A Weighted Voting Algorithm for Replicated Directories

(Revised Issue)

Joshua J. Bloch, Dean S. Daniels and Alfred Z. Spector

July 3, 1986

Abstract

Weighted voting is used as the basis for a replication technique for directories. This technique affords arbitrarily high data availability as well as high concurrency. Efficient algorithms are presented for all of the standard directory operations. A structural property of the replicated directory that permits the construction of an efficient algorithm for deletions is proven. Simulation results are presented and the system is modeled and analyzed. The analysis agrees well with the simulation, and the space and time performance are shown to be good for all configurations of the system.

Technical Report CMU-CS-86-132, Revision of CMU-CS-84-114

This work was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520 and in part by the NSF under Contract MCS-8308805

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or the US government.

Table of Contents

1 Introduction

The goals of object replication on distributed computing systems are increased parallelism, reduced communications costs, and increased resilience in the presence of failures. In particular, replication can permit increased object *availability* - continued access to an object despite the failure of one or more of the nodes on which it is stored. Unfortunately, it is difficult to achieve high performance and availability while ensuring that the semantics of replicated data objects are identical to those of their non-replicated counterparts.

In this paper, we describe and analyze a scheme for replicating directories that permits concurrent operations and arbitrarily high data availability. A measure of availability appropriate to this work is the number of node failures that a directory can tolerate while still guaranteeing that an operation can be performed. The semantics of the replicated directory are identical to a directory stored on a single node and accessed serially. Thus the replication algorithm is said to be *transparent.*

We define a *directory* as an abstract data object that maps *keys* to *values.* Keys are chosen from a set of constants called the *key space.* The only condition we impose on the key space is that it be *totally ordered.* In other words, there must be an ordering function '>' with the property that for any two members of the key space x and y, one and only one of the following conditions hold: $x > y$, $y > x$ or $x = y$. We make no other assumptions about the structure of the key space; it can be finite, countably infinite, or uncountably infinite, and either dense (like the rationals) or sparse (like the integers). For all of the examples in this paper, we use the set of finite length alphabetic strings as the key space, with lexical comparison as the ordering function. Whenever we use words like 'less than', 'above' or 'in between' in reference to keys, we mean according to the ordering function.

Directory operations execute as part of distributed transactions, which provide uniform synchronization and recovery properties for operations on arbitrary shared abstract types. (Transactions have been described by many; see, for example, a survey by Gray or recent work by Spector et al. [Gray 80, Spector et al. 85a].) The replicated directory is an example of a distributed abstract data type that is constructed from a collection of more primitive, non-distributed types. Transactions simplify the maintenance of the invariants necessary to make this replication algorithm work.

Directories are accessed and modified with the following operations:

- **Insert(IN K:Key, V:Value)** - Associates the value V with the key K. Once inserted, the key is said to be *in the directory.* This operation is permitted only when K is not already in the directory.
- **Update(IN K:Key, V:Value)** - Associates the (new) value V with the key K. This operation is permitted only when K is already in the directory.
- **Delete(IN K:Key)** - Removes K from the directory. This operation is permitted only when K is in the directory. After this operation is performed, K will no longer be in the directory.

- **Lookup(IN K:Key; OUT IsIn:Boolean, V:Value)** - Returns TRUE, and the value associated with K, if K is in the directory. Returns FALSE and an undefined value if K is not in the directory.

Attempting to perform an operation that is not permitted generates an exception. It does not affect the contents of the directory. Minor modifications of our scheme may be used to implement sets, multisets or similar abstractions.

The replication algorithm described here is an extension of one initially presented by Daniels and Spector [Daniels and Spector 83]. It is based on Gifford's weighted voting algorithm [Gifford 79, Gifford 81], and has similar performance and reliability advantages. However, unlike Gifford's algorithm, this algorithm efficiently associates a separate version number with each possible key at every replica. This permits concurrent operations on different entries and solves certain problems in the implementation of the **Delete** operation. Unlike most replication algorithms, which are concerned with simple objects having only *read* and *write* operations, this algorithm uses the semantic properties of directories, and thereby gains increased performance.

This research on replication was done as part of the TABS (Transaction-based Systems) Project, which constructed a distributed transaction facility that supports operations on shared abstract data types [Spector and Schwarz 83, Schwarz and Spector 84, Spector et al. 85b, Spector et al. 85a]. This directory replication algorithm was implemented and videotaped to demonstrate the facility's operation. Groups at Cornell, MIT, and Georgia Institute of Technology are also investigating the wider use of transactions [Allchin and McKendry 83, Allchin 83, Birman et al. 83, Liskov and Scheifler 83, Weihl and Liskov 83, Weihl 83].

In the following sections, we survey related replication work and provide motivation for our directory replication algorithm. We describe the algorithm in detail and present efficient algorithms for each directory operation. A basic structural property of the replicated directory, which permits the construction of an efficient algorithm for the **Delete** operation, is proven. We show that the system's concurrency performance can be improved by relaxing the synchronization requirements for the directory replicas. We present performance data obtained by simulation and develop a mathematical model of the system simulated. We analyze the model and compare the results of the simulation and the analysis. These results demonstrate that the algorithm's space and time requirements are good in all configurations of the system. Finally, we discuss the advantages and uses of the algorithm.

2 Related Work and Motivation

2.1 Related Work

There are non-distributed and distributed approaches to data replication. In the non-distributed approaches, a single controlling node utilizes dual-copy, or mirrored, storage. Storage is typically on disks that are located in close proximity to each other. Data is written sequentially to both copies, but read from only one. Should a controlling node crash, another node gains control of the storage. Mirroring is commonly used on commercially available systems; for examples, see descriptions of the ACP or Tandem T16 systems [IBM Corporation 75, Bartlett 81].

Distributed replication techniques use a collection of cooperating nodes to store replicas of the data. Many of these techniques provide higher reliability and availability than mirroring, though they generally have higher overhead and complexity. For example, an object replicated with mirroring will not survive a single physical disaster that destroys both copies, while replication techniques employing geographical distribution will be less affected by such a physical disaster.

In this section we briefly survey the field of distributed replication algorithms. We present our motivation for developing a replication strategy for directories based on the weighted voting technique.

One fundamental distributed replication strategy is unanimous update: every update operation must be done on all replicas, but reads may be directed to any replica. This replication strategy guarantees single copy semantics if the systems storing each replica guarantee data consistency locally. Unfortunately, the availability for updates of any object is poor when large numbers of replicas are used. Update availability can be increased by using the communication system to buffer updates to replicas that are not available. The SDD-1 distributed database system uses this approach [Rothnie et al. 77]. A similar approach is taken in the *available copies* method [Bernstein and Goodman 84]. None of these methods handle network partitions.

A second approach to replication is based on keeping primary and secondary copies of data. The primary copy receives all updates and then relays the updates to the secondary copies [Alsberg and Day 76]. An inquiry may be sent to a secondary copy, but the result might not reflect the most recent updates. Because responses to inquiries might not reflect recent updates, it is difficult for a primary/secondary copy replication strategy to duplicate the semantics of a non-replicated object. Techniques for alleviating this problem have been developed. For example, each file open operation in the Locus distributed file system ensures the currency of data by consulting a known synchronization site [Popek et al. 81]. Locus maintains availability after synchronization site failure by nominating a new synchronization site.

A third basic approach to replication is *weighted voting* [Gifford 79, Gifford 81]. We describe this approach in more detail since it forms the basis of our algorithm. A file is stored as a collection of replicas, called

representatives, each of which is assigned a certain number of votes. A representative consists of a copy of the file and a version number. The entire collection of representatives is called a *file suite*. Write operations write an updated copy of the file to each representative in a group called a *write quorum* and associate a new version number with all of these representatives. The new version number is higher than any version number previously associated with the file. Read operations read from each representative in a *read quorum* and return the data from the representative with the highest version number. Write operations establish a higher version number by incrementing the highest version number encountered in a read quorum.

A write quorum consists of any set of representatives whose votes total at least W and a read quorum consists of any set of representatives whose votes total at least R. The constants R and W are chosen so that their sum is greater than the total number of votes assigned to all representatives, N. Thus, every read quorum has a non-null intersection with every write quorum and each inquiry is guaranteed to access at least one current copy of the data. Current copies will always have a higher version number than non-current copies so the read operation will always return current data. The values chosen for R and W control a tradeoff between the cost and availability of read and write operations.

Abbadi, Skeen and Christian extend the available copies approach to handle partitions [Abbadi et al. 85]. In this approach, the nodes maintain *virtual partitions*, which are logical groups corresponding to perceived actual partitions. The unanimous update approach is used within each virtual partition. Only a virtual partition containing a majority of the replicas for any object can access that object.

Abbadi and Toueg extend the virtual partions approach to gain added flexibility [Abbadi and Toueg 86]. In this system, nodes maintain *views*, similar to the virtual partitions described above. Within each view, the weighted voting technique is used. Performance and availability tradeoffs between read and write operations can be controlled by choosing appropriate quorum sizes.

All of the replication methods above apply to *files*, data objects supporting only read and write operations. Herlihy describes a technique called *generalized quorum consensus* whereby the weighted voting technique can be systematically applied to any abstract data type [Herlihy 86]. This technique is completely general but results in implementations that are costly in terms of communications, storage, and computation. Herlihy suggests some optimizations to decrease these costs, but the emphasis in his work is on complete generality and theoretical investigation of quorum intersection issues rather than on providing efficient implementations.

2.2 Motivation for the Use of Weighted Voting

Weighted voting has several attributes that make it appealing as the basis for the design of a replicated directory. The sizes of the read and write quorums may be varied to adjust the relative cost and availability of the operations. For example, read quorums can be made much smaller than write quorums if data is read

more frequently than it is written. Vote assignments can be adjusted to further refine availability tradeoffs. For instance, a node that is more likely to fail can be given fewer votes, so its absence will have less effect on system availability.

Another appealing attribute of weighted voting algorithms is that they automatically function correctly in the face of network partitions. They do so passively, without the need for dynamic reconfiguration, as in the virtual partitions and views techniques. Such dynamic reconfiguration adds great complexity to replication algorithms, and can substantially reduce availability during periods of reconfiguration.

Finally, algorithms based on weighted voting are simplified because consistency and recovery are primarily the responsibility of an underlying transaction facility. The use of a common underlying transaction facility greatly simplifies the task of ensuring that operations on multiple distributed objects interact properly.

While weighted voting is an appealing approach to replication, the basic algorithm cannot be directly applied to directories without undesirable concurrency limitations. Even though the semantics of directories permit concurrent operations on different keys, only a single transaction at a time could modify the directory if it were stored as a file suite. This is because each copy of the entire directory would have a single version number, which would cause the serialization of all operations that modified the directory. Furthermore, any modification to the directory would require sending the entire updated directory to each representative in a write quorum. For a large directory, this would result in excessive communications costs. In the following section, we develop an algorithm for replicated directories from the weighted voting algorithm for files. Our algorithm rectifies the deficiencies described above.

The use of version numbers in weighted voting has certain disadvantages. In order to write a file, a node must know the highest version number currently associated with the file. This requires access to a read quorum of representatives. Therefore, even if the write quorum size (W) is smaller than the read quorum size (R), it requires the services of R representatives to perform a write operation. If one were to configure a suite with $R < W$, the desired increase in availability of the write operation would never materialize, while the availability of the the read operation (and the write operation) would decrease. Thus, the use of version numbers is seen to restrict the permissible range of availability tradeoffs between the read and write operations.

The restriction of permissible quorum choices described above comes about because one must perform a "read operation" on a file's version number before performing a write operation on the file. It could be eliminated if one could determine a higher version number than those already used without consulting the version numbers present. Gifford suggests the use of timestamps instead of version numbers for this purpose [Gifford 81] and Herlihy's techniques use timestamps [Herlihy 86, Herlihy 85]. The advantages of the timestamp approach are not without cost. It is critical that timestamps reflect the serialization order on transactions. This requires support from the transaction system. Herlihy discusses this issue in the works cited above.

The algorithm presented in this paper is equally compatible with version numbers and timestamps. Our initial implementation used version numbers because the TABS system lacked support for timestamps.[1] The description of our algorithm in this paper reflects our initial use of version numbers.

3 Development of the Algorithm

In the previous section, we noted that weighted voting could not be directly applied to a directory file without excessive concurrency limitations and communications costs. It might seem that these limitations could be overcome if each *entry* in a directory representative were assigned a separate version number. (An *entry* is defined as the physical data associated with a key at a representative and consists of the key and an associated value.) However, if such an approach were used, some representatives might not have an entry for a key that had an entry at other representatives. Because of this fact, it would not always be possible to determine from an arbitrary read quorum whether a particular key were in the directory. This problem is illustrated in the example that follows.

The directory suite for the example will contain 3 representatives. In this example and those that follow, we will assume that each representative has one vote, though all results generalize to directory suites with arbitrary distributions of votes. The read quorum size for the example is 2 votes and the write quorum size is 3 votes. The notation N-R-W will refer to a suite having N representatives, a read quorum size of R and a write quorum size of W. Thus we call the suite in our example a 3-2-2.

Initially representatives A and B contain entries for keys "a" and "c", and each entry has version number 1 as shown in Figure 1[2]. Subsequently an entry for "b" is inserted into representatives A and C with version number 1 (Figure 2). If a request to look up the key "b" is sent to representatives B and C at this point, representative B will respond "not present," and representative C will respond "present with version number 1." If "b" is then removed from the directory by deleting its entry from representatives A and B (Figure 3), requests to look up "b" on representatives B and C will still elicit the responses "not present," and "present with version number 1." Thus, if a directory representative fails to associate a version number with keys for which it has no entry, the responses from a read quorum may not be sufficient to determine if a given key is in the directory.

The ambiguity demonstrated above is associated with deletions and will not occur if deletions are not permitted. Alternatively, deletions could be implemented by marking entries to be deleted and then performing a "garbage collection" operation periodically. However, that operation is expensive and would itself be a concurrency bottleneck. A third strategy is to eliminate the ambiguity by consulting additional

---

[1] Support for timestamps has since been added to TABS.

[2] The value field is omitted from all figures for clarity.

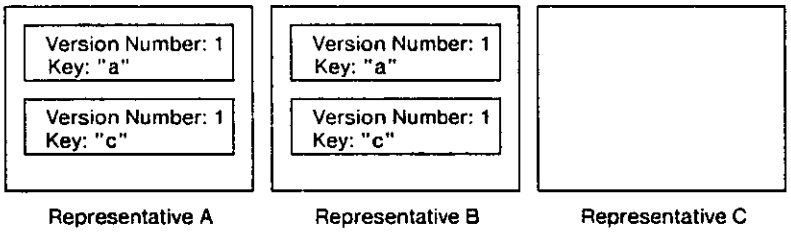

Figure 1: A 3-2-2 Directory Suite - Initial Configuration

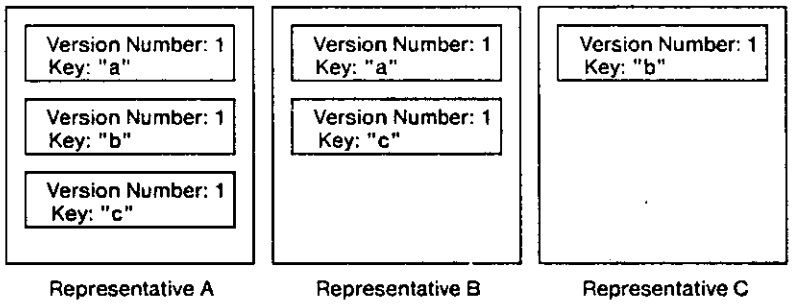

Figure 2: Directory Suite After Inserting "b"

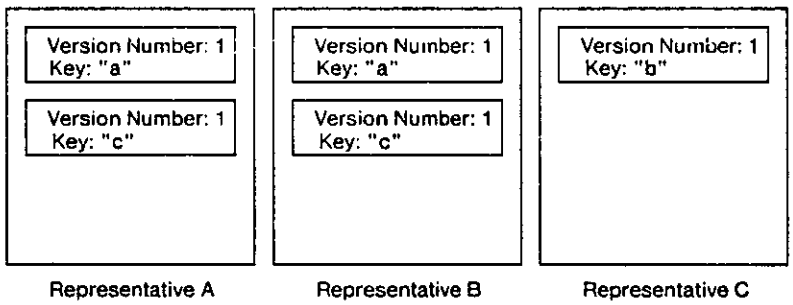

Figure 3: Directory Suite After Deleting "b"

representatives whenever an inquiry to an initial set of representatives does not result in a read quorum of replies all indicating "present" or "not present." Unfortunately, this approach drastically reduces availability.

None of the solutions presented thus far satisfy our demands for concurrency and availability. What is really needed is a scheme whereby version numbers can be associated with every possible key in the key space at each representative. This can be accomplished by partitioning the key space into disjoint sets and associating a version number with each set at every representative. The same partitions need not be used at all representatives.

One approach to partitioning is to divide the key space into ranges based on the order relation on the keys. The simplest partitioning scheme divides the key space into a number of fixed ranges. However, it is difficult to guarantee sufficient concurrency with such a *static partitioning* technique. If a small number of ranges are

used, then at most that number of transactions can modify a directory concurrently. If transactions modify entries in more than one range, concurrency will be further limited. Even if a large number of ranges are used, an uneven distribution of accesses could limit concurrency.

A more general method of partitioning is to allow the partitions at each representative to vary over time, on the basis of the entries currently in that representative. Such a *dynamic partitioning* technique is desirable for directories having sizes or access patterns that vary widely over time. A simple method of dynamically partitioning the key space at a representative is to create a partition for each key that has an entry in that representative and a partition for each range of keys between successive entries. These ranges are called *gaps*. This method forms the basis of our algorithm.

