Weighted Voting for Directories: A Comprehensive Study

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

April 15, 1984

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 algorithm is modeled and analyzed. The analysis agrees well with the simulation, and the space and time performance are shown to be good for all possible configurations of the system.

Technical Report CMU-CS-84-114

This work was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539 and in part by the NSF under Contract MCS-8308805

Table of Contents

List of Figures

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 *data availability* - continued access to objects despite failures of one or more storage nodes. Unfortunately, it is difficult to achieve high performance and reliability 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. The semantics of the replicated directory are typical of directories stored on a single node: We define a *directory* as an abstract data object that maps *keys* to *values*. Keys are chosen from a large ordered set of constants called the *key space*. Directories are accessed and modified with the following operations:

- **Insert(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(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(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(K:Key) Returns(Boolean, 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 provokes an error response but 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 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 deletion 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 work on replication is part of the TABS (Transaction-based Systems) Project, which is studying distributed systems that use a transaction facility to support operations on shared abstract data types [Schwarz

83a, Spector 83a]. The directory described in this paper is an example of a serializable, distributed abstract data type that is constructed from a collection of more primitive, non-serializable, non-distributed types, each of which use synchronization and recovery primitives supported by such a transaction facility. Additional components of our research address synchronization, recovery, and communication issues. Groups at Cornell, MIT, and Georgia Institute of Technology are also investigating the wider use of transactions [Allchin 83a, Allchin 83b, Birman 83, Liskov 82, Weihl 83a, Weihl 83b].

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 algorithm's concurrency performance can be improved by relaxing the synchronization requirements for the directory replicas. Following this presentation of the algorithm, we present performance data obtained by simulation and develop a mathematical model of the system being 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 possible possible configurations of the system. Finally, we discuss the advantages and uses of the algorithm.

2 Related Work and Motivation

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. 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].

We are more interested in replication techniques that use a distributed collection of cooperating nodes to store replicas of the data. Many of these techniques provide higher data reliability and availability than mirroring, though they generally have higher overhead and complexity. In this section we briefly review the fundamental distributed replication algorithms and develop a distributed replication strategy for directories that is based on weighted voting. (See Lindsay for a more complete survey of some of these approaches [Lindsay 79].)

One fundamental distributed replication strategy is unanimous update: any update operation must be done on all replicas, but reads may be directed to any replica. This replication strategy guarantees data consistency 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 77].

In replication strategies based on keeping primary and secondary copies of data, the primary copy receives all updates and then relays the updates to secondary copies [Alsberg 76]. An inquiry may be sent to a secondary copy, but the result may not reflect the most current 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 81]. Locus maintains availability after synchronization site failure by nominating a new synchronization site.

Gifford designed a strategy for replication of files, called *weighted voting* [Gifford 79, Gifford 81]. In this strategy, a file is stored as 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 modify 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 this data. Read operations read from each representative in a *read quorum* and return the data from the representative with the highest version number. In the version of weighted voting described in this paper, 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. 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.

Weighted voting has several attributes that make it particularly appealing as the basis for the design of a replicated directory. First, the sizes of the read and write quorums may be varied to adjust the relative cost and availability of reads and writes. For example, a unanimous update strategy may be specified if the data is read much more frequently than it is written. Second, a representative with zero votes may be used to store a locally cached copy that is usually current. This situation naturally occurs if a single local site performs most updates and always updates the cached copy. An example use of such a cache is in a distributed file system

where there is strong locality of file access. Last, the algorithm is simplified because consistency and recovery are primarily the responsibility of an underlying transaction facility that is assumed to exist on each representative. The use of a common underlying transaction facility is advantageous because the facility can simplify distributed applications that use other types of objects in addition to files.

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 representative has a single version number, which causes the serialization of operations that modify the directory.

It might seem that these concurrency limitations could be overcome if each *entry* in a directory representative were assigned a separate version number. An *entry* is the physical data associated with a key, and consists of the key and an associated value. However, if such an approach were used, some representatives might not have a version number for a key that was stored on other representatives. Because of this fact, it is not always possible to determine from an arbitrary read quorum whether a particular key is in the directory. This problem is illustrated in the example that follows.

Consider a 3-representative directory suite having a read quorum of 2 and a write quorum of 2; we call this a 3-2-2 directory suite.[1] Initially, each representative in the suite contains 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 B with version number 1 (Figure 2). If a request to look up the key "b" is sent to representatives A and C at this point, representative A will respond "present with version number 1," and representative C will respond "not present." If "b" is then removed from the directory by deleting its entry from representatives B and C (Figure 3), requests to look up "b" on representatives A and C will still elicit the responses "present with version number 1," and "not present." 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

---

[1]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. For simplicity, all examples in this paper assume that each representative is assigned one vote. All results generalize to directory suites with arbitrary distributions of votes.

