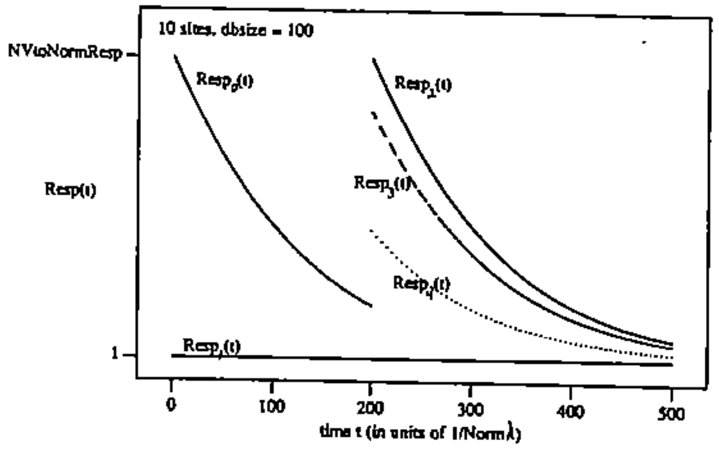
(b) Repair after 100*(1/Normλ) seconds

Figure 1 Comparison of response time for different methods of handling repair of network partitioning

LEGEND:	
$Norm\lambda$	normal transaction processing rate
	Response time ratios
$Resp_0(t)$	following network partitioning and empty new view formation
$Resp_1(t)$	with complete reconfiguration
$Resp_2(t)$	following repair of partitioning and empty new view formation
$Resp_3(t)$	following repair of partitioning and inheritance from pre-failure view
$Resp_4(t)$	following repair of partitioning and inheritance from larger failure view



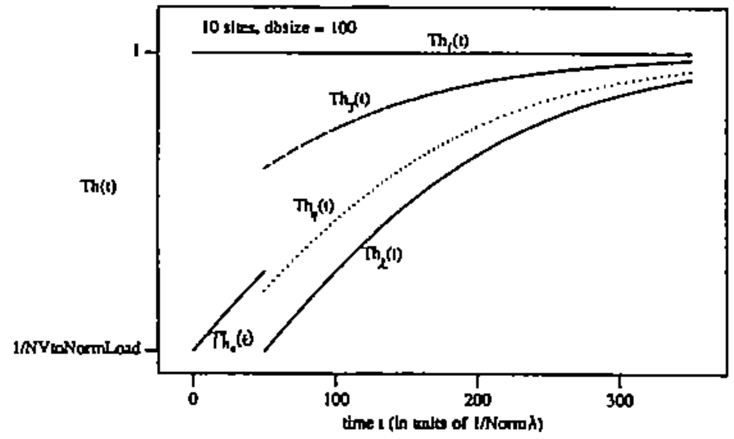
(c) Repair after 110*(1/Normλ) seconds



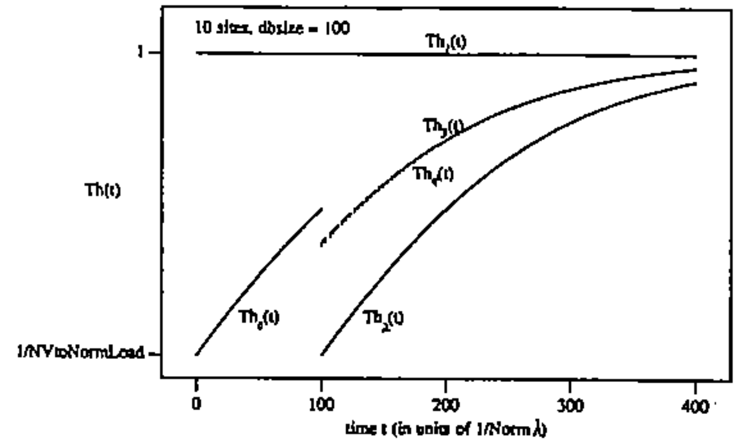
(d) Repair after 200*(1/Normλ) seconds

Figure 1 continued

LEGEND:	
$Norm\lambda$	normal transaction processing rate
	Throughput ratios
$Th_0(t)$	following network partitioning and empty new view formation
$Th_1(t)$	with complete reconfiguration
$Th_2(t)$	following repair of partitioning and empty new view formation
$Th_3(t)$	following repair of partitioning and inheritance from pre-failure view
$Th_4(t)$	following repair of partitioning and inheritance from larger failure view