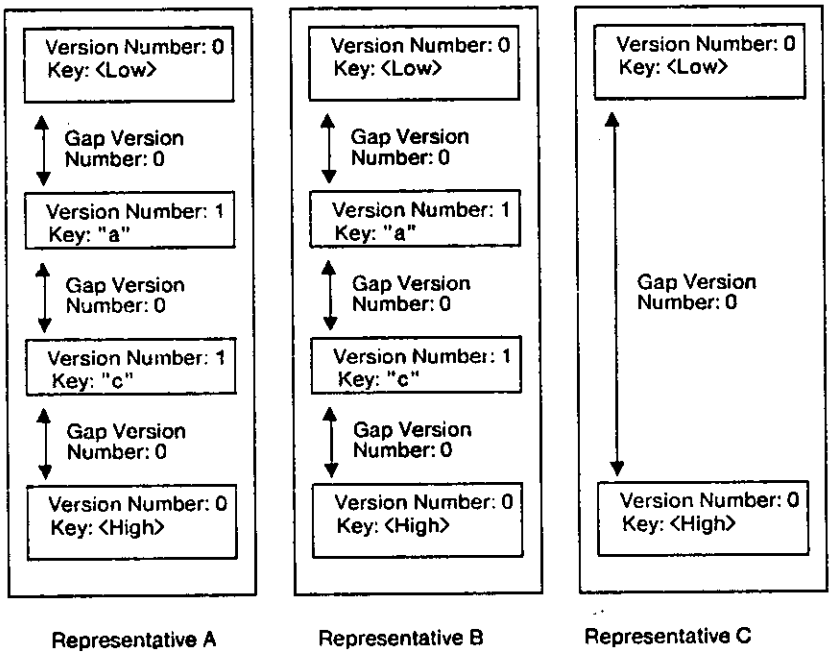

Figure 4: Directory Suite Containing Keys "a" and "c"

In this dynamic partitioning approach, lookup requests sent to a representative containing an entry for the key being looked up return the version number of the entry. Lookup requests on keys for which no entry is stored return the version number of the gap in which the key lies. Update requests increment the version number of the entry for the key being updated, insertion requests split a gap, and deletions coalesce the gaps and entries in a range of keys into a single gap. The details of these operations will be discussed at length in Section 4.

The suite containing entries for keys "a" and "c" in representatives A and B of our previous example

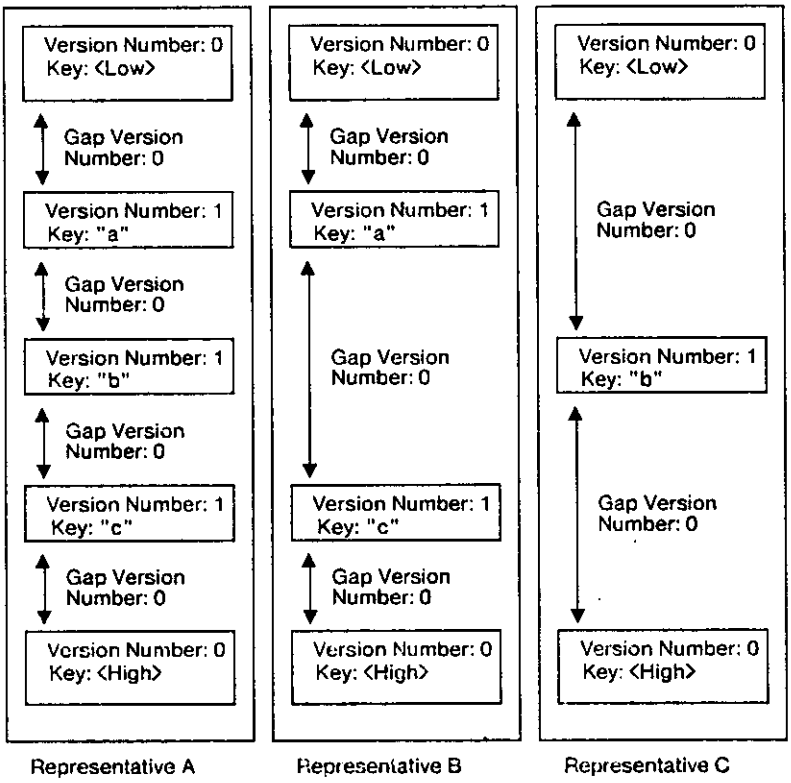

Figure 5: Directory Suite After Inserting "b"

(Figure 1) would be represented as in Figure 4.[3] If the key "b" is inserted into a write quorum consisting of representatives A and C, the suite in Figure 5 results. Note that, in representatives A and C, the entry for key "b" is assigned version number 1, which is one greater than the version number of the gap between "a" and "c".

If a request to look up "b" were sent to representatives B and C at this point, representative B would respond "not present with version number 0" and representative C would respond "present with version number 1." Using these responses, a client could determine that "b" was in the directory since the entry had a higher version number than the gap. If "b" were subsequently deleted from representatives A and B, then the two gaps on either side of "b" on representative B would be coalesced. On both representatives, the gap between "a" and "c" would be assigned version number 2 (Figure 6). Now, if a request to look up "b" were sent to representatives B and C, B would respond "Not present with version number 2" and C would respond "present with version number number 1." This response indicates that the key no longer exists, resolving the ambiguity that occurred in the initial example, wherein version numbers were associated only with entries.

[3] The directory representatives in Figure 4 contain the special keys LOW and HIGH, which delimit the first and last gaps in the representatives.

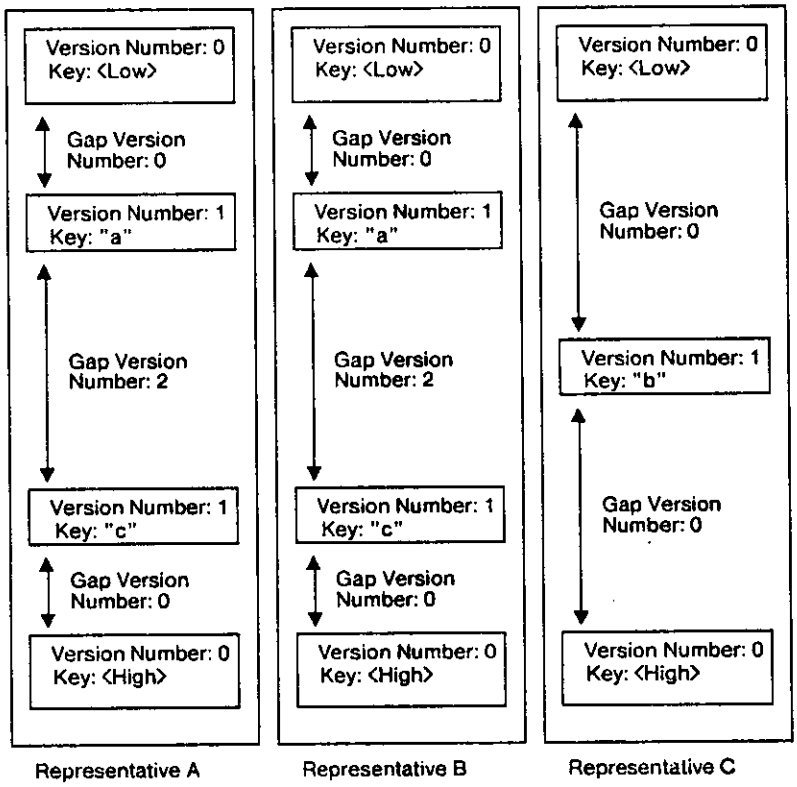

Figure 6: Directory Suite After Deleting "b"

4 Details of the Algorithm

This section presents the details of the algorithm sketched in the previous section. The descriptions are illustrated with program text in a Pascal-like language that includes a remote procedure call primitive [Birrell and Nelson 84]. Remote procedures are declared like ordinary procedures except that the first parameter is always the identifier of a remote server and that other parameters may be declared as IN, or OUT. Parameters are passed by value in messages. Remote procedure calls have the same syntax as local procedure calls and the general purpose distributed transaction facility which is assumed as the underpinning of this algorithm guarantees that the remote procedure calls have exactly-once semantics. Hence, the only exception mechanism provided is **ReportError**. Transactions are aborted by node failures, timeout and other system determined errors. Clarity is emphasized over performance in the programs. Optimizations that would be used in practical implementations are described in accompanying text.

Operations on directory representatives and directory suites are presented in the first two subsections. The next two subsections develop an essential component of the deletion algorithm. Arguments for the correctness of the replication algorithm are then presented. The final subsection discusses ways of modifying the algorithm's synchronization policies to provide higher concurrency.

4.1 Directory Representatives

In a replicated directory, each directory representative is an instance of an abstract object that stores one (approximate) copy of the directory data. Arbitrarily complex atomic transactions may be constructed using the basic operations provided by directory representatives. Thus, directory representatives must synchronize concurrent operations performed by different transactions and store critical information in a fashion that recovers from failures. Gifford's weighted voting algorithm makes similar requirements of its file representatives.

Every instance of a directory representative contains two distinguished keys, HIGH and LOW. HIGH is greater than any key in the key space and LOW is less than any key in the key space. HIGH and LOW simplify the **delete** operation by ensuring that all keys have a *real predecessor* and *real successor* in the directory. Real predecessor and real successor have an intuitive meaning, but are defined precisely in Section 4.2.

Directory representatives provide two operations that are analogous to typical directory primitives: **DirRepLookup** and **DirRepInsert**. **DirRepInsert** is defined to be useful for both the **Insert** and **Update** operations on directory suites. In addition, directory representatives provide specialized operations that are used to implement the directory suite deletion operation: **DirRepPredecessor**, **DirRepSuccessor**, **DirRepSuperseder**, and **DirRepCoalesce**. Figure 7 gives procedure headings for each of these operations. In Figure 7 the last line of each description specifies the locks set by the operation for synchronization. These locks are discussed below.

DirRepPredecessor returns the key and version number of the entry in the representative that is the immediate predecessor of the key passed as an argument; it also returns the version number of the gap between the keys. (Note that a version number is maintained for the gap between each pair of entries, even if the key space has no keys in the range that the entries delimit.) **DirRepSuccessor** is analogous to **DirRepPredecessor**. Deletions are performed on a directory representative using the **DirRepCoalesce** operation, which deletes any entries appearing in a range between two specified entries and assigns a single version number to the resultant gap. Thus, **DirRepCoalesce** coalesces a collection of keys and gaps into a single gap.

DirRepSuperseder is used in implementing the **delete** operation on directory suites. The operation searches a range starting with key x and ending with key y, and returns the entry closest to x with a version number greater than the one passed as a parameter. If the search reaches key y without locating an entry to return, then the entry for y (if one exists) is returned. The operation locates the first entry that "supersedes" a gap with the specified version number.

Each directory representative must synchronize the concurrent operations of different transactions. While this might be accomplished in many ways, the discussion presented here will assume that type-specific locking

```
DirRepLookup(IN server:DirRep,x:key;
             OUT Present:boolean,ver:version,val:value);
{ If there is an entry for x, returns TRUE, the version number of
  the entry, and its value; otherwise returns FALSE and the
  version number of the gap containing x.

  Locks RepLookup(x,x). }

DirRepInsert(IN server:DirRep,x:key,v:version,z:value);
{ Creates an entry for key x with version number v and value z.
  Updates the entry for key x if one already exists.

  Locks RepModify(x,x).}

DirRepPredecessor(IN server:DirRep,x:key;
            OUT Pred:key,PredVer:version,PredVal:value,GapVer:version);
{ Returns the key, version number, and value of the entry with the
  highest key less than x.  Also returns the version number of the gap
  between x and its predecessor.  There need not be an entry for x.

  Locks RepLookup(Pred,x) }

DirRepSuccessor(IN server:DirRep,x:key;
            OUT Succ:key,SuccVer:version,SuccVal:value,GapVer:version);
{ Analogous to above procedure.

  Locks RepLookup(x,Succ) }

DirRepCoalesce(IN server:DirRep,l:key,lver:version,lval:value,
                  h:key,hver:version,hval:value,gapver:version);
{ Inserts entries for l and h if they are not present.
  Deletes entries for any keys between (but not including) l and h.
  The resulting gap is assigned version number gapver.

  Locks RepModify(l,h). }

DirRepSuperseder(IN server:DirRep,x:key,v:version,y:key
                 OUT superseded:boolean,Superseder:key,
                     SupVer:version,SupVal:value);
{ Searches the range between x and y, starting from x. Returns TRUE,
  together with the key, version number, and value of the first
  entry examined between x and y (exclusive) with version
  number greater than v. Returns TRUE and the entry for y if it
  exists and no entry closer to x has version number greater
  than v.  Returns FALSE if there is no entry for y and no entry
  between x and y with version number greater than v.

  Locks RepLookup(x,Superseder).}
```

Figure 7: Directory Representative Operations

is used [Korth 83, Schwarz and Spector 84]. In type-specific locking, every operation on an abstract object acquires a lock that is a member of the set of locks associated with that object. A lock compatibility relation is used to determine whether a lock may be acquired by a particular transaction.

The lock classes used in synchronizing a directory representative are analogues of the lock classes for a single-copy directory, given by Schwarz [Schwarz and Spector 84]. However, instead of locking single keys, the lock classes are generalized to lock an entire range of keys and the granting of a lock depends on whether a range of keys to be locked intersects the range of keys already locked by some other transaction. Inquiry operations (**DirRepLookup**, **DirRepPredecessor**, **DirRepSuccessor**, and **DirRepSuperseder**) set **RepLookup(σ,τ)** locks, where the range of keys explicitly or implicitly accessed by the operation consists of the keys greater than or equal to σ and less than or equal to τ. A **RepModify(σ,τ)** lock is obtained on the range of keys modified by the **DirRepInsert** and **DirRepCoalesce** operations.

The lock compatibility relation for operations on directory representatives is illustrated in Figure 8. In the figure, [σ...τ] and [σ'...τ'] are arbitrary non-intersecting ranges of keys, and [σ...τ] and [σ"...τ"] are arbitrary intersecting key ranges. Locks are compatible except that a **RepModify** lock may not specify a range which intersects the range already specified by another **RepModify** lock, a **RepModify** lock may not specify a range which intersects the range already specified by a **RepLookup** lock, and a **RepLookup** lock may not specify a range which intersects a range already specified by a **RepModify** lock. For example, the compatibility relation specifies that a transaction may not be granted a **RepModify(σ",τ")** lock if another transaction already holds a **RepModify(σ,τ)** lock.

		Lock Held	
Lock Requested	None	**RepLookup(σ,τ)**	**RepModify(σ,τ)**
RepLookup(σ',τ')	OK	OK	Ok
RepModify(σ',τ')	OK	OK	Ok
RepLookup(σ",τ")	OK	OK	No
RepModify(σ",τ")	OK	No	No

Note: [σ..τ] *intersects* [σ"..τ"] *and* [σ..τ] *does not intersect* [σ'..τ']

Figure 8: Compatibility of Directory Representative Lock Classes

As specified above, the lock compatibility relation is sufficiently strong to guarantee that the actions of transactions operating on a directory representative are serializable [Traiger et al. 82], provided that two phase locking is used. This form of synchronization simplifies the correctness arguments given in Section 4.4. (Section 4.5 presents modifications to these locking rules that permit greater concurrency.)

Each directory representative is responsible for recovery processing. Recovery processing is necessary to undo the effects of partially completed transactions after a crash or when a transaction abort is requested by a client. In any recovery scheme it is necessary for a directory representative to record enough information

reliably to redo or undo the effects of those operations that modify the state of the representative. The details of recovery processing are specific to the implementation of a directory representative and depend on the recovery approach used by the underlying transaction system. Gray et al., Lindsay et al., and Schwarz, among others, present more details on general recovery algorithms [Gray et al. 81, Lindsay et al. 79, Schwarz 84].

To redo insert and update operations, the representative must have available the key, version number, and value of the modified entry. To undo updates, the old value and version number of the entry must also be recorded. Inserts are undone by coalescing the gaps on either side of the entry which was inserted. It is not necessary to record an old version number when performing an insertion, since the version number of the gaps on either side of an inserted key is the same as the old version number.

A coalesce operation may be redone in a straightforward manner. To be prepared to undo a coalesce operation, a representative must reliably record the key, value, version numbers of all entries deleted by the coalesce operation, and the version numbers of the gaps between entries.

4.2 Directory Suites

A directory suite consists of a set of N directory representatives, an assignment of votes to representatives, and the read and write quorum sizes R and W. The quorum sizes are chosen to conform to the constraints described below. Directory suites implement the operations **Lookup**, **Insert**, **Update**, and **Delete**, as specified in Section 1. Operations on directory representatives are combined to implement a replicated directory based on the weighted voting rules described in Section 2.

It can be seen from the specifications for the directory operations in Section 1 that the operations that modify the directory (**Insert**, **Update** and **Delete**) must first look up the key being modified in order to ensure the legality of the operation. Thus all operations require access to R representatives, and no additional availability can be gained by choosing $W<R$. Therefore we assume $W \geq R$. Recall from section 2.1 that $R+W>N$. Combining these two inequalities, we have $2W>N$. Thus any value between $\lfloor \frac{N}{2} \rfloor + 1$ and N, inclusive, can be chosen for W. R will generally be set to $N-W+1$, the smallest value necessary to ensure the required quorum intersection.

The **Lookup** operation calls the procedure **ILookup**, which is shown in Figure 9, and discards the version number, returning only a boolean indicating the presence of the key in the directory, and the value associated with key, if it is present. **ILookup** first calls **CollectReadQuorum**, a function that returns identifiers for a read quorum of directory representatives. Then, **DirRepLookup** operations are performed on the quorum and the entry with the highest version number is returned.

CollectReadQuorum and its companion function, **CollectWriteQuorum**, bind identifiers to instances of directory representatives. This may involve message exchanges to establish communications sessions, so it is

```
ILookup(IN k:key;OUT Present:boolean,Ver:version,Val:value)
{ Internal lookup procedure.  Return True, the version number, and the
  value associated with k if it is in the directory; False otherwise.}

var
   { read quorum has R members }
   quorum : array[1..R] of DirRep;
   RepsVer : version;
   RepsVal : value;
   RepsPresent,bestisin : boolean;
   i : integer;

begin
  { collect a read quorum for this operation}
  quorum := CollectReadQuorum;

  Ver := LowestVersion - 1; { a constant }
  { send inquiries to each quorum member }
  for i := 1 to R do
    begin
      DirRepLookup(quorum[i],k,RepsPresent,RepsVer,RepsVal);
      if RepsVer>Ver then
        begin
          Ver := RepsVer;
          Val := RepsVal;
          Present := RepPresent;
        end
    end;
end
```

Figure 9: ILookup Operation

desirable for the implementors of these operations to cache information to be used in subsequent invocations. Efficiency and availability are improved if the quorums returned by these functions overlap as much as possible.

The **Insert** operation is quite simple. **Insert** first uses **ILookup** to look up the key to be inserted in a read quorum and obtain the highest version number currently associated with the key. A version number one higher than this number is used for the new entry, which is then inserted into a write quorum of representatives. Figure 10 illustrates this operation. The **Update** operation is similar.