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

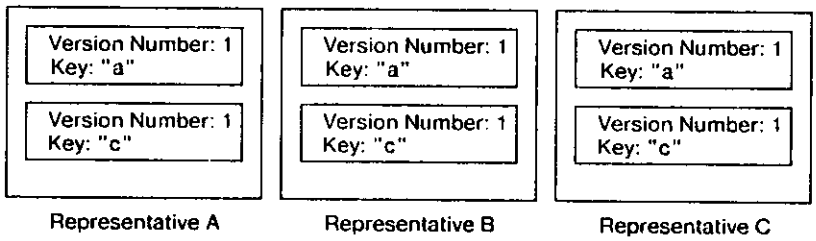

Figure 1: A 3-2-2 Directory Suite - Initial Configuration

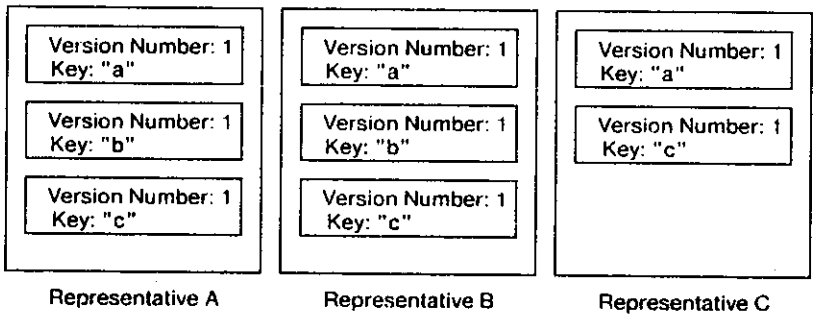

Figure 2: Directory Suite After Inserting "b"

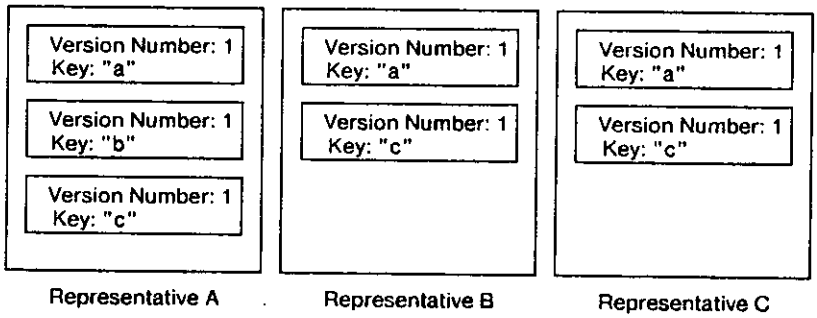

Figure 3: Directory Suite After Deleting "b"

itself be a concurrency bottleneck. A third strategy is to eliminate the ambiguity by consulting additional 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 can drastically reduce 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. Of course, the same partitions need not be used at all representatives.

The key space could be partitioned at each representative by placing each key for which there is an entry in a separate partition, and maintaining a single additional partition for all keys that do not have entries. In other words, each representative keeps a version number for each entry and an additional version number for use with "not present" responses. Under this scheme, deletions increment the "not present" version number. Since the "not present" version number applies to a very large set of keys, this approach suffers from concurrency limitations that are similar to the limitations of the approach of having a single version number per representative.

A more promising approach is to partition the key space into ranges on the basis of the order relation on the keys. The simplest partitioning scheme is to divide 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.

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. For example, using this approach, an entry for "b" would be inserted into representatives A and B of Figure 1 with version number 1, which is one greater than the version number of the gap between "a" and "c" (Figure 4)[3]. If a request to look up "b" were sent to representatives A