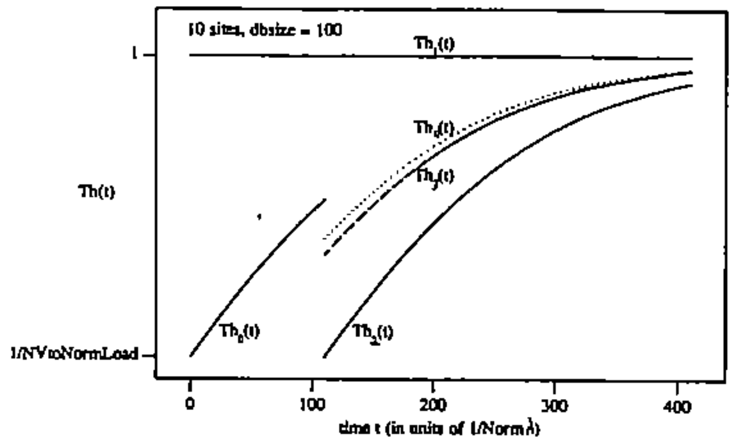
(a) Repair after 50*(1/Normλ) seconds



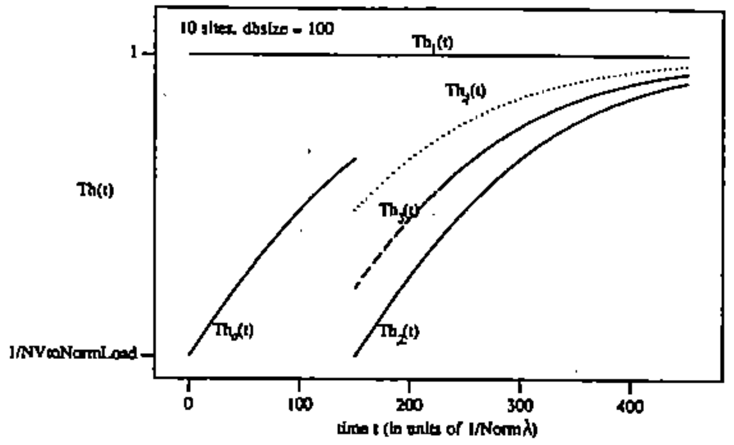
(b) Repair after 100*(1/Normλ) seconds

Figure 2 Comparison of throughput for different methods of handling repair of network partitioning

LEGEND:	
$Norm\lambda$	normal transaction processing rate
	Throughput ratios
$Th_0(t)$	following network partitioning and empty new view formation
$Th_1(t)$	with complete reconfiguration
$Th_2(t)$	following repair of partitioning and empty new view formation
$Th_3(t)$	following repair of partitioning and inheritance from pre-failure view
$Th_4(t)$	following repair of partitioning and inheritance from larger -failure view



(c) Repair after 110*(1/Normλ) seconds



(d) Repair after 150*(1/Normλ) seconds

Figure 2 continued

1. new view formation with reconfiguration,
2. empty new view formation,
3. view formation with inheritance from the old view that existed prior to the site failure,
4. extension of the current view to include the recovering site.

As an alternative to method 4., inheritance from the current view would also be possible, but view extension is cheaper since fewer messages are sent. View extension also has the advantage that transactions in progress concurrently will not be aborted because of a change in view id. Hence, extending the current view seems clearly preferable to inheriting from it.