Delete must delete an entry from a write quorum by coalescing a range of keys that includes the entry to be deleted and assigning a version number to the resulting gap that is higher than that of any entry or gap previously contained in the range. To avoid asserting the nonexistence of keys that are actually in directory, the range to be coalesced may not contain keys in the directory other than the one to be deleted. **Delete** coalesces a range that extends from the *real predecessor* of the key to be deleted to its *real successor*, thereby ensuring that there are no keys in the directory that lie in the coalesced range. The real predecessor of a key k is the highest key less than k that is in the directory. The real successor of a key is defined analogously. The

```
Insert(nkey:key,nval:value);
{Insert a new entry with key nkey and value nval }
var
  { write quorum has W members }
  quorum : array[1..W] of DirRep;
  i : integer;
  k : key;
  ver : version;
  val : value;
  isin: boolean;

begin
  { first, lookup key to find the current version number }
  ILookup(nkey,isin,ver,val);
  { val ignored }
  if isin then ReportError;

  { find a write quorum }
  quorum := CollectWriteQuorum;

  { The new entry's version number must be higher than its
    previous version number as returned by the Lookup call }
  ver:=ver+1;

  { Insert the entry in each quorum member }
  for i:= 1 to W do
    DirRepInsert(quorum[i],nkey,ver,nval);

end
```

Figure 10: Insert Operation

coalesce operation inserts the real predecessor and successor into representatives where they are not already present, to delimit the newly formed gap. The entries between a key's real predecessor and its real successor on a representative comprise the key's *delete list* on that representative. The delete list is so named because it consists of the entries that are expunged when performing the coalesce operation required to delete a key.

Locating the real predecessor and real successor of a key to be deleted is complex. There may be *ghost* entries located between the key to be deleted and its real predecessor or real successor. A ghost is defined as an entry for a key that is no longer present in the directory suite. In addition, the real predecessor or real successor of a key might not be present in some members of the read quorum.

These problems are partially illustrated in the following example. Consider the suite in Figure 5. Suppose we delete key "a", using representatives A and C as the write quorum for the delete. This operation is straightforward, resulting in the suite shown in Figure 11. Now suppose we delete key "b", using representatives B and C as the write quorum. Figure 11 shows that the real successor of the key "b" is the key "c". However, no entry for "c" appears in appears in representative C, and the ghost of entry "a" appears between "b" and LOW (the real predecessor of "b") in representative B. To delete "b" from representatives B

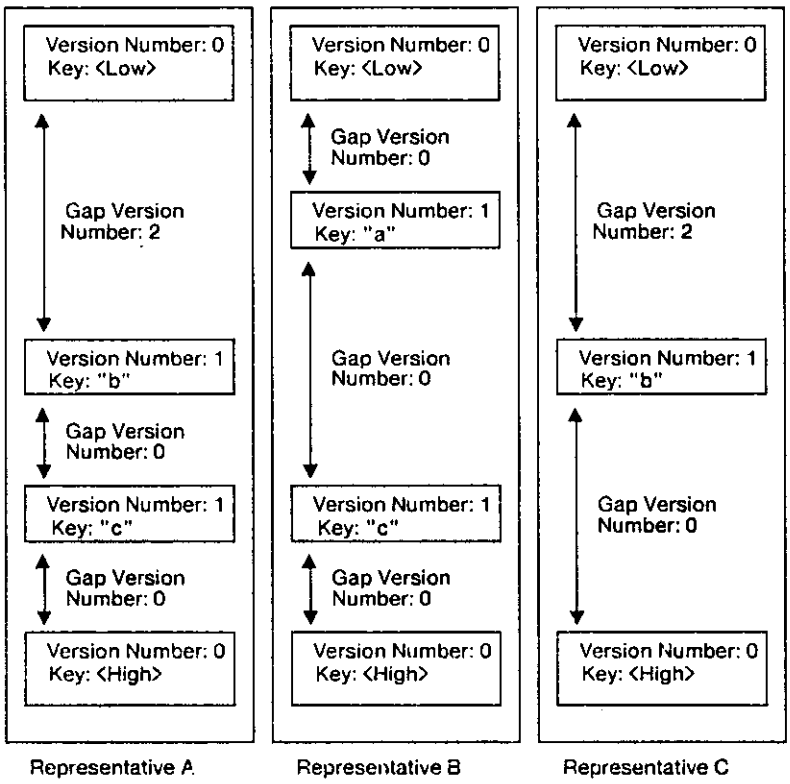

Figure 11: Directory Suite from Figure 4 After Deleting "a"

and C, the real successor, "c", must be inserted into representative C. The coalescing of the range from **LOW** to "c" eliminates the ghost of entry "a" from representative B. The resulting suite is shown in Figure 12.

A simple **Delete** procedure is illustrated in Figure 13. Finding the real predecessor and successor of a key is the heart of this operation. Given an input key, **RealPredecessor** returns the key, value, and version number of the input key's real predecessor. **RealPredecessor** also returns the version number for the gap between the key and its real predecessor. The **RealSuccessor** operation is analogous. The straightforward procedure given by Daniels and Spector [Daniels and Spector 83] for performing the real predecessor operation suffers from a serious drawback: it requires that messages be sent between the node determining the real predecessor and the nodes containing each member of a read quorum, for every ghost between the key being deleted and its real predecessor in all representatives of the quorum. While this message traffic can be reduced by combining messages, and while the simulations and analysis show that average performance is not too bad, the number of fixed length messages that must be transmitted for a single **Delete** operation is potentially unbounded.[4] All other directory suite operations, as presented previously [Daniels and Spector 83], require only a constant number of small, fixed length communications; it would be highly desirable to have an

[4] In fact, it is bounded by 2R * (the cardinality of the key space), where R is the read quorum size. For finite key spaces, this expression will be large but finite.

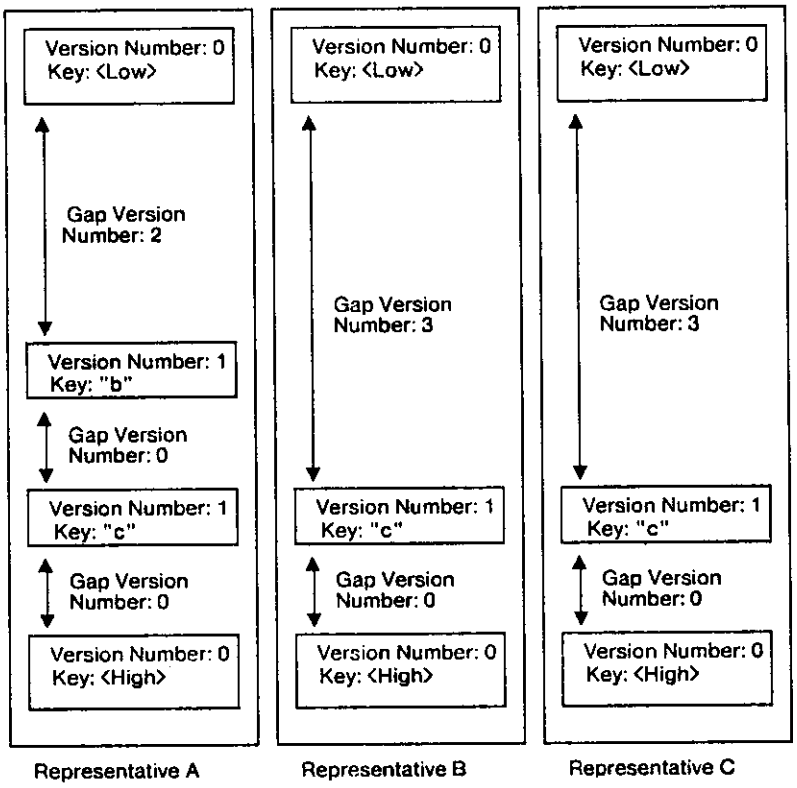

Figure 12: Directory Suite from Figure 11 After Deleting "b"

algorithm for the real predecessor operation (hence the **Delete** operation) that has this property as well. We develop such an algorithm in the next section.

4.3 An Efficient Algorithm for the Real Predecessor Operation

An algorithm for finding the real predecessor must in effect *prove* that a certain key is the real predecessor. Such a proof involves showing that all entries in between a key and its real predecessor in each representative of a read quorum are superseded by a gap with a higher version number in some other representative of the quorum. The number of ghosts between an entry and its real predecessor is potentially unbounded in each representative, so at first the prospects for the existence of an algorithm that requires only a constant number of fixed length messages might appear dim.

However, directory suites have a property that constrains the system states that can occur. Because of this property, the minimum version number necessary for an entry to be current in a region guaranteed to contain the real predecessor can be determined in one round of messages. With this information, a single additional round of messages suffices to find the real predecessor. To state and prove the property that permits this efficient location of the real predecessor, we must introduce several terms.

A *region* is a set of keys; that is, a subset of the key space. A *range* is a region containing every key in the

```
Delete(delk: key);
{ Delete the key delk from the directory }
var
  quorum: array[1..W] of DirRep;
  i: integer;
  isin: boolean
  succ, pred, k: key;
  pval, sval, val: value;
  pver, pgver, sver, sgver, ver: version;

begin
  ILookup(delk,isin,ver,val);  { val ignored }
  if not isin then ReportError;

  { Find the predecessor and successor of delk }
  RealPredecessor(delk,pred,pval,pver,pgver);
  RealSuccessor(delk,succ,sval,sver,sgver);

  { The version number of the coalesced gap must be higher than
    the maximum of any version numbers in the range coalesced }
  ver := Max(ver,pgver,sgver);

  { find a write quorum }
  quorum := CollectWriteQuorum;

  { ensure that the predecessor and successor exist in every
    member of the quorum and coalesce the range in each member }
  for i:= 1 to W do
      DirRepCoalesce(quorum[i],pred,pver,pval,succ,sval,sver,ver+1);
end .
```

Figure 13: Delete Operation

key space between some key and another key. These definitions are consistent with our informal use of the term in previous sections. The notation (k_1,k_2) refers to the range from k_1 to k_2 excluding k_1 and k_2, the *endpoints* of the range.

A gap between entries for keys k_1 and k_2 is said to *cover* the region (k_1,k_2) and all of its subregions (subsets). The remaining terms are defined in the context of an entire directory suite. A gap g is said to be *current over the region* r if the following conditions hold:

1. The gap g covers r.

2. No gap in some other representative covering any non-null subregion of r has a higher version number than g does.

3. No entry in some other representative for a key in r has a higher version number than g does.

Intuitively, a gap is current over a region for which it expresses the most up to date information. A gap's *region of currency* is the entire region over which it is current.[5]

[5] Formally, the union of all regions over which it is current.

For example, consider the suite in Figure 14. Gap g covers ("c",HIGH) and all of its subregions, e.g. ("d","f"). Gap g is current over ("g","k"), for example. Gap g's region of currency is ("c","d")∪("e",HIGH).

We are now ready to state the property.

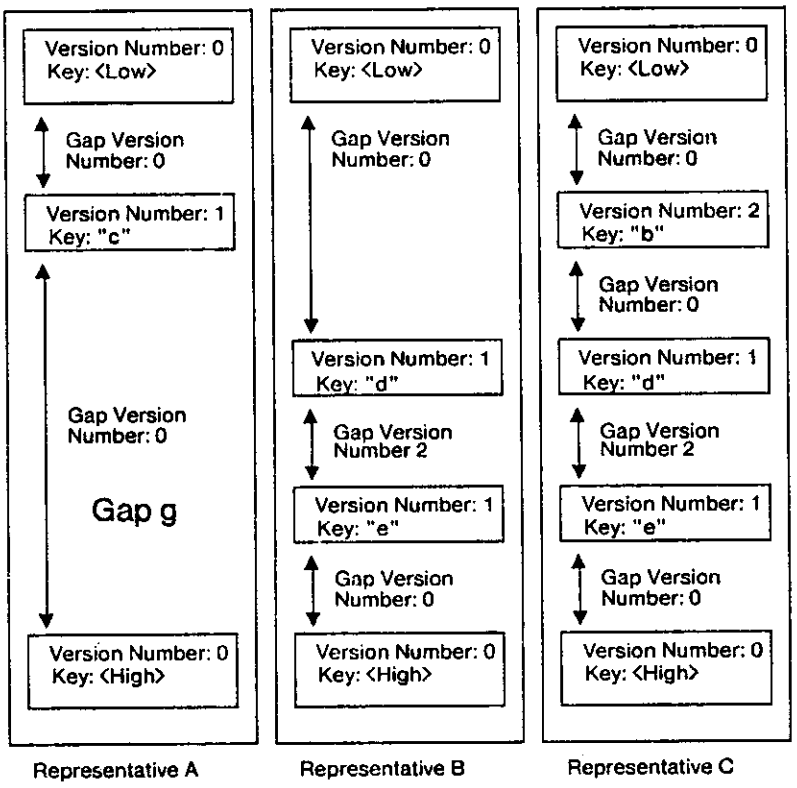

Figure 14: Suite for illustration of *region of currency* and related terminology

Theorem 1. In any occurring system state, every gap's region of currency can be expressed as the union of a finite number of ranges whose endpoints are keys currently in the directory.

Before we can prove Theorem 1 or present the real predecessor algorithm we must introduce one more term and present two lemmas. A collection of ranges $\{r_i|_{i=1}^{n}\}$, $r_i=(k_{i1},k_{i2})$ is said to be *canonical* if the ranges are in order ($k_{i1} < k_{(i+1)1}$) and non-intersecting ($k_{i2} \le k_{(i+1)1}$). The following lemma justifies the use of the term *canonical.*

Lemma 1. For any finite collection of ranges over a dense key space, there exists a unique canonical collection of ranges whose union comprises the same set as the union of the original collection. This is referred to as the *canonical form* of the original collection. (A key space is dense if, for every pair of keys k_1, k_2, with $k_1<k_2$, there exists a key k_3 such that $k_1<k_3<k_2$.) Further, the endpoints of the ranges in the canonical form are all endpoints of some range in the original collection.

Lemma 2. If a gap g is current over a range (k_1,k_2), and k_1 and k_2 are in the directory, then k_1 is k_2's real predecessor.

4.3.1 Proofs

This section may be skipped without loss of continuity. However, it is advised that the reader study the proof of Theorem 1 if a thorough understanding of the internal workings of the data structure is desired.

A rigorous proof of Lemma 1 would be tedious, but a detailed proof sketch follows. If a collection of ranges is not in order, it can be reordered. If any pair of ranges in the resulting collection overlap, their union is a range. Thus the pair of ranges can be replaced by the range that is their union. The endpoints of the union are both endpoints of one of the original ranges. This procedure is repeated until none of the remaining ranges intersect. At this point, the collection is in canonical form, and the union of the ranges in the collection is identical to the union of the ranges in the original collection. Any two canonical collections of ranges over a dense key space that are not identical have different unions, hence the canonical form of the collection is unique.

Lemma 2 follows immediately from the definition of currency over a region. By this definition, if g is current over (k_1,k_2), their are no entries for keys in (k_1,k_2) with a higher version number than g's. Thus there are no keys in between k_1 and k_2 in the directory, and k_1 is k_2's real predecessor.

Now we turn our attention to Theorem 1. We assume that the key space is dense. This assumption is made without loss of generality by the following argument. Any totally ordered set can be embedded in a dense set: given a sparse key space K_s there exists a dense key space K_d such that $K_s \subseteq K_d$. (For example, the integers from 1 to 10 can be embedded in the rationals from 1 to 10.) If we prove that Theorem 1 holds for a key space, we have also proved it for any subset of that key space, as the user could arbitrarily restrict his operations to members of that subset. Thus a proof that Theorem 1 holds for all dense key spaces implies that it also holds for all sparse key spaces. Note that this has no implications with regard to actual system implementation. It merely facilitates the proof.

The proof of Theorem 1 is by structural induction. For the base case, we observe that the theorem holds for a suite in its initial state: each representative contains a single gap whose region of currency is (**LOW**, **HIGH**), and the directory contains the (dummy) keys **LOW** and **HIGH**.

For the induction step, we must show that if the theorem holds for a given system state, then it holds for all states reachable from that state via a single **Insert**, **Update** or **Delete** operation. We shall consider these operations in turn. For each operation, we must show that the gaps contained in the representatives comprising the write quorum and the gaps contained in the representatives outside the write quorum satisfy the required condition after the operation. We further subdivide these gaps into those whose region of currency changes as a result of the operation and those whose region of currency remains unchanged.

First we show that the induction holds for for **Inserts**. The **Insert** operation does not remove any key from the directory, so any range whose endpoints were in the directory prior to the **Insert** will still have its endpoints in the directory after the **Insert**. Therefore, all gaps whose region of currency remains unchanged by the **Insert** will still satisfy the induction hypothesis after the operation (given only that they satisfied it before). Thus, we need only consider the gaps whose regions of currency are altered by the **Insert** operation.

The regions of currency of gaps in representatives outside of the write quorum for an **Insert** operation are affected only if they are current over the region $\{k\}$, where k is the key being inserted. The new entry for this key will have a higher version number than these gaps, so the insertion will have the effect of removing $\{k\}$ from their regions of currency. By hypothesis, the old region of currency of each of these gaps is expressible as a finite union of ranges whose endpoints are keys in the directory. Let us call these ranges $\{r_i|_{i=1}^{n}\}$, $r_i=(k_{i1},k_{i2})$. Lemma 1 allows us to assume without loss of generality that the collection of ranges is in canonical form.

Since the gaps in question contain k in their region of currency, one of the r_i must contain k. Let us call this range r_q. (The value of q may be different for each gap in question.) When $\{k\}$ is deleted from such a gap's region of currency, the resulting region will be

$$\{r_i|_{i=1}^{q-1}\}\cup\{(k_{q1},k),(k,k_{q2})\}\cup\{r_i|_{i=q+1}^{n}\}$$

But k and all of the k_{ij} are in the directory after the insertion, so the induction hypothesis is preserved in all representatives outside of the write quorum.