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

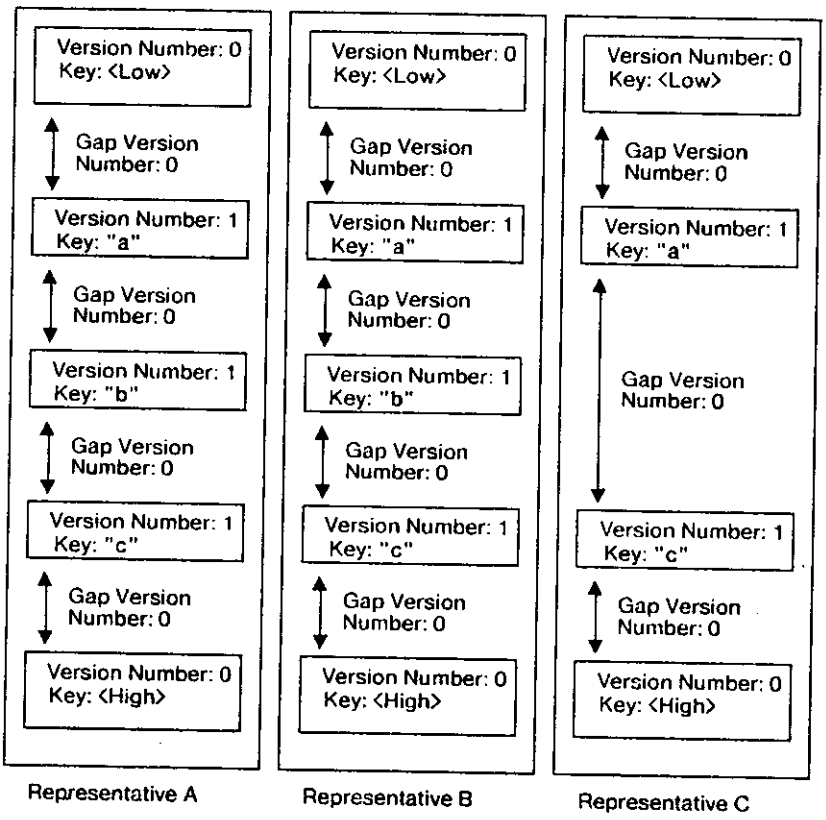

Figure 4: Directory Suite After Inserting "b"

and C at this point, representative A would respond "present with version number 1," and representative B would respond "not present with version number 0." Using these responses, a client could determine that there is an entry for "b" since that response has the larger version number. If "b" were subsequently deleted from representatives B and C, 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 5). Now, if a request to look up "b" is sent to any two representatives, at least one will return "not present with version number 2." This resolves the ambiguity that occurred in the initial example, when version numbers were associated only with entries.

3 Details of the Algorithm

This section presents the details of the approach to directory replication sketched in the previous section. The descriptions are illustrated with program text in a Pascal-like language that allows procedures to return multiple values and includes a remote procedure call primitive. Remote procedure calls are written as "**Send(<procedure invocation>) to(<object instance>)**" and are assumed to return values in the same fashion as a normal procedure invocation. These remote procedure calls have similar semantics to those of ARGUS

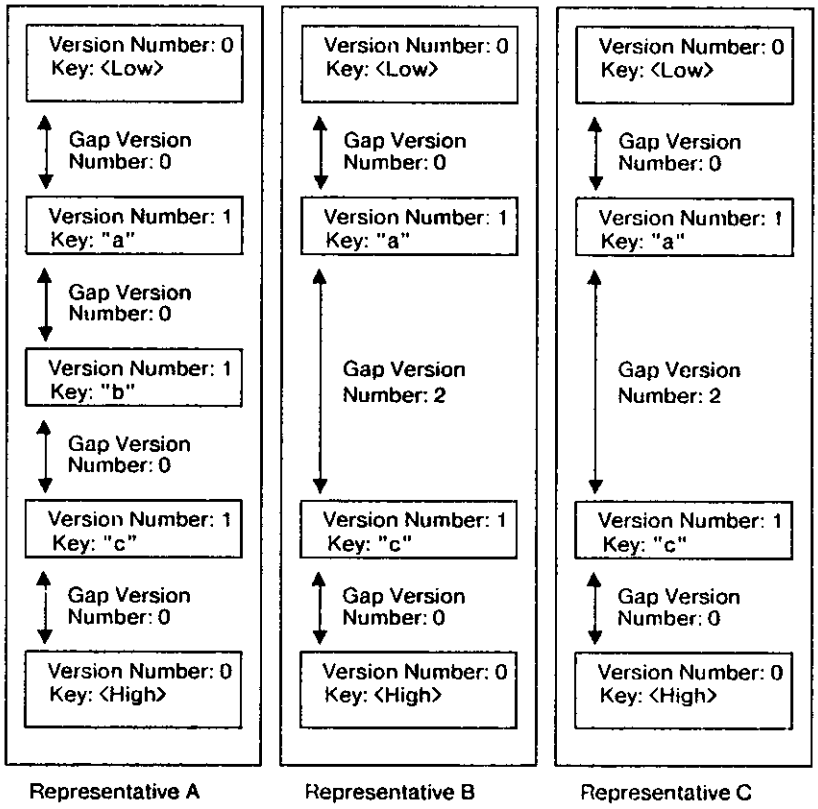

Figure 5: Directory Suite After Deleting "b"

[Liskov 82], except that error responses, such as timeouts, are not considered in these examples. 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 sections. The two following sections develop an essential component of the deletion algorithm. Arguments for the correctness of the replication algorithm are then presented. The final section discusses ways of modifying the algorithm's synchronization policies to provide higher concurrency.