Using an analysis similar to that in section 4.1, we observed the same relationship between the methods for site recovery as for the repair of network partitioning. Details of this analysis may be found in [10]. The curves for methods 3. and 4. again showed crossover points. If recovery occurs before the crossover point, subsequent transaction processing performance is better if recovery is handled using inheritance from the old view. If recovery occurs after this point, performance is better if recovery is handled using view extension.

5 Prototype Implementation

As a first step in studying the behavior of our proposed algorithms, we have implemented our quorum-based transaction processing and view formation protocols on a local-area network of Sun workstations. To carry out the implementation of our dynamic quorum methods, we have modified and extended the RAID distributed database system, described in [8]. Modifications needed include support for partial replication and the use of views and quorums for replication control. Since RAID is designed in a modular fashion, we were able to concentrate most of our efforts in the Replication Controller module, while leaving the other servers relatively unchanged.

Transactions and view formations/extensions were run on five Sun 3/50's connected by a ten megabit/second Ethernet for configurations with three sites, five sites, and ten sites. The degree of data replication was set to two, three, and six, respectively (i.e., to approximately 0.6 times the number of sites). Active quorum assignments were set to read-one write-all (in

	T_{Norm}	T_{NV}	T_{LW}
3 sites	.48	.54	.64
5 sites	.51	.64	.57
10 sites	.69	.92	.74

Table 1: Execution time in seconds for read-write transactions

	Empty view formation	View extension
3 sites	.06	.06
5 sites	.10	.10
10 sites	.23	.20

Table 2: Execution time in seconds for view formation and extension

the current view) and backup quorum assignments were set to read-majority write-majority. Each user transaction accessed a single relation consisting of 100 tuples of length 50 bytes. A read operation read the entire relation, a write operation wrote ten tuples. Data values were averaged over a sufficient number of runs to obtain 90 percent confidence intervals, calculated using the Student's distribution [15], of $\pm$ 0.05 sec for values under 0.40 sec and $\pm$ ten percent for values greater than or equal to 0.40 sec. In the tables of data, T_{Norm} denotes a normal transaction, T_{NV} a transaction that moves a relation to a new view, and T_{LW} a transaction that carries out lightweight quorum assignment change.

Normal transactions. Execution time for a transaction was measured at the coordinating site from the time the transaction was submitted for processing until the commit decision was reached. This time did not include the cost of interpreting the query nor of translating it to a transaction. Execution time for a normal read-write transaction in the absence of failures included a local read and phase one of the commit protocol. Phase one sends a round of approximately 1400-byte messages to the members of a write quorum, with each site of which does local logging. Times for normal transactions are shown in Table 1.

Empty view formation and moving a relation to a new view. Elapsed time for

	No deleted relations	5 deleted relations	10 deleted relations	50 deleted relations	100 deleted relations
3 sites	.08	.10	.10	.14	.16
5 sites	.10	.10	.12	.16	.18
10 sites	.22	.24	.25	.30	.34

Table 3: Execution time in seconds for view formation with inheritance

empty view formation included a round of approximately 60-byte messages to all sites, but no disk accesses. Times for empty view formation are shown in Table 2. Execution times for read-write transactions that move relations to a new view are shown in Table 1. These times are 20 to 40 percent greater than normal for the five- and ten-site cases, due to the need for two additional rounds of messages during the read phase to contact a backup read quorum and read an up-to-date copy of the relation. Additional rounds are not needed for the three-site case, however, since the degree of data replication for this case is two and backup read quorums are of size one. Using the values for ten sites, a rough estimate for *NVtoNormResp* is 1.4. In our implementation, a normal transaction generates an average of 1.8 requests per Transaction Manager server, while a transaction that moves the relation to a new view generates an average of 2.4 requests per server. Hence, disregarding the lengths of the requests, a rough estimate for NVtoNormLoad is 1.3.