Within the write quorum one of two things can happen. If an entry is already present for k,[6] no gap's region of currency will be affected by the operation. If no entry for k exists, then the gap g into which the key falls will be split into two new gaps. Let us call them g_1 and g_2. By the induction hypothesis, g's region of currency can be expressed as a finite union of ranges whose endpoints are in the directory. Let us call them $\{r_i\ |_{i=1}^{n}\}$. We assume the ranges are in canonical form, by Lemma 1. If k is in g's region of currency, it is in one of the r_i. Let us call this range r_q. Then g_1's region of currency will be

$$\{r_i|_{i=1}^{q-1}\}\cup\{(k_{q1},k)\},$$

and g_2's region of currency will be

$$\{(k_{q1},k)\}\cup\{r_i|_{i=q+1}^{n}\}$$

(Figure 15). All the endpoints of the ranges comprising g_1 and g_2's regions of currency are in the directory after the insert. If the key being inserted falls outside of the original gap's region of currency, let q be the largest integer such that $k<k_{q1}$. Then g_1's region of currency will be $\{r_i|_{i=1}^{q}\}$ and g_2's region of currency will be $\{r_i|_{i=q+1}^{n}\}$. Thus, the induction hypothesis is preserved in all representatives for **Insert** operations.

---

[6] This entry is necessarily a ghost, as the Insert operation would not be permitted if k were already in the directory.

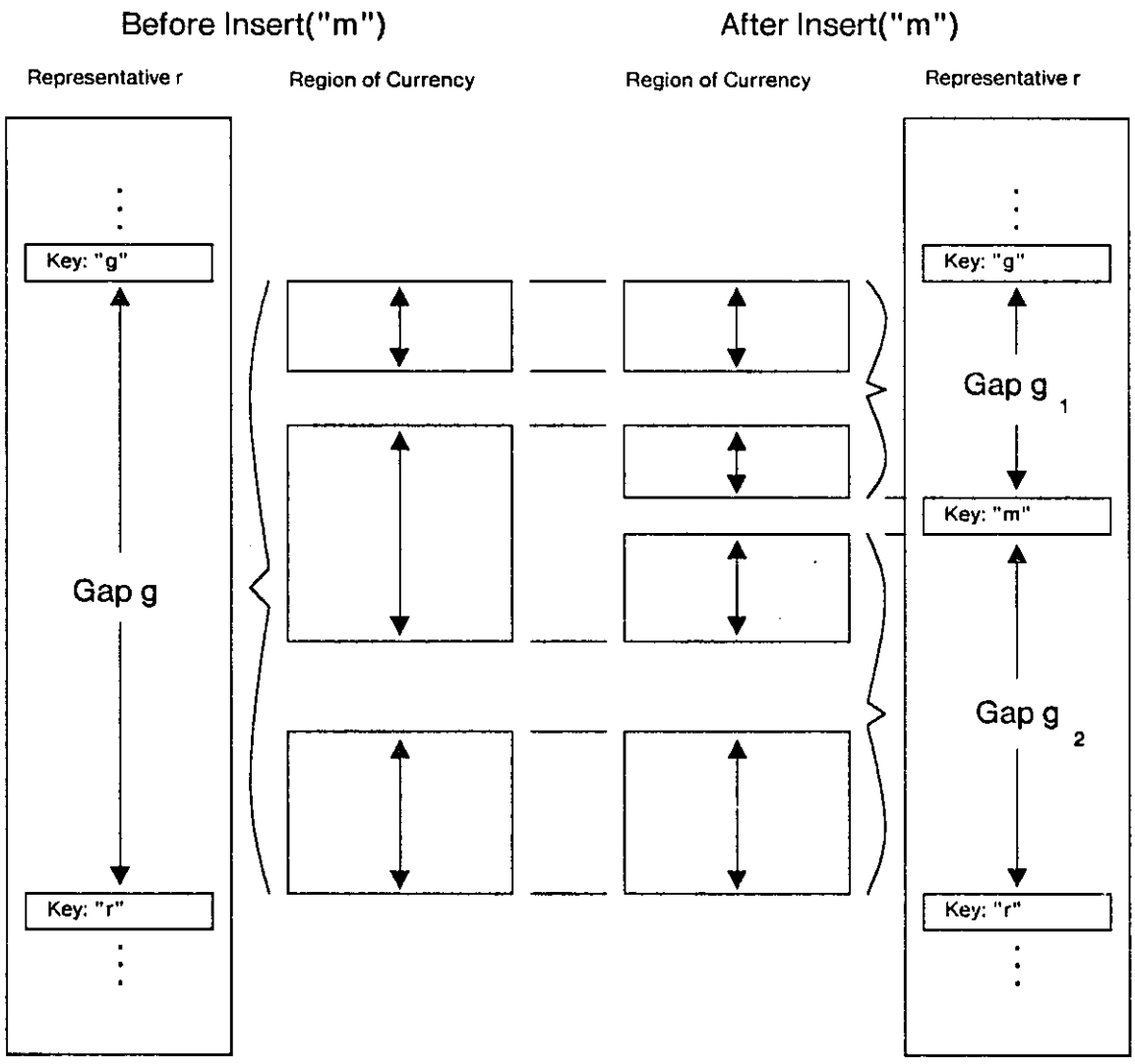

Figure 15: Effect of the **insert** operation on regions of currency, within write quorum

Next we show that the induction holds for **Update** operations. Like **Insert** operations, we need only consider the gaps whose regions of currency are altered by the operation, as **Updates** do not remove any keys from the directory. No gaps in representatives outside of the write quorum have their regions of currency affected by this operation. It increases only the version number associated with the key being updated, k, and no gap could have had $\{k\}$ in its region of currency before the update operation took place. The highest version number associated with k at that time belonged to an entry and not a gap, as updates can only occur on keys that are already in the directory. Within the write quorum the effects of the **Update** operation on regions of currency are identical to those of the **Insert** operation, and the identical argument shows that the induction hypothesis is preserved. (Unlike the case of the **Insert** operation, the key being updated will never fall within a range in a gap's region of currency, because no gap can be current over a range containing a key in the directory.)

Finally we show that the induction holds for **Delete** operations. In each representative in the write quorum, a new gap is created whose region of currency is (p,s), where p is the real predecessor of the key being deleted and s the real successor. If p was not already present in a representative, it is inserted. The region of currency of the new gap extending backward from p consists of the ranges before p previously in the canonical form of the region of currency of the gap from which the new gap was split off. Similarly, if s is inserted, the gap extending forward from s will have as its region of currency the ranges after s previously in the canonical form of the region of currency of the gap from which the new gap was split off. (Figure 16)

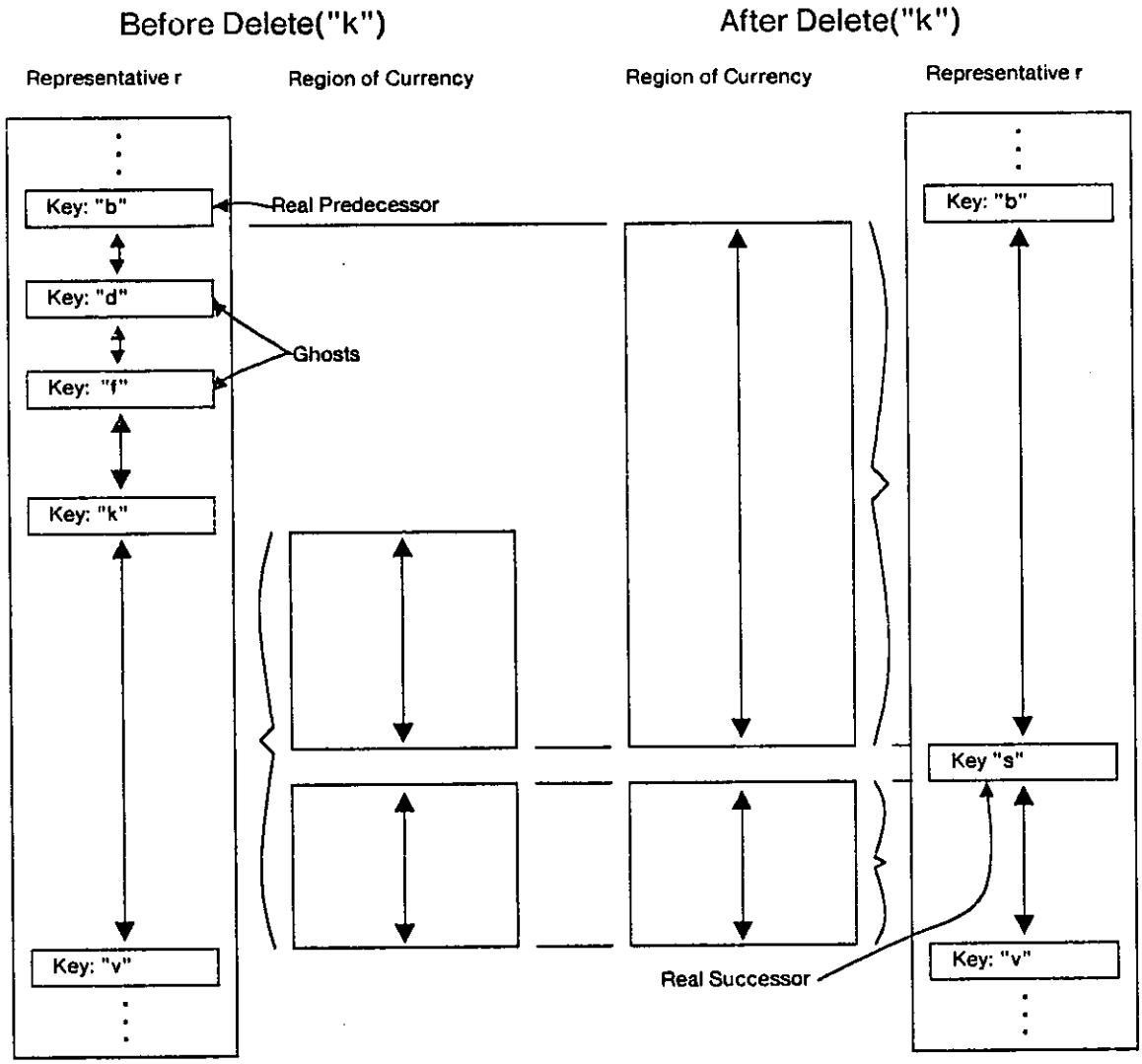

Figure 16: Effect of the delete operation on regions of currency, within write quorum

The keys p and s are, by definition, currently in the directory, so all of the gaps whose regions of currency are modified still satisfy the induction hypothesis.

The gaps whose regions of currency were not modified could not have had any ranges bounded by k in the

canonical forms of their regions of currency. If this were the case, the gaps would of necessity have covered or bordered k. In either case, the deletion of k from the representative would have modified the gaps' region of currency, which, a priori, did not happen. Thus, these gaps satisfy the induction hypothesis given only that they satisfied it before the **delete** and the induction hypothesis holds within the write quorum.

Outside of the write quorum the situation is as follows: The new gap in the representatives of the write quorum covers (p,s). Gaps whose regions of currency did not intersect this region are unaffected. The new gap has a higher version number than all others in this region, so any portions of other gaps' regions of currency that lay in (p,s) are no longer in their regions of currency. Thus, the deletion has the effect of removing ranges entirely contained within (p,s) from the canonical forms of the gaps' regions of currency. By Lemma 2, any range in the canonical form that had k as one endpoint must have had p or s as its other endpoint and so was contained in (p,s). Thus, all ranges remaining in the canonical form after the deletion are bordered by keys other than k that were previously in the directory. But these keys are still in the directory after the deletion, so the induction hypothesis is preserved for gaps outside of the write quorum in a **Delete** operation. This completes the proof.

4.3.2 The Algorithm

In this section we describe our real predecessor algorithm. An argument for the correctness of the algorithm is presented as the algorithm is described. A formal statement of the algorithm is given in Figure 17.

The node determining k's real predecessor asks each representative in a read quorum to return the gap that covers k or has k as its high boundary. All of these "predecessor gaps" cover some range in common, which we call (k_1,k). (k_1 is the highest of the low endpoints of the returned gaps.) Furthermore, the gaps represent information from an entire read quorum so no representative contains any higher version numbered information pertaining to (k_1,k). Thus, the predecessor gap with the highest version number, which we call g_{curr}, is current over the region (k_1, k).

By Theorem 1 and Lemma 1, g_{curr}'s region of currency can be expressed in canonical form as a union of ranges bounded by keys in the directory. Since (k_1, k) is in g_{curr}'s region of currency, it must be contained entirely in one of these ranges. The high end point of this range is k (since k is in the directory prior to the delete operation), and the low end point is k's real predecessor, by Lemma 2. Of course, the low end point must lie within g_{curr} or at its low boundary, which we call k_2.

In the final stage of the algorithm, each representative in the read quorum is asked to return the entry for the highest key less than k within the range of g_{curr} whose version number is higher than g_{curr}'s. If a representative contains no entry in the specified range with a sufficiently high version number, it returns a message to that effect. At this point, two things can happen. If none of the representatives can return an entry, g_{curr}'s low end point, k_2, is k's real predecessor. If one or more such entries exist, the highest key for which an entry is returned, p, is k's real predecessor, by the following argument.

All keys for which entries are returned must lie outside g_{curr}'s region of currency. Thus, p lies outside of g_{curr}'s region of currency. Therefore, the key that delimits the range in the canonical form of g_{curr}'s region of currency in which (k_1,k) lies must be $\geq p$. But no key between p and k, exclusive, is currently in the directory; if there were such a key, at least one of the representatives in the write quorum would have contained a current entry for it, which it would have returned in the final stage of the algorithm. Thus p is k's real predecessor, and the highest version numbered entry returned for p is current.

4.3.3 Enhancements to the Real Predecessor Algorithm

As in the other procedures presented, efficiency is sometimes sacrificed for clarity in the **RealPredecessor** procedure of Figure 17. There are several additional improvements that would be made in any practical implementation of the algorithm. The procedure would check if the second round of information exchange were necessary before doing it. If the highest predecessor key returned in response to the first request for information has a higher version number than any of the returned gaps that cover it, then this key must be the real predecessor, and there is no need to continue searching.

This technique can be used to reduce message traffic even further by having each representative return several gaps and entries preceding the key being deleted rather than just one. The procedure would check if any key for which it had information (entry or covering gap) from all representatives had a higher version number than any covering gap. If this is the case, then the highest such key is the real predecessor, and no second stage is necessary. The number of entries returned by the representatives in the first stage of the algorithm controls a performance trade off between execution time at the nodes and inter-node message traffic. If many entries are returned, it is likely that the second round of information exchange will not be necessary; however, the execution time at each node is proportional to the number of entries sent. The number of entries between the key being deleted and its real predecessor will on average be half of the key's delete list size. (Recall that the delete list consists of all of the entries between a key's real predecessor and its real successor.) Thus, the formula developed in Section 5.2.4 that enables us to predict the average length of a delete list can aid in choosing an appropriate number of entries to return in the first stage. In fact, the limiting behavior described in Section 5.2.5 shows that that the second stage of the algorithm can almost always be avoided if several entries are returned in the first stage.

Even if the second stage is required, it may not be necessary to ask for additional information from all of the representatives in the read quorum. Any representative that has already sent entry or gap information for the entire range that has been determined to contain the real predecessor (the range covered by g_{curr}) has no more information to add and need not participate in the second round.

The real predecessor and real successor can be determined simultaneously by putting requests and responses for both tasks in each message, thus reducing by almost one-half the message traffic required to find the real predecessor and successor. In the actual implementation, there would be a single

```
RealPredecessor(IN k:key; OUT pred:key,pval:value,pver,gver:version);
{Returns the key, value and version number of k's real predecessor,
and the highest version number in the range bounded by k and k's
real predecessor, exclusive.}
var quorum: array[1..R] of DirRep,
    MaxGapVer,CandGapVer,CandKeyVer: version,
    MaxGapKey,CandKey: key,
    CandKeyVal: value;
    MaxGapRep: integer;
    CandFlag: boolean;
begin
  quorum := CollectReadQuorum();

  {Get info on predecessor gaps in each rep in the read quorum & find
   out which rep has the gap w/ the highest version number. (g-curr)}
  MaxGapVer :=  LowestVersion - 1;  {A Constant}
  for i := 1 to R do
    begin
     DirRepPredecessor(quorum[i],k,CandKey,CandKeyVer,CandKeyVal,
                       CandGapVer);
     if CandGapVer > MaxGapVer then
       begin
         MaxGapVer := CandGapVer;
         MaxGapKey := pred := CandKey;
         pver := CandKeyVer;
         pval := CandKeyVal;
         MaxGapRep := i;
       end
    end;

 {Find closest entry which supersedes g-curr in any rep in the
  read quorum. This will be the real predecessor. }
 for i := 1 to R do
    if i <> MaxGapRep then
      begin
        DirRepSuperseder(quorum[i],k,MaxGapVer,MaxGapKey,
                         CandFlag,CandKey,CandKeyVer,CandKeyVal);
        if CandFlag  {If this rep has a candidate for real pred...}
          {and it's closer than the closest candidate thus far, or
           equally close with a higher version number then...}
          and (CandKey > pred
             or (CandKey = pred and CandKeyVer > pver)) then
          begin  {Tentatively select the candidate}
            pred := CandKey;
            pval := CandKeyVal;
            pver := CandKeyVer
          end
      end
end
```

Figure 17: Real Predecessor Operation

"RealNeighbors" procedure instead of separate **RealPredecessor** and **RealSuccessor** procedures. The procedure would initially ask for gaps and entries surrounding the key on both sides. If this did not provide enough information to find the key's real predecessor and successor, it would send a request for a "superseder" of either or both "current gaps," as required.

In the procedure for the **Delete** operation in Figure 13, the key to be deleted is looked up prior to determining its real neighbors. In practice the lookup would be combined with the first stage of the real neighbors determination.

The critical factor determining the execution speed of the directory operation procedures presented is the number of small, fixed length messages sent in performing the operation. Thus we use this number as a complexity measure for our algorithms. Our real predecessor algorithm, with the improvements described, is extremely fast in the average and worst cases. The average performance of this algorithm is close to the trivial lower bound of one exchange of messages with each member of a read quorum. The worst case performance is two rounds. The **Delete** operation requires one additional round to coalesce the range between the real predecessor and successor.

The procedure, including the improvements, is easy to implement. It also has the following useful property. The correctness of the algorithm does not depend on the fact that the key whose real predecessor is being determined is actually in the directory. Thus, one can locate the real neighbors of any key, regardless of whether it is in the directory. This could, for instance, be used to implement a "range delete" operation, which deleted all of the keys between one key and another. This operation would require no more message transmissions than the deletion of a single key.

4.4 Correctness Arguments

The correctness of a directory suite's operations depends on **Lookup** always returning current information about a key. Because every read quorum intersects every write quorum, **Lookup** will return current information as long as that information has a version number greater than that of any non-current information. These correctness conditions are the same as those required for Gifford's file replication algorithm.