3.1 Directory Representatives

In a replicated directory, each directory representative is an instance of an abstract object that stores one 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.

```
DirRepLookup(x:key) Returns(boolean,version,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). }

DirRepPredecessor(x:key) Returns(key, version, version);
{ Returns the key and version number of the entry with the largest
  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(y,x) where y is the key returned. }

DirRepSuccessor(x:key) Returns(key,version,version);
{ Analogous to above procedure.

  Locks RepLookup(x,y) where y is the key returned.}

DirRepSuperseder(x:key,v:version,y:key)
                Returns(boolean,key,version,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,z) where z is the key returned or y if no key
  is returned.}

DirRepInsert(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).}

DirRepCoalesce(l:key,h:key,v:version);
{ Deletes entries for any keys between (but not including) l and h.
  The resulting gap is assigned version number v.  An error is
  indicated if entries do not exist for keys l and  h.

  Locks RepModify(l,h). }
```

Figure 6: Directory Representative Operations

Every instance of a directory representative contains two distinguished keys, HIGH and LOW. HIGH is greater than any key that can be inserted into the representative, and LOW is less than any key. HIGH and LOW simplify the directory suite 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 3.2.

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 6 gives procedure headings for each of these operations.

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. **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 range of keys 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 any) 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 is used [Korth 83, Schwarz 83a]. 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 the obvious analogues of the lock classes for a single-copy directory, given by Schwarz [Schwarz 83a]. 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 is those keys greater than or equal to σ and less than or equal to τ. A **RepModify**(σ,τ) lock is obtained on the keys of entries modified by the **DirRepInsert** and **DirRepCoalesce** operations.

The lock compatibility relation for operations on directory representatives is illustrated in Figure 7. 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 Requested	None	Lock Held **RepModify**(σ,τ)	**RepLookup**(σ,τ)
RepModify(σ",τ")	OK	No	No
RepModify(σ',τ')	OK	OK	OK
RepLookup(σ",τ")	OK	No	OK
RepLookup(σ',τ')	OK	OK	OK

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

Figure 7: 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 82], provided that two phase locking is used. This form of synchronization simplifies the correctness arguments given in Section 3.5. (Section 3.6 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 either 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 transaction system underlying the representative's implementation. Gray et al., Lindsay et al., and Schwarz and Spector, among others, present more details on general recovery algorithms [Gray 81, Lindsay 79, Schwarz 83b].

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, providing that the recovery system redoes operations in the order in which they were originally performed. An error would occur if a coalesce operation were redone before the insertions of the entries at either end of the range to be coalesced were redone. 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.

3.2 Directory Suites

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

The **Lookup** operation sends **DirRepLookup** requests to a read quorum of representatives and returns the resulting entry[4] with the highest version number. Code for this operation is given in Figure 8.

Operations that modify the directory suite must ensure that the version number of a modified entry is higher than any version number that had been previously associated with the entry's key. In addition, the **Delete** operation must exercise care not to coalesce too large a region and thereby inadvertently assert the nonexistence of keys that are in the directory.

The **Insert** operation is quite simple. **Insert** first looks up the key to be inserted in a read quorum to 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 9 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 gaps that is higher than that of any entry contained in the gaps. To avoid asserting the nonexistence of keys that are actually in the 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 the largest key less than k that is in the directory. The real successor of a key is defined analogously. The entries between a key's real predecessor and its real successor on a representative comprise the key's *delete list* on that representative.