View formation with inheritance. Execution times for view formation with inheritance are shown in Table 3. The increase with the number of deleted relations is due to the longer messages needed to contain the identifiers of deleted relations. Following view formation with inheritance, a transaction accessing a relation that had not been deleted from the old view took the same amount of time as in the absence of failures. A transaction accessing a relation that had been deleted took the same amount of time as if the transaction had followed empty view formation, because it had to move the relation to the new view.

View extension. Times for view extension are shown in Table 2. View extension took about the same amount of time as empty view formation. Fewer total messages are required for view extension, but the number and sizes of messages sent between when the timing starts and the view change is committed are the same in both cases.

Lightweight quorum assignment change. Execution times for transactions that carry out a lightweight quorum assignment changes following view extension are shown in Table 1. The increase over the normal execution time, about 10 to 20 percent for the five- and ten-site cases, is less than the additional time for a move to a new view. This is because only one additional round of messages is required in the read phase and only one site needs to be read from for lightweight change, compared to two additional rounds and reading from a backup read quorum for a move to a new view. The exception is the three-site case, where the degree of data replication is two. Our software was not smart enough to recognize in this case that, with a backup write quorum consisting of both sites having copies, the local copy of a relation was guaranteed to be up-to-date following view extension. Instead the transaction read at the remote site, making the lightweight change more expensive for the three-site case than a move to a new view.

Again using the values for ten sites, a rough estimate for *LWtoNormResp* is 1.1. In our implementation, the average number of requests per TM server for a transaction that does a lightweight quorum assignment change is 1.9 for a 10-site configuration, compared to 1.8 for a normal transaction. Thus, a rough estimate for *LWtoNormLoad* is 1.05.

6 Conclusions

We have investigated techniques for adapting the recovery actions of a dynamic quorum method to the length and extent of site failures and network partitioning. The major contributions of this research are the following:

- We have proposed a new recovery technique called inheritance that allows restoration of a previous configuration of quorum assignments with a minimal amount of work. Inheritance should be useful for recovery from a short-lived single site failure or simple partitioning, or from multiple site failures if all are repaired fairly quickly. For example, inheritance would be applicable to a redundant system with backup components that may be brought on-line quickly. For longer failures, inheritance allows the system to acquire quorum assignments from the largest current configuration, with the additional work required depending on the number of sites outside this configuration.

- For the special case of site recovery, we have proposed a new recovery technique that allows the current view to be extended to include the recovered site. Such view extension is expected to provide efficient recovery from lengthy site failures.
- We have narrowed down the need for formation of a new view to the case where no active quorum is available for an operation. Quorum assignment changes invoked for other reasons, such as addition or deletion of copies or the restructuring of backups quorums, may be done without forming a new view.
- We have implemented our dynamic quorum method in an actual distributed system. We have collected preliminary experimental data, and in the next section we point out directions for future experimental work.

Calculations based on our analytical model show that inheritance and view extension improve transaction processing performance following the repair of failures. Our calculations support our conjecture that when view formation with inheritance is used to handle the repair of a network partitioning, the type of inheritance that will yield better performance for subsequent transaction processing depends on the duration of the partitioning. If the partitioning is of short duration, then inheritance from the old view that existed prior to the partitioning gives better performance. On the other hand, if the network has been partitioned for a long time, then better performance will result if inheritance is from the view for the larger of the two components of the partitioned network.

We obtained similar results for site recovery, in that the choice of whether to use inheritance from the old view that existed prior to the site failure, or to extend the current view for the non-failed part of the network to include the failed site, depends on the duration of the site failure. For recovery from a short-lived failure, inheritance yields better subsequent transaction processing performance. For a long failure, view extension is better.