Two phase locking and the lock compatibility matrices specified in Section 4.1 are strong enough to guarantee the serializability of transactions at any single representative. Traiger et al. [Traiger et al. 82] have shown that if all nodes participating in a distributed transaction execution follow two phase locking protocols that guarantee the serializability of transactions at individual nodes, then the resulting global schedule is equivalent to some serial schedule of transactions. Thus, the directory replication algorithm preserves the serializability of transactions that use it.

The **Insert** and **Update** operations both set the version number of the entries they modify to be greater than the greatest version number previously associated with the keys of those entries. Therefore, the current data for each key has a version number greater than that of any non-current data for that key.

Delete coalesces the range between the real predecessor and real successor of the key to be deleted. By the definitions of real predecessor and real successor, there can be no current entries (other than the entry to be deleted) in the range to be coalesced. The operation assigns to the gap covering the coalesced range a new version number that is higher than any version number previously associated with any key in that range. Therefore, as with **Insert** and **Update**, the current data for each key in the range has a version number greater than that of any non-current data for that key.

4.5 More on Synchronization and Recovery

Directory representatives, as described in Section 4.1, are synchronized to ensure that all transactions using their operations can be made serializable.[7] In addition, all information in a representative is recoverable and operations can be completely redone or undone by recovery processing. Thus, arbitrary directory representative operations may be composed in atomic transactions. This property simplifies the correctness arguments for the directory replication algorithm by allowing the algorithm to ignore the consequences of concurrency and failures during directory suite operations. However, the use of directory representative operations is not arbitrary, and the restrictions that the directory replication algorithm imposes on their use can be exploited to enhance the synchronization and recovery performance of directory representatives. The resultant directory representative objects may show non-serial behavior [Schwarz and Spector 84] if they are used outside of this directory replication algorithm.

The basis for improvements to concurrency and simplification of recovery in **Delete** is Gifford's observation [Gifford 81] that data and its version number in one representative may be replaced at any time by more current data with a higher version number from another representative. It is easy to see that the contents of the directory, as observed by the results of **Insert**, **Update**, **Delete**, and **Lookup** operations are unaffected by such a replacement. Of course, care must be taken to prevent an independently executing update from being overwritten with the data and version number from the other representative. A temporary write lock on the data being replaced is sufficient concurrency control for this purpose.

Improvements to concurrency and recovery can be accomplished with modifications to **DirRepCoalesce**. The **Delete** operation is the only invoker of **DirRepCoalesce**, and it always passes the real predecessor and real successor of a key to be deleted as arguments; therefore the only current entry modified by **DirRepCoalesce** is the entry being deleted from the directory. To increase concurrency and simplify recovery, the **DirRepCoalesce** operation can be redefined to take three additional arguments. The first new

---

[7] For these transactions to be serializable, all other types of objects used by the transaction must also preserve serializability.

argument is the key of the entry being deleted. If the transaction performing the **DirRepCoalesce** is aborted this key is used to determine the entry that must be restored. When the **DirRepCoalesce** operation is undone, the gaps on either side of the entry being deleted receive the current version numbers for those gaps, which are determined along with the real predecessor and real successor and passed as the second and third additional arguments to **DirRepCoalesce**. It is unnecessary to restore any ghost entries during the undo of a **DirRepCoalesce** operation.

Concurrency can be increased by releasing the **RepModify** locks set by **DirRepCoalesce** on all keys, except for the key of the entry actually being deleted, as soon the operation completes. The locks do not need to be retained, because the operation does not modify data other than version numbers in these gaps, and version numbers are used in very well defined ways by the weighted voting algorithm.

Additionally, **RepLookup** locks on data less than the real predecessor and greater than real successor of a key being deleted need not be held beyond the first phase of the **RealPredecessor** and **RealSuccessor** operations. These locks are obtained only to guarantee that the algorithm for determining the real predecessor and successor sees a consistent version of the directory suite.

5 Performance Characterization

In this section, we present the results of simulations and construct and analyze a model of the algorithm as applied in the simulations. The system studied in the simulations and the model consists of a directory suite initially containing a certain number of keys into which **Inserts**, **Updates** and **Deletes** occur sequentially with equal likelihood. The keys to be inserted are chosen randomly from those not in the directory, and the keys to be updated or deleted are chosen randomly from those in the directory. Read and write quorums are selected randomly. (**Lookups** are not performed as they have no effect on the contents of the directory suite.)

The key space used in the simulations consists of the integers from one to one billion. The mathematical model is described in sufficient generality to apply to any finite key space. It does not make sense to consider the system with an infinite key space, as keys to be inserted are chosen at random from those not already in the directory. If an infinite key space were used, this would amount to selecting an object at random from an infinite set, an operation which is not well defined. Interestingly, the cardinality of the key space does not affect the analysis except insofar as it affects the validity of several simplifying assumptions. This fact is discussed at greater length in Section 5.2.6.

Various performance measures can be used to evaluate the performance of our algorithm. In our view, the most important performance measure is the number of rounds of message exchanges with a read or write quorum necessary to perform each directory operation. With one exception, this measure is a constant which does does not vary from instance to instance of a given operation. The exception is the **Delete** operation, which, with the suggested enhancements, requires either two or three rounds of messages depending on the

results of the first round. (See Section 4.3.3.) The communications cost of the directory operations are summarized in Figure 5.

Operation	Rounds to Read Quorum	Rounds to Write Quorum	Total # Rounds	Total # Messages
Insert	1	1	2	2(R+W)
Update	1	1	2	2(R+W)
Delete	1 or 2	1	2 or 3	2(R+W) or $\leq$ 2(2R-1+W)
Lookup	1	0	1	2R

Figure 18: Communications Costs of Directory Operations

The node doing a directory operation has to send RPC's to read and write quorums and, in the case of read quorums, scan the responses to determine the current information. Thus the work done by this node is proportional to the total number of messages required for an operation, and is generally small. More interesting is the work done by the nodes storing the directory representatives. All directory representative operations except for the second and third steps of the **Delete** operation amount to either looking up or updating the information associated with a single key or gap. The time required to perform this operation depends on the number of entries in the representative and data structure used to store the entries. If balanced trees are used, the time is proportional to the log of the number of entries. The storage space required at each representative is proportional to the number of entries stored at the representative.

Thus, the first performance measure we concentrate on in our performance studies, which we call the *size ratio*, is the ratio of entries in a directory representative to keys in the directory. The size ratio indicates the storage required at each representative as a function of the storage required for a single site directory. A size ratio of one indicates that a node has exactly as many entries as a single site directory containing the same keys. The simulations measure the size ratio directly, while the analytic model allows us to break the size ratio down into three *composition ratios* based on a classification of directory entries into three categories. The size ratio is the sum of the three composition ratios.

In the second step of the **Delete** operation (**DirRepSuperseder**) each representative has to scan the delete list for the key being deleted. In the third step (**DirRepCoalesce**), each representative has to coalesce the delete list into a single gap. In both steps, the total work required is proportional to the delete list length. Thus, the second performance measure we study in our simulations and analysis is the *average delete list length.* The average is taken over all keys in the directory.

As explained in Section 4.3.3, the second step of the **Delete** operation is necessary only if one or more nodes in the read quorum did not return their entire delete list in the first step. Thus knowing the expected value of the average delete list length allows us to ask for enough information in the first step so that the second step will usually be unnecessary. Of course this would not be feasible if the expected value of the average delete list length were high. However, this turns out not to be the case.

In summary, the size ratio characterizes the space complexity of our algorithm. The size ratio and average delete list length characterize the significant components of the time requirements of our algorithm. Knowledge of the average delete list length is useful in ensuring that the first round of the delete operation returns enough data so that the second round in unnecessary. The size ratio and average delete list length are the performance measures that form the basis of our performance studies. In the analysis, the size ratio is further subdivided into composition ratios which tell us more about how the storage space is being used.

5.1 Simulation Results

The shaded bars in Figures 19 and 20 show the size ratios and delete list lengths measured in simulations for a variety of directory configurations. (The unshaded bars show predicted values obtained from the mathematical model in Section 5.2.) In the simulations, each directory suite initially contained one thousand entries. The duration of each simulation was twenty thousand operations, and performance measures were gathered during the final ten thousand operations.

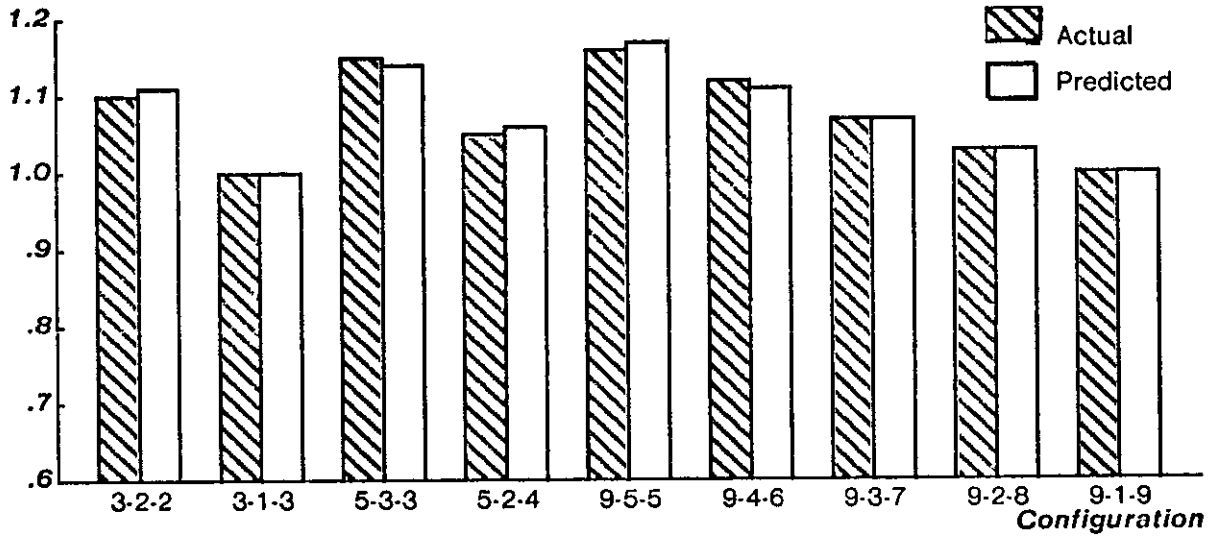

Figure 19: Size Ratios for Various Directory Suites

The simulation results in Figure 19 show that the size ratio remains very close to one for all of the suite configurations tested. Thus the storage required at each representative and the time required to locate an entry at a representative are only slightly higher than for a single site directory. The results in Figure 20 show that the average delete list length is less than a single entry for every configuration tested. This implies that the second and third steps of the Delete operation will run very quickly at the representatives, and the second step will rarely be necessary if a few entries are returned in the first step.

More detailed simulation results for 3-2-2 directory suites with one hundred, one thousand, and ten thousand keys initially in the directory are shown in Figure 21. The duration of each of these simulations was

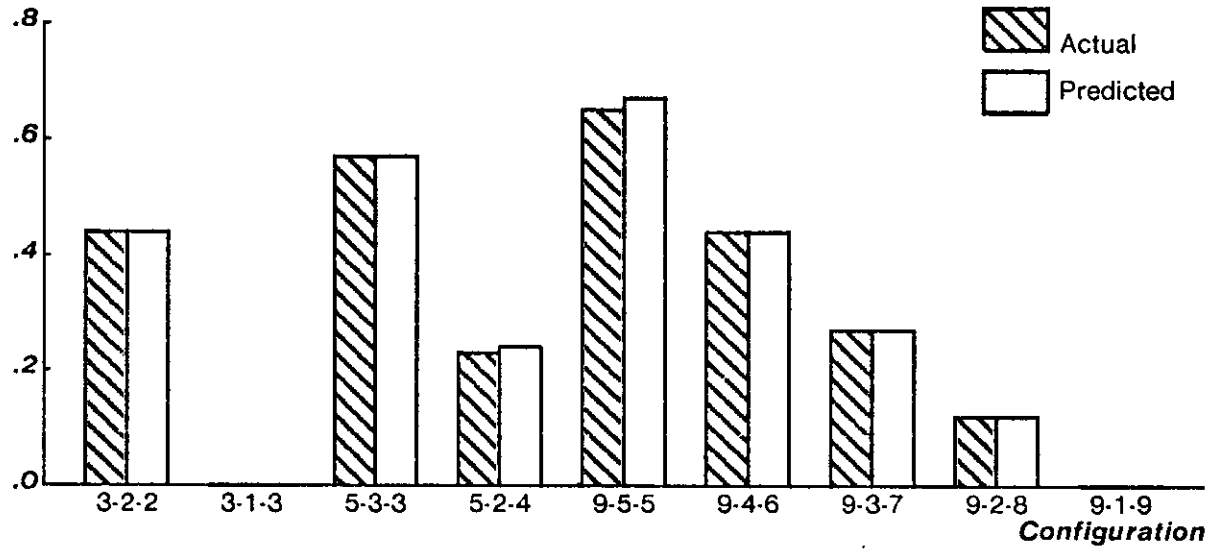

Figure 20: Delete List Lengths for Various Directory Suites

two hundred thousand operations, with performance data gathered during the final one hundred thousand operations.

100 Keys			1000 Keys			10000 Keys		
Size Ratio								
Avg	Max	Std Dev	Avg	Max	Std Dev	Avg	Max	Std Dev
1.11	1.27	0.03	1.11	1.19	0.02	1.11	1.13	0.01
Delete List Size								
Avg	Max	Std Dev	Avg	Max	Std Dev	Avg	Max	Std Dev
0.44	9	0.81	0.44	9	0.81	0.44	10	0.81

Figure 21: Detailed Simulation Results for three 3-2-2 Directory Suites

These additional simulations indicate that the average values of the performance measures do not depend on the initial number of keys in the directory suite. Thus, average space requirements appear to be proportional to the number of keys in the directory, just as in a single site directory. The time requirements depend on the number of keys in the directory in the same manner as for a single site directory. The standard deviation of the size ratio decreases as the number of entries increases. This is easily explained by the fact that the numerator and denominator of the size ratio are the number of entries in a representative and the number of keys in the directory, respectively. Similar variation should be observed in both of these random processes regardless of the directory size, but a given variation in the numerator or denominator will cause a greater change in the fraction if the denominator (the number of keys in the directory) is large.

The maximum delete list size observed was 10. This is an indication of the *worst case* time to perform the directory representative operations for the second and third steps of the delete operation. Care should be

taken not to interpret this as the true worst case time for any possible run. Theoretically, a delete list can be as long as the number of keys that have ever been deleted from the directory. The longer a run, the higher the maximum observed delete list is likely to be. However, the fact that the largest delete list observed in three runs of one hundred thousand operations each was only 10 entries indicates that large delete lists will probably not be a problem in practice.

5.2 Analytic Model

The algorithm as applied in the simulations was modeled and analyzed to predict various performance characteristics. The goals of the analysis were to increase our confidence in the simulations by corroborating their results, to gain further insight into the behavior of the algorithm, and to produce a fast, reliable method for determining the performance of the algorithm in a given application.

In this section, we describe the model and our method of analysis, and present the analysis. A set of formulae to predict performance characteristics are derived in the analysis. These formulae are used to check the results obtained from the simulations and predict performance trends exhibited by the algorithm under various conditions.

5.2.1 Construction of the Model

The system can be modeled as a Markov chain in a straightforward fashion. One state corresponds to each possible contents of the entire directory suite, henceforth called a *system state*. The transitions correspond to the changes in system state effected by the operations. Transition probabilities are induced by the fact that the operation to be performed (**Insert**, **Update**, or **Delete**), the key to be operated upon, and the write quorum are chosen at random.

In the simulations, the system appeared to display equilibrium behavior: each system attribute being monitored approached an average value that did not vary over multiple runs of sufficient length. For a Markov model to be of use to us in calculating these values, it too must display this equilibrium behavior. It is sufficient that the model achieve stochastic equilibrium. The simplest class of Markov chains achieving stochastic equilibrium are those that are finite and irreducible. (By *finite*, we mean that they contain a finite number of states, and by *irreducible*, we mean that each state can be reached from every other state.)

The straightforward model described above does not possess either of the requisite properties. It is not finite, as version numbers can grow without bound. Repeatedly updating a single key produces an infinite sequence of distinct states. Neither is the straightforward model irreducible: once the system leaves *any* state, it can never get back to that state. This can be seen by observing that the version numbers associated with a fixed key in a fixed representative in successive states form an increasing sequence. Any operation results in the version number associated with some key increasing in some representative and it can never return to its original value. However, the model displays an extremely high degree of *lumpability* [Kemeny and Snell 60].

That is to say, many states are practically identical to some other state, so sets of similar states can be lumped together to produce a smaller, simpler model. We shall attempt to construct a new model that possesses the desired properties by this process of lumping.

This is not the straightforward task that it might appear to be. The obvious way to deal with the fact that version numbers increase without bound is to equate states with identical ordering of pairs of keys by version number, thus eliminating the absolute version numbers. However, attempts to lump states based on order relations between version numbers alone run into complications. Even if such an attempt succeeded, the model produced might well be finite but not irreducible. An alternative approach, which involves abandoning the version numbers entirely, produces the desired result. Before we describe it, we must take care of some preliminaries.

All of the entries in each representative of a directory suite can be divided into three classes that correspond to terms introduced in previous sections. A *current* entry is an entry for a key that is still in the directory that has highest version number associated with that key in any representative. Current entries are the only entries that contain up to date information. An *outdated* entry is a non-current entry for a key that is still in the directory. If an entry is outdated then some other representative contains an entry for the same key with a higher version number. A *ghost* entry is an entry for a key that is no longer in the directory suite. A ghost entry can be thought of as the ghost of a key that used to "live" in the directory. It should be clear that all entries in a representative fall into one and only one of these classes.

Let us call a representative with all version numbers removed and with the class of each entry (current, outdated or ghost) appended to the entry the *concise representation* of the representative. Note that the concise representation contains no explicit information about the gaps between entries. By extension, we call the collection of concise representations of all representatives in a suite the concise representation of the suite. The concise representation has two properties that make it useful:

1. Given the concise representation of a system state, an operation to be performed on the suite (**Insert**(key), **Update**(key) or **Delete**(key)) and the write quorum selected for the operation, one can determine the concise representation of the resulting system state.