Analyses similar to our analyses of simple partitioning and single site failures can be carried out for the cases of multiple partitioning (when the network is partitioned into more than two components) and multiple site failures. In these cases, repair may be partial, in that proper subsets of the network may recover, and perhaps be subject to further failures, before the entire network is reconnected. The analyses for these cases are more complicated, but use similar techniques. Algorithms are needed that take as input all the information available about previous and current views and determine whether an old or current view should be used for inheritance or extension. Rather than making simplifying assumptions about the access pattern and processing rate of transactions that run during the failures, as we did in our analyses, these algorithms should use actual transaction execution histories, or summaries thereof, to do a more accurate comparison of the different options for recovery. Such on-the-fly analysis would make the system truly adaptable to the types and duration of failures.

References

[1] A. E. Abbadi and S. Toueg. The group paradigm for concurrency control protocols. In *ACM-SIGMOD '88 International Conference on Management of Data*, pages 126–134, June 1988.

[2] A. E. Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. Database Syst.*, 14(2):264–290, June 1989.

[3] N. A. Alexandridis. Adaptable software and hardware: Problems and solutions. *IEEE Computer*, 19(2), Feb. 1986.

[4] A. Avizienis. Fault-tolerant systems. *IEEE Transactions on Computers*, C-25(12):1304–1312, Dec. 1976.

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

[6] B. Bhargava, P. Noll, and D. Sabo. An experimental analysis of replicated copy control during site failure and recovery. In *Proc. 4th IEEE Data Engineering Conference*, pages 82–91, Los Angeles, Feb. 1988.

[7] B. Bhargava and J. Riedl. A model for adaptable systems for transaction processing. *IEEE Trans. on Knowledge and Data Engineering*, 1(4):433–449, Dec. 1989.

[8] B. Bhargava and J. Riedl. The RAID distributed database system. *IEEE Trans. on Software Engineering*, SE-15(6):726–736, June 1989.

[9] B. Bhargava and Z. Ruan. Site recovery in replicated distributed database systems. In B. Bhargava, editor, *Concurrency Control and Reliability in Distributed Systems*, pages 334–347. Van Nostrand Reinhold, 1987.

[10] S. Browne. *Recovery in Replicated Database Systems.* PhD thesis, Purdue University, 1990. in preparation.

[11] D. K. Gifford. Weighted voting for replicated data. In *Proc. Seventh Symposium on Operating Systems Principles*, pages 150–162. ACM, Dec. 1979.

[12] A. A. Heddaya. *Managing Event-based Replication for Abstract Data Types in Distributed Systems.* PhD thesis, Harvard University, Oct. 1988. TR-20-88.

[13] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst.*, 12(2):170–194, June 1987.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[15] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*, chapter 4. McGraw-Hill Book Company, 1982.

[16] Z. Ruan. *File replication in distributed systems.* PhD thesis, Purdue University, Aug. 1986.

[17] D. Skeen. A quorum-based commit protocol. In *Proc. 6th Berkeley Workshop*, pages 69–80, Feb. 1982.

A Appendix – Proof of Correctness

The generally accepted criterion for correct transaction processing in replicated databases is one-copy serializability, as defined in [5]. In [1], the problem of achieving one-copy serializability is divided into two subproblems. Transactions are considered to be divided into groups, which are disjoint sets of transactions. One subproblem consists of using a local policy to serialize the transactions within a single group. The second subproblem consists of finding a global policy to serialize all the groups.

In our adaptable dynamic quorum method, all transactions executing within the same view form a group. We first show that our quorum intersection rules and the rules for lightweight quorum assignment changes are sufficient to ensure one-copy serializability within a group. To carry out the proof, we use the concept of a one-serializability testing graph (1-STG), as defined in [9]. Secondly, we prove that our methods for view formation, view change, and object inheritance satisfy the sufficient conditions for global one-copy serializability given in [1].

Notation: A logical object has a set of physical copies. A logical object is denoted by a capital letter. A physical copy is denoted by the lower-case letter subscripted by the site at which it resides. An execution history $(H, <)$ of a set of transactions $T = \{T_a, T_b, \ldots\}$ is a partially ordered set containing all the physical operations of the transactions. Physically conflicting operations are ordered. The physical conflict graph, or CG, for a history $(H, <)$ is a graph $(T, \rightarrow)$ with the edges giving the partial order. A transaction T_b *reads-x_i-from* T_a, denoted $T_a \Rightarrow_{x_i} T_b$, if T_a writes x_i (denoted $w_a[x_i]$), T_b reads x_i (denoted $r_b[x_i]$), $w_a[x_i] < r_b[x_i]$, and there is no T_c such that $w_a[x_i] < w_c[x_i] < r_b[x_i]$. A transaction T_b *READS-X-FROM* T_a, denoted $T_a \Rightarrow_X T_b$, if there is some x_i such that $T_a \Rightarrow_{x_i} T_b$ and T_b uses the value written to x_i by T_a.

A.1 Proof of one-copy serializability within a single view

Before giving the proof, we present some necessary background from [9].

A one-serializability testing graph, or 1-STG, of a history $(H, <)$ is a graph $(T, \rightarrow)$ with the following properties:

1. If $T_a \Rightarrow_X T_b$, then there exists an edge $T_a \rightarrow T_b$ (read-from edge).

2. There is an edge between any two transactions that write to copies of the same object X (write-order edge, denoted $\rightarrow_X$).

3. If $T_a \Rightarrow_X T_b$ and $T_a \rightarrow_X T_c$, then there is an edge $T_b \rightarrow T_c$ (read-before edge).

It is shown in [16] that a history $(H, <)$ is one-copy serializable if and only if $(H, <)$ has an acyclic 1-STG. Hence, we need to show that the execution history $(H, <)$ of all the transactions in a single view has an acyclic 1-STG. The transactions to be considered include all user transactions as well as lightweight quorum assignment change transactions.

We show that the CG of $(H, <)$ can be augmented to form an acyclic 1-STG. Since we assume a correct distributed concurrency control protocol is being used, CG is acyclic. First we take the transitive closure of CG, denoted CG*.

Proof. The properties of a 1-STG are satisfied as follows:

1. Suppose $T_a \Rightarrow_X T_b$. By the definition of the *READ-FROM* relation, there is an edge in CG from T_a to T_b.

2. Suppose T_a and T_b both write object X. Add an edge between T_a and T_b in the direction of increasing version numbers or commit timestamps written by the write operations, unless such an edge already exists. That such an edge cannot create a cycle can be shown as follows:

   (a) Case 1: Suppose version numbers are being used.
   Then T_a and T_b access read quorums as well as write quorums.

      i. Case 1a: T_a and T_b use the same active quorum assignment. Then T_a and T_b physically conflict, and hence there is an edge in CG between T_a and T_b. The edge is in the direction of increasing version numbers.

      ii. Case 1b: T_a and T_b use different active quorum assignments. By an argument similar to that in 3(b) below, there is a path in CG, and hence an edge in CG*, in the direction of increasing version numbers.

   (b) Case 2: Suppose commit timestamps are being used.
   Suppose adding the edge in the direction of increasing commit timestamps creates a cycle. By the properties of logical clocks and conflict-based concurrency control, edges in CG are also in the direction of increasing commit timestamps. Hence, commit timestamps must increase along every edge in the cycle, which is clearly impossible. Thus, adding the edge cannot create a cycle.

3. Suppose $T_a \Rightarrow_X T_b$ and $T_a \rightarrow_X T_c$.

   (a) Case 1: T_b and T_c use the same active quorum assignments. Then there is an edge in CG from T_b to T_c since the read quorum used by T_b must intersect the write quorum used by T_c at some copy x_i and it must be the case that $r_b[x_i] < w_c[x_i]$.