2. All of the important information concerning a system state is fully determined by its concise representation; that is, all system states sharing a concise representation coincide in all *important attributes*. By *important attributes*, we mean the performance measures: delete list length and composition ratios, and several other attributes for which we assert that equilibrium distributions exist in the analysis of our model.

The proof of Property 1 is a somewhat tedious case analysis, which is implicitly performed for other reasons in Appendix I. The intuition behind the proof is that version numbers are used solely to find out which class an entry belongs to, when performing the various operations on the suite.

Property 2 must be proven separately for each *important attribute*. It is true for the composition ratios, as

the concise representation of a representative clearly contains the same number of current, outdated and ghost entries as the representative itself. It is true for average delete list length, as delete lists consist of all of the ghost entries between two keys in the directory, and system states sharing a concise representation represent the same directory, and have ghost entries for the same keys at corresponding representatives. The reader can easily check that this property holds for all other system attributes on which we assert the existence of an equilibrium distribution in the analysis.

We are now ready to describe the method by which we simplify our model. We define a new model where all system states sharing each concise representation are lumped together to form the model states. Property 1 above tells us that the induced transition probabilities in this model are well defined. This is required for the model to be a well defined Markov chain.

The new model is finite by the following argument. The key space is finite, and each representative contains entries for some subset thereof. Each entry belongs to one of the three classes; thus, there are only a finite number of possible concise representations for representatives. A suite consists of a fixed number of representatives, so there are only a finite number of possible concise representations for system states. This places a finite an upper bound on the number of states in our model.

The model is irreducible by the following argument. From any system state, it is possible to reach a system state where all representatives contain no entries. This can be accomplished as follows: first delete all of the keys in the directory in any order with any write quorums. At this point, all of the representatives can only contain ghost entries, and if a single key is inserted into the directory and then deleted using the same write quorum, all of the representatives in the quorum will be completely empty. Repeat this insert/delete process as many times as necessary to include each representative in at least one write quorum. All system states where none of the representatives contain any entries have the same concise representation hence they are represented by a single state in the model. But this state also represents the initial system state, from which all other system states can be reached. Thus, any model state reachable from the initial state can be reached from every state.

The Markov model achieves stochastic equilibrium, because it is finite and irreducible. There is one other property that the model must have in order to fulfill our requirements: it must not lump together system states that are not really equivalent. In other words, all system states represented by each model state must be functionally identical in the sense that they coincide in all attributes for which we wish to infer the existence of an equilibrium distribution. However, this is precisely what Property 2 tells us.

5.2.2 Method of Analysis

Our model is guaranteed to achieve stochastic equilibrium, so it is theoretically possible to determine the precise probability of being in any state. In practice, this would be impossible due to the huge size of the system. Also, the resulting probability distribution would not be particularly informative as such, and the processing necessary to derive any useful figures from it would be prohibitive due to its size. However, the existence of this model proves that any attributes common to all system states represented by each state of the model have well defined average values. Thus it makes sense to formulate relationships among such averages and solve for them.

The performance characteristics of primary concern to us are all intimately related to the composition of each representative in terms of the three classes into which entries are divided. As a consequence of the existence of our model we can assert that a dynamic equilibrium exists in each of these classes in each representative. These assertions can take the form of *balance equations* equating the rates of flow into and out of each category in a single representative. Such equations hold equally well for all of the representatives in the suite due to the symmetry of the system. In the course of the analysis, we focus our attention on a single representative, but the results apply to every representative in the suite.

These balance equations are naturally constructed in terms of three independent variables, and the system parameters N and W (defined in Section 5.2.3). In constructing the balance equations, we make some simplifying assumptions in the form of approximations in the equations. Each approximation will be noted and justified. The resulting equations constitute a linear system than can be solved easily. Expressions for the desired performance measures can be constructed from the independent variables, though we need to make a simplifying approximation in one derivation.

5.2.3 Formulation of Balance Equations

The following variables are used in formulating the balance equations. Script capitals represent stochastic variables, small letters represent unknowns in the balance equations, and capital letters represent constants (system parameters).

$\mathcal{C}$ The number of current entries in the representative under observation.

$\mathcal{O}$ The number of outdated entries in the representative under observation.

$\mathcal{G}$ The number of ghost entries in the representative under observation.

$\mathcal{E}$ The total number of entries in the representative under observation. Note that $\mathcal{E} = \mathcal{C} + \mathcal{O} + \mathcal{G}$.

$\mathcal{K}$ The number of keys currently in the directory.

$\mathcal{D}_k$ The number of entries in the *delete list* of a key k currently in the directory, in the representative under observation. (The delete list of a key consists of all of the ghost entries between the *real predecessor* and *real successor* of the key in a representative.)

$\mathcal{D}$ $(\sum^{k \in \mathrm{Dir}} \mathcal{D}_k)/\mathcal{K}$. $\mathcal{D}$ is the average delete list size in the representative under observation. Note that $\mathcal{D}$ is only defined in states where $\mathcal{K} \neq 0$ (i.e. the directory contains one or more keys).

c' $E[\mathcal{C}/\mathcal{K}]$ The expected value is taken over all states that represent directories containing one or more keys. $\mathcal{C}/\mathcal{K}$ is the fraction of keys in the directory that have current entries in the representative under observation. Thus, c' is equal to the probability that a randomly chosen key in the directory has a current entry in the representative under observation.

o' $E[\mathcal{O}/\mathcal{K}]$ The expected value is taken over all states that represent directories containing one or more keys. $\mathcal{O}/\mathcal{K}$ is the fraction of keys in the directory that have outdated entries in the representative under observation. Thus, o' is equal to the probability that a randomly chosen key in the directory has an outdated entry in the representative under observation.

d $E[\mathcal{D}]$ The expected value is taken over all states that represent directories containing one or more keys. d is the expected size of a delete list for a key chosen at random from those in the directory.

N The number of representatives in the directory suite being modeled.

W The write quorum size for the directory suite being modeled.

A formal statement of the rate balance assertion for current entries is:

E[The number of entries entering the current class in a chosen representative in one operation]
= E[The number of entries leaving the current class in a chosen representative in one operation].

The expected values are computed over a space consisting of all the state transitions in our model. Analogous assertions are made for outdated and ghost entries. The expected values can be recast in terms of c', o' and d. These expansions, though relatively straightforward, are somewhat tedious, as they entail examining the inner workings of the directory suite operations in great detail. They can be found in Appendix I.

The expansions yield the following balance equations, for current, outdated and ghost entries respectively:

$$(N+W)c' + Wo' = 2W$$

$$o' = \frac{N-W}{N+W}c'$$

$$d = \frac{N-W}{W}(c'+o').$$

5.2.4 Solution of Balance Equations

The solution to the balance equations derived in the previous section is:

$$c' = \frac{2W(N+W)}{N(N+3W)}$$

$$o' = \frac{2W(N-W)}{N(N+3W)}$$

$$d = \frac{4(N-W)}{N+3W}.$$

The first performance measure for which we desire a formula is the expected value of the average delete list size:

$$E[\mathfrak{D}] = d.$$

The second performance measure is the expected value of the size ratio:

$$\begin{aligned} &E[\mathfrak{S}/\mathfrak{K}] \\ &= E[(\mathfrak{C}+\mathfrak{O}+\mathfrak{G})/\mathfrak{K}] \\ &= E[\mathfrak{C}/\mathfrak{K}] + E[\mathfrak{O}/\mathfrak{K}] + E[\mathfrak{G}/\mathfrak{K}] \\ &= c' + o' + E[\mathfrak{G}/\mathfrak{K}]. \end{aligned}$$

The three terms of this expression ($E[\mathfrak{C}/\mathfrak{K}]$, $E[\mathfrak{O}/\mathfrak{K}]$ and $E[\mathfrak{G}/\mathfrak{K}]$) are the composition ratios. While we cannot exactly express the third term of this expression in terms of our unknowns we can make a very good approximation based on the fact that almost every ghost in a representative appears in two delete lists, that of its real predecessor and that of its real successor. The exceptions are the ghosts before the first key in the directory and those after the last, which only appear in a single delete list. But in the vast majority of states,

very few ghosts fall into this category. Thus the sum of the sizes of all delete lists in a representative is approximately equal to twice the number of ghosts. A formal statement of this assumption is:

$$2\mathcal{G} = \sum_{k \in \mathrm{Dir}} \mathcal{D}_k.$$

Dividing both sides of this equation by $2\mathcal{K}$ and taking expected values over all states representing non-empty directories, we get:

$$E[\mathcal{G}/\mathcal{K}] = E[(\sum_{k \in \mathrm{Dir}} \mathcal{D}_k)/2\mathcal{K}]$$

$$= \frac{1}{2} E[\mathcal{D}]$$

$$= \frac{d}{2}.$$

Substituting back, our formula for the size ratio becomes:

$$E[\mathcal{S}/\mathcal{K}] = c' + o' + \frac{d}{2}.$$

$$= \frac{2(N+W)}{N+3W}$$

5.2.5 Results

Figure 19 (p. 32) compares the average size ratios observed in the simulations with predictions obtained from the formula developed in the previous section. Figure 20 (p. 33) compares actual and predicted average delete list lengths. The predicted values are nearly identical to the observed values. We compared simulation and analysis results for many other system attributes and observed this level of agreement uniformly.

Figure 22 shows the predicted average composition ratios in a $20 \cdot (21 - W) \cdot W$ suite, for all possible values of W. Figure 23 shows predicted delete list lengths for these suites. Varying the quorum sizes in a fixed size directory suite in this manner controls a fairly complex performance tradeoff: increasing the write quorum size increases the availability of the read operation while decreasing its cost, and decreases the availability of the write operation, increasing its cost. In the delete operation, the work done at each node decreases, but the number of messages that must be sent increases. At one end of the spectrum ($W=20$) there is the universal update strategy; at the other ($W=11$), there is a strategy where roughly half the representatives are written and half are read. Note that in the universal update strategy, the size ratio is 1 and there are no outdated or ghost entries, as the representatives are just copies of the single site directory. The graphs show that for the spectrum under investigation, the representatives contain at worst 20% more entries than a single site directory and the average delete list size remains shorter than a single entry.

Figures 24 and 25 show respectively the predicted average composition ratios and delete list lengths in $(2i-1) \cdot i \cdot i$ suites. Increasing read quorum, write quorum and suite sizes simultaneously, as illustrated in

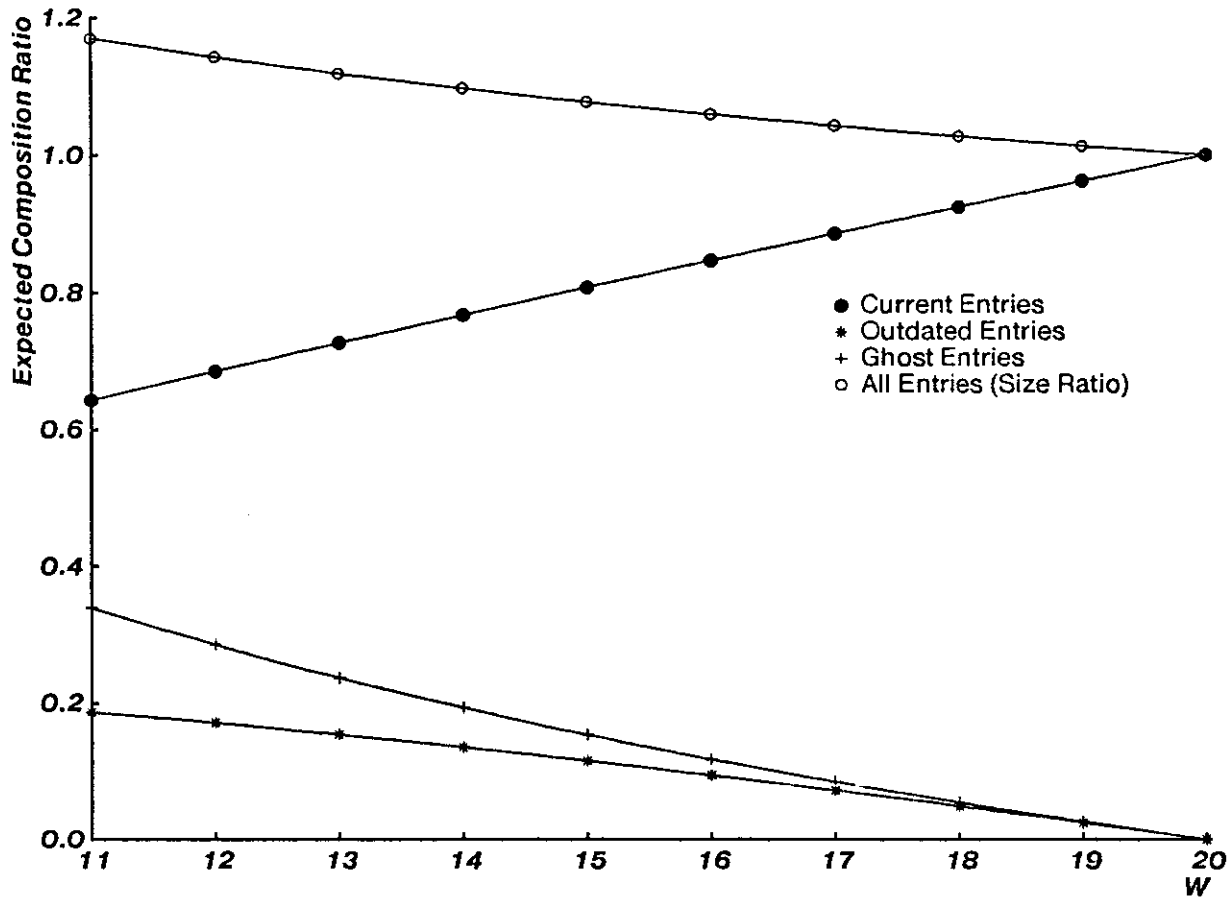

Figure 22: Expected Composition Ratios in a 20 - $(21-W)$ - W Suite

these graphs represents a fairly straightforward performance tradeoff: As the sizes increase, the availability of the system increases, but the number of messages that must be transmitted for all operations increases as well. Specifically, the number of representatives that can be destroyed while still maintaining availability in a $(2i-1)$ - i - i suite is $i-1$. The flatness of the curves shows that the amount of work at each node in a **Delete** operation, and the size and makeup of each representative do not vary appreciably over the spectrum. Thus the cost scales up proportionately to the increased availability with no added penalty for very high availability.

Finally, we present some fairly surprising results concerning the limiting behavior of the performance measures. First let us examine the expected length of a delete list, *d*. Recall, the formula for d is:

$$\frac{4(N-W)}{N+3W}.$$

Let us maximize it subject to the (real) constraints that $N \geq 1$ and $\frac{N}{2} \leq W \leq N$. As we would expect, this expression grows as the write quorum decreases. Thus the expression achieves its maximum when W is set to $\frac{N}{W}$ its lowest permissible value. So:

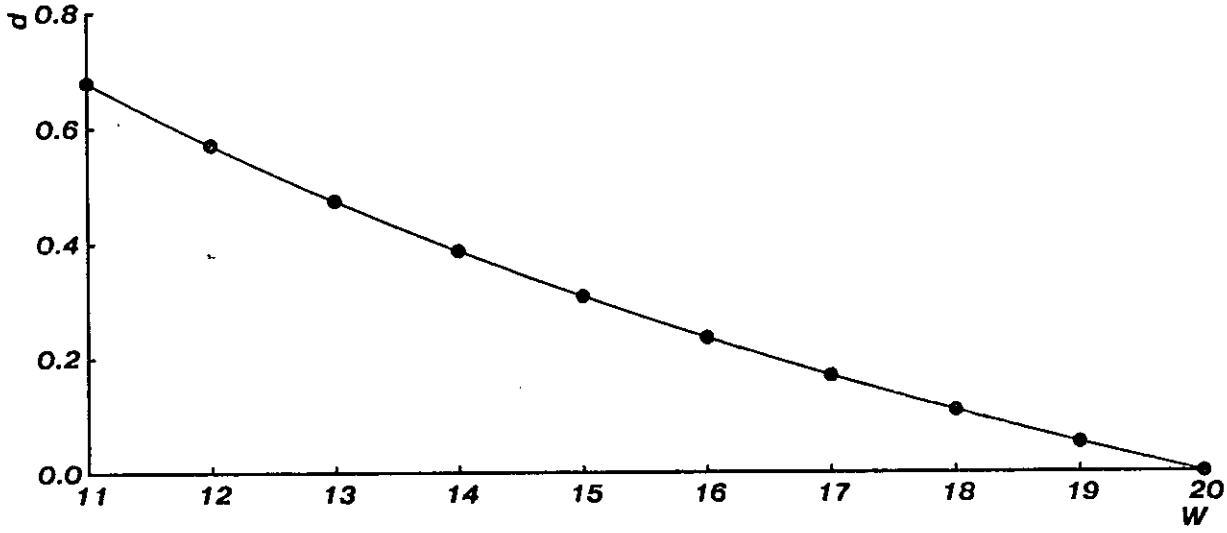

Figure 23: Expected Delete List Lengths in a 20 - (21 – W) - W Suite

$$d \leq \frac{4(N-\frac{N}{2})}{N+3\frac{N}{2}}$$

$$= \frac{4}{5}.$$

In other words, the average size of a delete list will not grow beyond .8, no matter what values we pick for the parameters.

A similar result holds for the size ratio (E[S/36]). The expression for this quantity is:

$$\frac{2(N+W)}{N+3W}.$$

Standard methods show that this expression, subject to the same constraints as before, also achieves its maximum when $W=\frac{N}{2}$, independent of N. Thus its value is bounded by:

$$\frac{2(N+\frac{N}{2})}{N+3\frac{N}{2}}$$

$$= \frac{6}{5}$$

These two performance measures completely specify the significant time and space requirements of the system. Therefore, average performance cannot degrade without bound, regardless of what values we choose for the parameters.

In the simulations and analysis. we assumed that the directory modification operations (Insert, Update and Delete) occur with equal likelihood. In practice, the operation mix will vary from application to application.