   (b) Case 2: T_b and T_c use different active quorum assignments. Then one or more quorum change transactions for X must have been executed in the view. We consider only one such quorum change; the argument for more than one quorum change is similar. Let T_d be the quorum change transaction. Since T_d writes the quorum assignment change to an old read coquorum and an old write coquorum and to all new quorum members, and both T_b and T_c read the quorum assignment for X, there is path via T_d in CG, and hence an edge in CG*, between T_b and T_c. Suppose the direction of this path is from T_c to T_b. Then since the version written by T_c is more recent than the version written by T_a and T_d copies from an old write coquorum to a new read coquorum, T_b would not have read from T_a, a contradiction. Hence, the edge in CG* is from T_b to T_c. $\bowtie$

A.2 Proof of global one-copy serializability

In [1], conditions on the global policy sufficient to guarantee one-copy serializability, provided the transactions within each view are one-copy serializable with order $<_v$, are given as follows:

> A group v writes (reads) an object X if there is a transaction in v that writes (reads) X. A group v_j *READS-X-FROM* v_i is v_j reads a value of X written by v_i. A global policy ensures one-copy serializability if it imposes a total order $<$ on all groups such that if v_j *READS-X-FROM* v_i and $i \neq j$, then
>
> 1. $v_i < v_j$,
> 2. there is no v_k that writes X such that $v_i < v_k < v_j$, and
> 3. v_j reads the final value of X in v_i with respect to $<_{v_i}$.

We now show that our view formation, view change, and object inheritance rules make up a global policy that satisfies the above conditions. The view ids impose a total order on the views, or groups. If commit timestamps are issued by reading a logical clock [14], then the logical clock value should be prefixed by the view id so that the commit timestamps will be consistent with the global serialization order. We consider a transaction that moves an object into a new view to execute in the new view.

Proof. Suppose v_j *READS-X-FROM* v_i. The conditions are satisfied as follows:

Case 1: X was moved into v_j by a view change transaction.

1. The transaction in v_j that reads X from v_i is the view change transaction that moves X into the new view. The value for X is read from a backup read quorum. Since the view change transaction aborts unless the view ids returned by all members of the backup read quorum are less than the new view id, $v_i < v_j$.

2. Suppose there is a view v_k that writes X such that $v_i < v_k < v_j$. Assume that v_k is the greatest such view. Then all the active write quorums used in v_k either intersect the backup read quorum Q used by the view change transaction for X in v_j, or had the last value written to them copied to a new backup write quorum that does intersect Q. Hence, v_j would have read X from v_k, a contradiction.

3. Since the last (according to $<_{v_i}$) active write quorum used in v_i either intersects the backup read quorum Q used by the view change transaction in v_j, or had the last value written to it copied to a backup write quorum that does intersect Q, v_j reads the last value of X written in v_i, as given by version numbers or commit timestamps, both of which are consistent with the serialization order $<_{v_i}$.

Case 2. X was inherited by v_j from v_l.

1. It must be the case that $v_l < v_j$ since a site only allows an object to be inherited if the new view is greater than the old one. Since read access is inherited by v_j, X must have had read access in v_l. We claim that $v_i \leq v_l$. Otherwise, if $v_l < v_i$, write access for X would have been deleted from v_l. Since the view to which X would have been moved with write access could not have been disjoint from v_l (remember that backup read and write quorums must intersect), at least one site in v_l would have known about the deletion and the inheritance from v_l to v_j would not have been allowed. Hence, $v_i < v_j$.

2. Suppose there is a view v_k that writes X such that $v_i < v_k < v_j$. Assume v_k is the greatest such view. Since v_k writes X, by an argument similar to that above for v_i, $v_k \leq v_l$. If v_l writes X, then $v_k = v_l$. Otherwise, there is a chain of inheritances from v_k to v_j. In either case, since the last write quorum used in v_k must intersect the read quorum used in v_j, v_j would have read X from v_k, a contradiction.

3. Since the last (according to $<_{v_i}$) active write quorum used in v_i intersects the read quorum used in v_j (any quorum assignment change transaction that would have caused this not to be true is ruled out by 2. above), and version numbers or commit timestamps are consistent with $<_{v_i}$, v_j will read the last value of X written in v_i. $\bowtie$