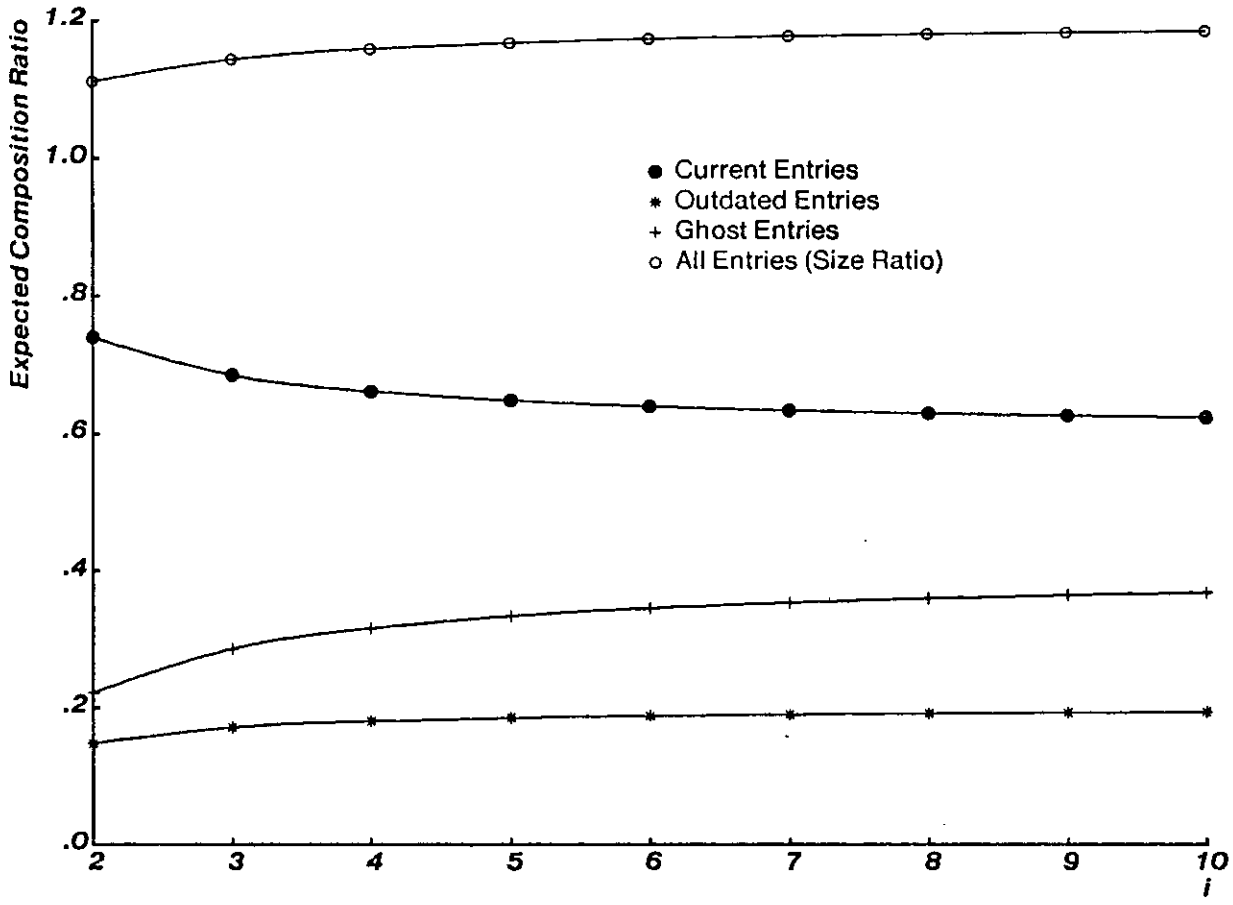

Figure 24: Expected Composition Ratios in a $(2i-1)$ - i - i Suite

It is straightforward to extend the analysis to cover other operation mixes. This is accomplished by substituting the frequency of each operation for the appropriate terms in the balance equations, instead of assuming that all such terms are 1/3 (Appendix I). We extended the analysis along these lines. For brevity's sake, we will not present the details of the analysis, but briefly summarize the results.

We allow the probability that the operation is an **Update**, which we call P_u, to vary from zero to one. If the **Insert** probability is unequal to the **Delete** probability, the number of keys in the directory will dwindle to zero or increase without bound; thus we assume they are equal. Under this assumption, P_u completely specifies all the operation frequencies. The extended analysis consisted of recasting the balance equations in terms of P_u, solving them and studying the solutions.

For a 3-2-2 suite, the average delete list length does not vary significantly over the entire spectrum of P_u values, achieving a minimum of .43 at $P_U=0$ and a maximum of .5 at $P_U=1$. Similarly, the size ratio achieves a minimum of 1.07 at $P_U=0$ and a maximum of 1.25 at $P_U=1$. In fact, the favorable limiting behavior results presented above can be generalized. For all legal values of N, W and P_u, the average delete list size will

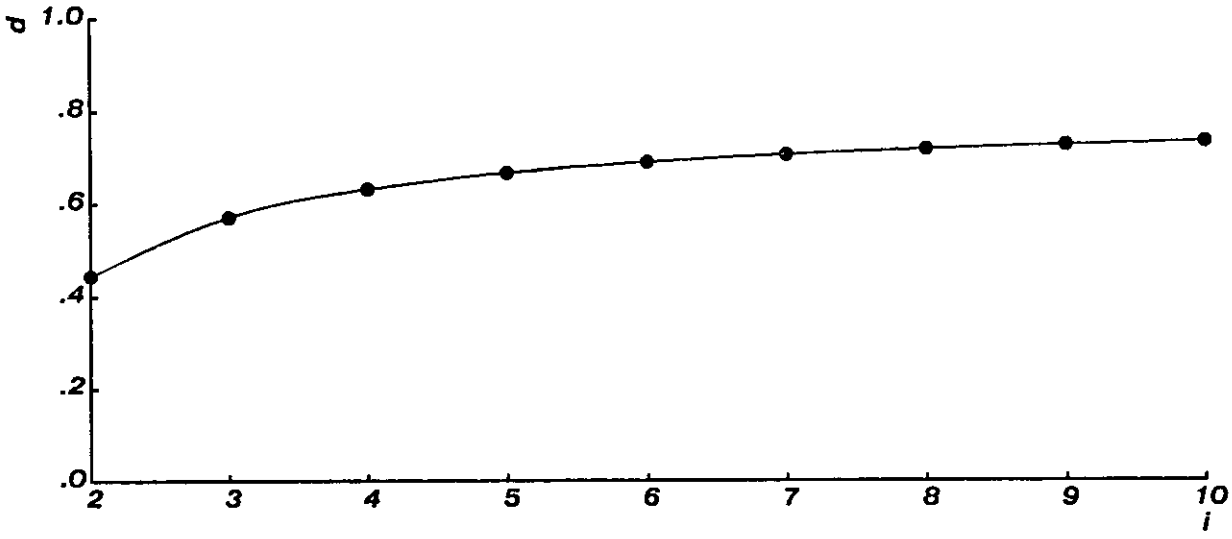

Figure 25: Expected Delete List Lengths in a $(2i-1)$ - i - i Suite

always be <1, and the size ratio will always be <1.5. Thus the performance of the system remains good for any (random) operation mix.

5.2.6 Discussion of the Analysis

The primary purpose of this section is to discuss the validity of the analysis and applicability of the results. Since the model itself is exact, the correctness of the assumptions embodied in the analysis determine its validity. Therefore, we shall enumerate and examine the four assumptions:

1. In each balance equation, we assumed that the three operations (**Insert**, **Update** and **Delete**) occur with equal probability. (p. 51)

2. In the balance equation for current entries, we assumed that the probability that a representative contains an entry for the real predecessor of a randomly chosen key in the directory was equal to the probability that it contained a randomly chosen key in the directory. (p. 52)

3. In the balance equations for current and ghost entries we ignored the possibility of a ghost entry becoming outdated or current in the **Insert** operation. (pp. 53, 55)

4. In the formula for $E[\mathcal{G}/\mathcal{R}]$ we assumed that each Ghost in a representative appeared in exactly two delete lists. (p. 39)

The first assumption holds in all states of the model except those representing directories containing every key in the key space or no keys at all. One cannot insert a key if there are no more keys to insert, and one cannot delete a key if there are no keys in the directory. However, these "boundary states" represent a negligible fraction of all system states and occur with extremely low probability, assuming the key space is reasonably large. If the key space is small, it takes a much shorter run of inserts to fill the directory or deletes to empty it; thus these boundary states occur with much greater likelihood. In fact, the key space used in the simulations was large enough that these states were never encountered.

The second assumption concerns the probability that a representative contains an entry for the real predecessor of a chosen key. In any given system state, the number of keys in the directory that have an entry in a given representative can differ by at most one from the number of keys whose real predecessor has an entry in this representative. This is so because all of the keys in the directory except the last one are the real predecessor of another key in the directory. Thus, the probability that a randomly selected key from the directory has an entry in this representative differs by at most $1/\mathfrak{K}$ from the probability that the real predecessor of a randomly selected key has an entry in the representative. But if the key space is large, $\mathfrak{K}$ will be large in the system states that occur with high probability and this assumption will be almost correct.

The third assumption is that ghost entries cannot enter the outdated or current class in an **Insert** operation. This actually occurs when a key that has been deleted from the directory is reinserted while a ghost for the original incarnation of the key still exists in some representative. This event is extremely unlikely when the key space is large compared to the number of entries in a representative. The simulations were not run long enough for the directory to contain a sizable fraction of the key space, thus they erred in the same direction as this assumption. This assumption would seem to break down in ghost prone configurations where N is much greater than W. However, as long as the representatives contain ghosts for a negligible fraction of the key space, the assumption remains valid.

The fourth assumption is very similar to the second. All ghosts in a representative except those before the first key in the directory and after the last key in the directory do occur in two delete lists. The other ghosts occur in only one delete list. However, in all reasonably likely states, the ghosts are fairly well distributed among the keys in the directory, thus on average, only a small constant number of ghosts will be on only one delete list. For representatives containing reasonably many entries, these few ghosts will be "swamped" by the ghosts that appear on two delete lists, and $\mathfrak{D}/2$ will be almost identical to $\mathfrak{G}/\mathfrak{K}$. If the key space is reasonably large, the approximation will be good in all reasonably likely states and the assumption will be valid.

In summary, all of the assumptions quickly become reasonable as the key space gets large. This is the only point where the cardinality of the key space enters into our analysis. It was not used explicitly in any of the equations. None of the assumptions break down when N or W get large (assuming the key space is large); thus, the results concerning limiting behavior are valid. This also implies that the formulae can be used with confidence for any parameter values.

A note should be added concerning the equilibria observed in the simulations. These equilibria definitely did not represent true equilibrium state distributions over our entire model. This is clearly demonstrated by the fact that the simulations did not generate identical average values for the number of keys in the directory ($\mathfrak{K}$) from run to run. The observed average values for $\mathfrak{K}$ were clearly related to the initial number of keys in the directory in each run. This is not at all surprising, when one considers that the number of states in the model is exponential in the cardinality of the key space, and the simulations were run for far fewer steps than

the key space cardinality itself. We proved that a simulation of sufficient length would display equilibrium behavior over the entire model, but our runs were not of sufficient length. This leaves unexplained the fact that the runs exhibited predictable equilibrium behavior for all of the performance measures of concern to us.

The explanation for this phenomenon lies in the fact that our simplified model is still highly lumpable. Moderately sized "clumps" of contiguous states with reasonably high probabilities of occurrence, such as those traversed in each run of the simulation, have the same average values for the performance variables as those predicted for the entire model. In fact, our analysis captures these clumps better than it captures the entire state space, as the clumps tend not to contain the "boundary states" where the assumptions break down.

5.3 Discussion of Performance Characterization

The system simulated and analyzed was not entirely realistic. Read and write quorums would not be chosen randomly in practice. A node would more naturally communicate with easily accessible nodes. Also, because of the cost of establishing a communication session, the node would probably continue to communicate with the same nodes until it had no need for further communication or a failure occurred. Thus, in practice, the read and write quorums used by any given node would probably change infrequently. The random distribution of operations and keys was also unrealistic. However, we conjecture that the performance observed under real conditions will be as good as or better than that of the system studied.

One possible usage pattern for the system is the following: a single read/write quorum that changes infrequently is used for all operations. This is a special case of the scenario described in the previous paragraph. We performed additional simulations to investigate the behavior of the system under this usage pattern. These simulations were identical to the ones previously described except that before each operation, a decision to change the quorum was made with probability p. Whenever it was determined that the quorum was to change, a single, randomly chosen member of the quorum was replaced with a representative chosen at random from those not already in the quorum. Thus, on any given iteration at most one member of the write quorum changed. This usage pattern could occur if a directory suite were being used by a single requester.

Simulations were performed on 3-2-2 directories initially containing 100 keys, with p values of 0.1, 0.01, 0.001, and 0.0001. Two hundred thousand operations were performed in each simulation and data was collected during the final one hundred thousand operations. The results showed that as the value of p decreases, the average delete list size decreases significantly from the value observed under random usage. The size ratios did not change significantly from the size ratios observed under random usage. These results indicate that the total number of outdated and ghost entries remains close to the total under random usage, but they are now concentrated outside of the write quorum. Thus, the delete lists actually encountered tended to be shorter than those observed under random usage.

The results of this simulation are consistent with our conjecture that the performance of the system will be at least as good under realistic usage patterns as it was under the random usage studied in the simulations and analysis.

As previously noted, the algorithm can be used with infinite key spaces. In fact, a natural choice for the key domain is the set of all alphanumeric strings, which is in principle infinite. The system studied in the simulations and analysis was not well defined for infinite key spaces, so it is natural to ask how well the results of the analysis apply to key spaces which are in principal infinite. In practice, the effect of using theoretically infinite key spaces is identical to that of using large but finite key spaces. Namely, it keeps the system away from boundary states where the assumptions made in the analysis break down. Thus, the analysis captures actual usage patterns over infinite key spaces as well as it captures any other actual usage patterns.

One disadvantage of our analysis technique is that it can only be used to determine expected values for the performance measures. Thus we can only characterize the average case performance of our algorithm. It would be nice to have additional information on the probability distributions of the performance measures. The simulations give us some information along these lines, and we can gain some insight by reasoning directly about the worst case performance of our algorithm.

The simulations and our intuition indicate that under realistic access patterns the size ratio will not vary much from its average value. But it is worth noting that one could construct a pathological sequence of operations wherein ghosts were allowed to accumulate in one representative while the directory remained almost empty, causing the size ratio to grow without bound. This could be accomplished by selecting one write quorum for all **Insert** operations and a second write quorum for all **Delete** operations that intersected the first in only one representative. However there is no reason this should occur in practice.

Similar pathological sequences of operations to those described in the previous paragraph can cause delete lists to grow without bound in the representatives outside of the write quorum for **Deletes**. As long as the pattern continues, the long delete lists will not actually be encountered. If these representatives are eventually used in the write quorums for **Deletes** again, the first few **Deletes** at these representatives will encounter long delete lists. Thus these first few **Deletes** will run slowly at these representatives, but in the process, they will purge the representatives of excess ghosts, so future **Delete** operations will run quickly. Furthermore it should be noted that even in such a pathological case, a maximum of three rounds of messages are sufficient to perform the **Delete** operation; the extra work is all local to the representatives. This sort of situation is very unlikely to occur in practice, and even if it does occur, it should not cause problems, as even a very long delete list (say 100 entries) can be scanned and purged quickly if an efficient data structure is used to store the representative. If it is particularly important for some application that all **Delete** operations run fast, care can be taken to ensure that all representatives are frequently used in the write quorums for **Delete** operations, and so kept clean of excess ghosts.

6 Discussion

The comparison of weighted voting with non-distributed techniques such as mirroring is a complex topic that this paper will not attempt to cover. However, it appears that there is a clear tradeoff between function and performance. Weighted voting provides higher survivability, reliability, availability, and easier maintenance than mirroring, but requires more inter-node communication and incurs the inefficiency and complexity of an underlying transaction mechanism. The advantages of weighted voting primarily result from the storage of data at autonomous nodes that can be physically separated. Though the overhead of transaction and communication mechanisms may be reduced (or accepted because of their utility in constructing complex systems), directory suite operations will always require at least one non-local operation to preserve availability.

Our algorithm may be used in various ways to implement replicated directories that support a high volume of operations. If **Lookup** operations predominate, suite configurations with a large number of representatives and a write quorum much larger than the read quorum permit a high degree of parallelism; readers may simultaneously execute on the nodes that have copies of the data. For supporting a large volume of **Insert**, **Delete** or **Update** operations, it may be best to represent the directory as a collection of subdirectories, each using only a moderate number of directory representative servers. Then, multiple updates on the various subdirectories can occur in parallel. However the availability of the subdirectories may be lower due to the smaller suite sizes.

Directory suites can be configured to take advantage of locality of reference with respect to keys. In particular, quorums can be chosen that permit reads to be done locally and non-local writes to be distributed among all the non-local representatives. For example, consider a 4-2-3 directory suite with key values in the range of 1 to 100, and locality such that transactions of Type A operate on entries having keys 1 to 50, and transactions of Type B operate on entries having keys 51 to 100. We assume that representatives A1 and A2 are local to transactions of Type A and representatives B1 and B2 are local to transactions of Type B. As shown in Figure 26, Type A transactions read from representatives A1 and A2 and direct their updates to A1, A2, and either B1 or B2. Transactions of Type B behave analogously. In this example, all inquiries can be done locally and the non-local write that is required for modification operations is evenly distributed among the remote representatives.

Throughout the paper, we have assumed that the four directory operations use the same read and write quorum sizes in a given suite. Herlihy points out that additional quorum choices are opened up if this restriction is dropped [Herlihy 86]. In his work, each operation has its own read and write quorum size, referred to as the operation's *initial* and *final* quorum size, respectively. Our algorithm can be easily extended to handle such quorum choices. The directory representative operations which are now performed at R or W nodes in the course of an operation are instead performed at an initial or final quorum for the operation, respectively. All of our correctness arguments remain valid. However, if such a quorum choice is employed,

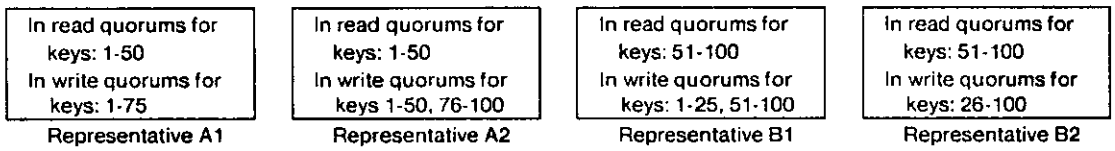

Figure 26: A 4-2-3 Directory Suite Partitioned for Locality

it is essential that timestamps be used rather than version numbers, as initial quorums will not generally contain the most recent entry associated with a key.

The new quorum choices provided by Herlihy's method increase the availability of the Update operation at the expense of the **Lookup** operation. Thus, such quorum choices could result in increased performance for Update intensive directories. It would be straightforward to extend our analysis to cover such quorum choices. In constructing the balance equations, the terms representing the probabilities of being in the read or write quorum for an operation would be replaced by the probability of being in the appropriate initial or final quorum. While we have not performed this analysis, we strongly conjecture that the performance of our algorithm would remain good for the new quorum choices. Specifically, we conjecture that the average delete list length and size ratio would still be bounded by small constants.

As mentioned in the introduction, we have implemented a version of this algorithm on the TABS prototype distributed transaction facility. In doing so, we have resolved some details not addressed in this paper. For example, our implementation stores data for directory representatives as B-trees [Comer 79], and version numbers for gaps are stored in fields of their bounding entries. Physical shared/exclusive mode locking, rather than range locking, was used since the implementation was unlikely to have concurrency control bottlenecks. As might be expected, the major complexity lay in the implementation of the directory representatives, primarily because they were stored as B-trees.

In summary, we have presented a replication algorithm for directories that exhibits favorable performance and availability properties. As is the case with Gifford's weighted voting algorithm, the exact configuration of suites can be tailored to control availability and performance tradeoffs. This algorithm achieves high concurrency while maintaining consistency by dynamically partitioning the key space into ranges at each representative and associating a version number with each range. We proved a property of directory suites that permits deletions to be done in only one or two exchanges of small, fixed length messages with a read quorum and one exchange with a write quorum. Thus all operations can be performed in a small constant number of rounds of messages. We presented a novel analysis of our algorithm which agreed remarkably well with simulation results. The analysis indicates that the space and time costs associated with our algorithm are low for any permissible suite configuration.

Acknowledgments

James Driscoll suggested improvements to our initial dynamic partitioning algorithm that resulted in the data structure described in this paper. John Lehoczky provided invaluable assistance in the definition and analysis of our analytic model. Discussions with Maurice Herlihy provided useful insights into quorum intersection issues. David Gifford, Solom Heddaya, Cynthia Hibbard, and Robert Sansom read and commented on drafts of this paper.

I. Detailed Formulation of Balance Equations

Let us first construct the balance equation for current entries. A formal statement of the rate balance assertion is:

E[The number of entries entering the current class in a chosen representative in one operation]
= E[The number of entries leaving the current class in a chosen representative in one operation].

These expected values are computed over a space consisting of all of the possible state transitions in our model. We expand the expectation values on both sides of the equation by breaking the space up into three subspaces: the transitions that result from **Insert** operations, **Update** operations and **Delete** operations:

P[Opr is **Insert**] × E[The number of entries entering the current class in one **Insert** opr]
+ P[Opr is **Update**] × E[The number of entries entering the current class in one **Update** opr]
+ P[Opr is **Delete**] × E[The number of entries entering the current class in one **Delete** opr]
= P[Opr is **Insert**] × E[The number of entries leaving the current class in one **Insert** opr]
+ P[Opr is **Update**] × E[The number of entries leaving the current class in one **Update** opr]
+ P[Opr is **Delete**] × E[The number of entries leaving the current class in one **Delete** opr].

We will assume that all of the probabilities in this equation are $\frac{1}{3}$, as **Inserts**, **Deletes** and **Updates** occur with almost equal likelihood. The reason that they do not occur with exactly equal likelihood is that **Deletes** and **Updates** cannot occur in states where the directory contains no keys, and **Inserts** cannot occur in states where the suite already contains every key in the key space. However, these states represent a negligible fraction of the state space and they all occur with extremely low probability. Each term has one of these factors, so under the assumption, they all cancel out.

To derive the first balance equation in terms of the unknowns, we expand the expected values in the order they appear in the equation. The first term is:

E[The number of entries entering the current class in one **Insert** operation].

A single entry will enter the current class if and only if the representative under observation is chosen for the write quorum of the **Insert** operation. Thus the expected value is merely the probability that the representative is chosen. Since there are N representatives in the suite, and W are chosen at random for the write quorum, this is $\frac{W}{N}$.

The second term is:

E[The number of entries entering the current class in one **Update** operation].

Again, an entry can enter the current class only if the representative is chosen for the write quorum. This time, however, the entry for the key being updated will not necessarily enter the current class, as the representative could already have contained a current entry for this key. In that case, no entry that was not already current would become current. Thus, the value of the term is:

P[The representative is chosen for the write quorum]
× (1 − P[The representative already contains a current entry for the key being updated]).

The probability that the representative is chosen for the write quorum is $\frac{W}{N}$. The key to be updated is chosen at random from those in the directory so:

P[The representative already contains a current entry for the key being updated]
= P[The representative contains a current entry for a randomly chosen key in the directory]
= c'

Thus, the value of the second term is:

$$\frac{W}{N}(1-c').$$

The third term is:

E[The number of entries entering the current class in one **Delete** operation] .

When a **Delete** operation occurs, entries for the real predecessor and real successor of the key being deleted are inserted into each member of the write quorum where they do not already appear. They are inserted with their latest version number so they become additional current entries in those representatives. This is the only way entries can enter the current class in a **Delete** operation. Thus the number of entries entering the current class in the observed representative in one **Delete** operation is zero if the representative is not chosen for the write quorum. If it is chosen for the write quorum, then one entry will become current if the representative does not contain an entry for the real predecessor of the key being deleted, and another entry will become current if the representative does not contain an entry for the real successor.

We introduce some notation for events to simplify the discussion that follows:

P = {The representative contains an entry for the real predecessor of the key being deleted}
S = {The representative contains an entry for the real successor of the key being deleted} .

On the basis of the previous observations, the value of the term being expanded is:

P[The representative is chosen for the write quorum] $\times$(P[P^c] + P[S^c])
$= \frac{W}{N}((1-\mathrm{P}[P])+(1-\mathrm{P}[S]))$.

While P[P] and P[S] cannot be exactly expressed in terms of our unknowns, they can be very closely approximated. The key to be deleted is chosen at random from those in the directory, and its real predecessor is merely the key immediately preceding it in the directory. If the key being deleted is the first key in the directory, its real predecessor is the dummy key **LOW**, which is always present in every representative. Thus the probability that the real predecessor is present in the representative (P[P]) is just slightly higher than the probability that a randomly chosen key in the directory is present in the representative. For a large key space like the one used in the simulations they will be practically identical. By symmetry, the same argument holds for the real successor. In fact, it shows that P[P] = P[S]. Therefore, we make the assumption that:

P[P] = P[The representative contains an entry for a randomly chosen key in the directory]
= P[The representative contains a current entry for a randomly chosen key in the directory]
+ P[The representative contains an outdated entry for a randomly chosen key in the dir.]
= $c' + o'$,

The third term becomes:

$$2\frac{W}{N}(1-(c'+o')).$$

Now we come to the terms on the right hand side of the balance equation. The first term on the right hand side is:

E[The number of entries leaving the current class in one **Insert** operation] .

This term vanishes, as no entries leave the current class in **Insert** operations.

The second term on the right hand side is:

E[The number of entries leaving the current class in one **Update** operation] .

If the representative under observation contains a current entry for the key being updated, and the representative is *not* chosen for the write quorum, then the current entry becomes outdated. Thus the value of this term is:

(1 − P[The representative is chosen for the write quorum])
× P[The representative contains a current entry for a randomly chosen key in the directory]
$= (1-\frac{W}{N})c'$.

The third term on the right hand side is:

E[The number of entries leaving the current class in one **Delete** operation] .

If the representative under observation contains a current entry for the key being deleted, the entry will leave the current class regardless of whether or not the representative is chosen for the write quorum. If it is chosen, the entry will be deleted outright; otherwise, the entry will become a ghost. Thus the value of this term is:

P[The representative contains a current entry for the key being updated]
$= c'$.

Combining all these terms, the balance equation for current entries is:

$$\frac{W}{N}+\frac{W}{N}(1-c')+2\frac{W}{N}(1-(c'+o'))=(1-\frac{W}{N})c'+c'.$$

Simplifying, we get:

$$(N+W)c'+Wo'=2W.$$

We now construct the balance equation for outdated entries. By an argument identical to the one used in the construction of the first balance equation, a formal statement of the rate balance assertion becomes:

E[The number of entries entering the outdated class in one **Insert** operation]
+ E[The number of entries entering the outdated class in one **Update** operation]
+ E[The number of entries entering the outdated class in one **Delete** operation]
= E[The number of entries leaving the outdated class in one **Insert** operation]
+ E[The number of entries leaving the outdated class in one **Update** operation]
+ E[The number of entries leaving the outdated class in one **Delete** operation] .

We shall assume that entries cannot enter the outdated class in **Insert** operations, so the first term of the left hand side of the equation vanishes. In fact, if a key is inserted when ghosts for a previous incarnation of that key still remain in representatives outside of the write quorum for the **Insert** operation, those ghosts will become outdated. However, this is an extremely unlikely event, hence this term of the equation is negligible compared to the others. Furthermore, it is not expressible in terms of the unknowns.

Entries cannot enter the outdated class in the **Delete** operation, so the third term of the equation also vanishes. In the **Update** operation an entry can become outdated as follows. If the representative is not chosen for the write quorum and it contains a current entry for the key being updated, then the entry becomes outdated. Thus the value of the second terms is:

(1 − P[The representative is chosen for the write quorum])
× P[The representative contains a current entry for a randomly chosen key in the directory]
$= (1-\frac{W}{N})c'$.

Entries cannot leave the outdated class in **Insert** operations, so the first term of the right hand side of the equation vanishes. In an **Update** operation, an entry can leave the outdated class as follows. If the representative is chosen for the write quorum and it contains an outdated entry for the key being updated, then this entry is replaced by a current one. Thus, the second term on the right hand side is:

P[The representative is chosen for the write quorum]
× P[The representative contains an outdated entry for the key being updated]
= P[The representative is chosen for the write quorum]
× P[The representative contains an outdated entry for a randomly chosen key in the directory]
$= \frac{W}{N}o'$.

In a **Delete** operation, an entry can leave the outdated class as follows: If the representative contains an outdated entry for the key being deleted, then the entry disappears if the representative is chosen for the write quorum, and it becomes a ghost if the representative is not chosen for the write quorum. Thus the third term on the right hand side is:

P[The representative contains an outdated entry for the key being deleted]
= P[The representative contains an outdated entry for a randomly chosen key in the directory]
$= o'$.

Putting it all together, the balance equation for outdated entries is:

$$(1-\frac{W}{N})c' = \frac{W}{N}o' + o'.$$

Simplifying, this becomes:

$$o' = \frac{N-W}{N+W}c'$$

Finally, we construct the balance equation for ghost entries. A formal statement of the balance assertion becomes:

E[The number of entries entering the ghost class in one **Insert** operation]
+ E[The number of entries entering the ghost class in one **Update** operation]
+ E[The number of entries entering the ghost class in one **Delete** operation]
= E[The number of entries leaving the ghost class in one **Insert** operation]
+ E[The number of entries leaving the ghost class in one **Update** operation]
+ E[The number of entries leaving the ghost class in one **Delete** operation].

Entries can only enter the ghost class in **Delete** operations; thus, the first and second terms of the equation

vanish. An entry becomes a ghost in a representative if its key is being deleted and that representative is not chosen for the write quorum of the delete operation. Thus the second term is:

$$
\begin{aligned}
&(1 - \text{P[The representative is chosen for the write quorum]}) \\
&\quad \times \text{P[The representative contains an entry for a randomly chosen key in the directory]} \\
&= (1-\frac{W}{N})(c'+o').
\end{aligned}
$$

Entries rarely leave the ghost class in **Insert** operations, thus we shall assume the first term on the right hand side vanishes. (This is essentially the same assumption we made on page 53 when constructing the balance equation for outdated entries.) Entries cannot leave the ghost class in **Update** operations, thus the second term on the right hand side actually does vanish. If the representative is chosen for the write quorum of the **Delete** operation then all of the ghosts constituting the delete list of the key being deleted will be removed from the representative. Thus the third term of the right hand side is:

$$
\begin{aligned}
&\text{P[The representative is chosen for the write quorum]} \\
&\quad \times \text{E[The size of the delete list of the the key being deleted]} \\
&= \text{P[The representative is chosen for the write quorum]} \\
&\quad \times \text{E[The size of the delete list of the a randomly chosen key in the directory]} \\
&= \frac{W}{N}d.
\end{aligned}
$$

Putting the terms together, the balance equation for ghosts is:

$$(1-\frac{W}{N})(c'+o')=\frac{W}{N}d.$$

Simplifying:

$$d=\frac{N-W}{W}(c'+o').$$

References

[Abbadi and Toueg 86]
Amr El Abbadi, Sam Toueg.
Availability in Partitioned Replicated Databases.
In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems.* 1986.

[Abbadi et al. 85] Amr El Abbadi, Dale Skeen, Flaviu Cristian.
An Efficient, Fault-Tolerant Protocol for Replicated Data Management.
In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems.* March, 1985.

[Allchin 83] James E. Allchin.
An Architecture for Reliable Distributed Systems.
PhD thesis, Georgia Institute of Technology, September, 1983.

[Allchin and McKendry 83]
J. E. Allchin, M.S. McKendry.
Synchronization and Recovery of Actions.
In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing,* pages 31-44. ACM, August, 1983.

[Alsberg and Day 76]
P. A. Alsberg, J. D. Day.
A Principle for Resilient Sharing of Distributed Resources.
In *Proceedings of the Second International Conference on Software Engineering,* pages 562-570. October, 1976.

[Bartlett 81] Joel Bartlett.
A NonStop™ Kernel.
In *Proceedings of the Eighth Symposium on Operating System Principles.* ACM, 1981.

[Bernstein and Goodman 84]
P. Bernstein and N. Goodman.
An algorithm for concurrency control and recovery in replicated distributed databases.
ACM Transactions on Database Systems 9(4):596-615, December, 1984.

[Birman et al. 83] K. P. Birman, D. Skeen, A. El Abbadi, W.C. Dietrich, T. Raeuchle.
Isis: An Environment for Constructing Fault-Tolerant Distributed Systems.
Technical Report 83-552, Cornell University, 1983.

[Birrell and Nelson 84]
Andrew D. Birrell, Bruce J. Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[Comer 79] Douglas Comer.
The Ubiquitous B-Tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[Daniels and Spector 83]
Dean S. Daniels, Alfred Z. Spector.
An Algorithm for Replicated Directories.
In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing,* pages 104-113. ACM, August, 1983.
Also available in Operating Systems Review 20(1), January 1986, pp. 24-43.

[Gifford 79] David K. Gifford .
Weighted Voting for Replicated Data.
In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 150-162. ACM, December, 1979.

[Gifford 81] David K. Gifford.
Information Storage in a Decentralized Computer System.
PhD thesis, Stanford University, 1981.
Available as Xerox Palo Alto Research Center Report CSL-81-8, March 1982.

[Gray 80] James N. Gray.
A Transaction Model.
Technical Report RJ2895, IBM Research Laboratory, San Jose, California, August, 1980.

[Gray et al. 81] James N. Gray, et al.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys 13(2):223-242, June, 1981.

[Herlihy 85] Maurice P. Herlihy.
Availability vs. atomicity: concurrency control for replicated data.
Technical Report CMU-CS-85-108, Carnegie-Mellon University, February, 1985.

[Herlihy 86] Maurice P. Herlihy.
A Quorum-Consensus Replication Method for Abstract Data Types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[IBM Corporation 75]
ACP System: Concept and Facilities
GH20-1473-1 edition, IBM Corporation, White Plains, New York, 1975.

[Kemeny and Snell 60]
John G. Kemeny, J. Laurie Snell.
Finite Markov Chains.
D. Van Nostrand & Co., New York, 1960.

[Korth 83] Henry F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1):55-79, January, 1983.

[Lindsay et al. 79]
Bruce G. Lindsay, et al.
Notes on Distributed Databases.
Technical Report RJ2571, IBM Research Laboratory, San Jose, California, July, 1979.
Also appears in Droffen and Poole (editors), *Distributed Databases*, Cambridge University Press, 1980.

[Liskov and Scheifler 83]
Barbara H. Liskov, Robert W. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[Popek et al. 81] G. Popek, B. Walker, J.Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel.
LOCUS: A Network Transparent, High Reliability Distributed System.
In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 169-177. ACM, 1981.

[Rothnie et al. 77]
J. B. Rothnie, N. Goodman, P.A. Bernstein.
The Redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully Redundant Case).
Technical Report CCA-77-02, Computer Corporation of America, 1977.

[Schwarz 84] Peter M. Schwarz.
Transactions on Typed Objects.
PhD thesis, Carnegie-Mellon University, December, 1984.
Available as Technical Report CMU-CS-84-166, Carnegie-Mellon University.

[Schwarz and Spector 84]
Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
ACM Transactions on Computer Systems 2(3):223-250, August, 1984.
Also available as Technical Report CMU-CS-83-163, Carnegie-Mellon University, November 1983.

[Spector and Schwarz 83]
Alfred Z. Spector, Peter M. Schwarz.
Transactions: A Construct for Reliable Distributed Computing.
Operating Systems Review 17(2):18-35, April, 1983.
Also available as Technical Report CMU-CS-82-143, Carnegie-Mellon University, January 1983.

[Spector et al. 85a]
Alfred Z. Spector, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Randy Pausch.
Distributed Transactions for Reliable Systems.
In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127-146. ACM, December, 1985.
Also available in *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company, New York, and as Technical Report CMU-CS-85-117, Carnegie-Mellon University, September 1985.

[Spector et al. 85b]
Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, Peter M. Schwarz.
Support for Distributed Transactions in the TABS Prototype.
IEEE Transactions on Software Engineering SE-11(6):520-530, June, 1985.
Also available in Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Springs, Maryland, IEEE, October, 1984 and as Technical Report CMU-CS-84-132, Carnegie-Mellon University, July, 1984.

[Traiger et al. 82] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, Bruce G. Lindsay.
Transactions and Consistency in Distributed Database Systems.
ACM Transactions on Database Systems 7(3):323-342, September, 1982.

[Weihl 83] William E. Weihl.
Data Dependent Concurrency Control and Recovery.
In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 63-75. ACM, August, 1983.

[Weihl and Liskov 83]
W. Weihl, B. Liskov.
Specification and Implementation of Resilient, Atomic Data Types.
In *Symposium on Programming Language Issues in Software Systems.* June, 1983.