---

[4] Figure 8 shows **Lookup** returning a version number as well as a boolean and the value associated with the key. The version number is used by the procedures **Insert**, and **Delete**. A user would ignore this number.

```
Lookup(k:key) Returns(boolean,version,value)
{ Return True, the version number, and the value of the entry for k
  if it exists; False otherwise. }
var
   { read quorum has R members }
   quorum : array[1..R] of DirRep;
   repsver, bestver : version;
   val, bestval : value;
   isin, bestisin : boolean;
   i : integer;

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

  bestver := LowestVersion - 1; { a constant }
  { send inquiries to each quorum member }
  for i := 1 to R do
    begin
      isin,repsver,val := Send(DirRepLookup(k)) to quorum[i];
      if repsver>bestver then
        begin
          bestver := repsver;
          bestval := val;
          bestisin := isin
        end
    end;
  return(bestisin,bestver,bestval)
end
```

Figure 8: Lookup Operation

Locating the real predecessor and real successor of a key that is 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 write quorum.

These problems are partially illustrated in Figure 10. In this figure, the real successor of the entry "a" is the entry "bb". However "bb" does not appear in representative C, and the ghost of entry "b" appears between "a" and "bb" in representative A. To delete "a" from representative A and C, the real successor, "bb", must first be located and then copied to representative C. The coalescing of the range from LOW to "bb" eliminates the ghost of entry "b" from representative A, as shown in Figure 11.

The Delete operation is illustrated in Figure 12. Finding the real predecessor and successor of a key is the heart of this operation. The straightforward procedure given by Daniels and Spector [Daniels 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,

```
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 }
  isin,ver,val:= Lookup(nkey);
  { 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
    Send(DirRepInsert(nkey,ver,nval)) to(quorum[i])

end
```

Figure 9: Insert Operation

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 including more information in each message, 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.[5] All other directory suite operations, as presented previously [Daniels 83], require only a constant number of fixed length communications; it would be highly desirable to have an algorithm for the real predecessor operation (hence the Delete operation) that has this property as well.

3.3 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 intervening entries 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

[5] In fact, it is bounded by 2R * (the size of the key space), but for all practical purposes, this is unbounded.

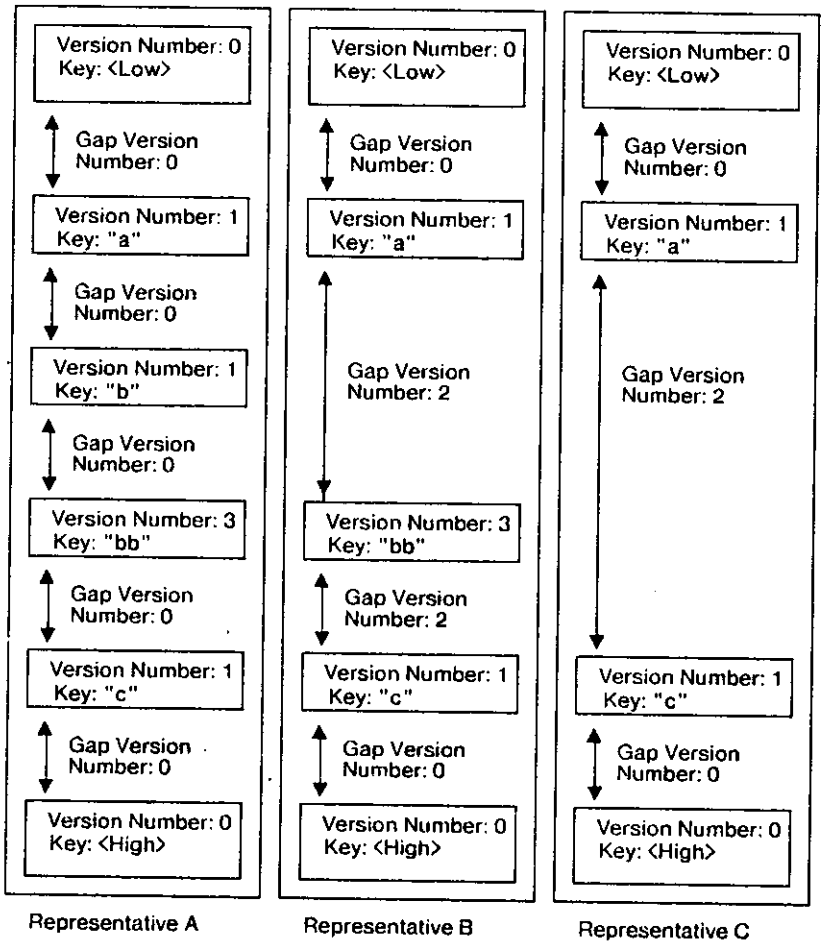

Figure 10: Directory Suite from Figure 5 After Inserting "bb"

the prospects for the existence of an algorithm that requires only a constant number of fixed length messages 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. In keeping with previous usage, we define a *range* as a region containing every key in the key space between some key and another key. 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.

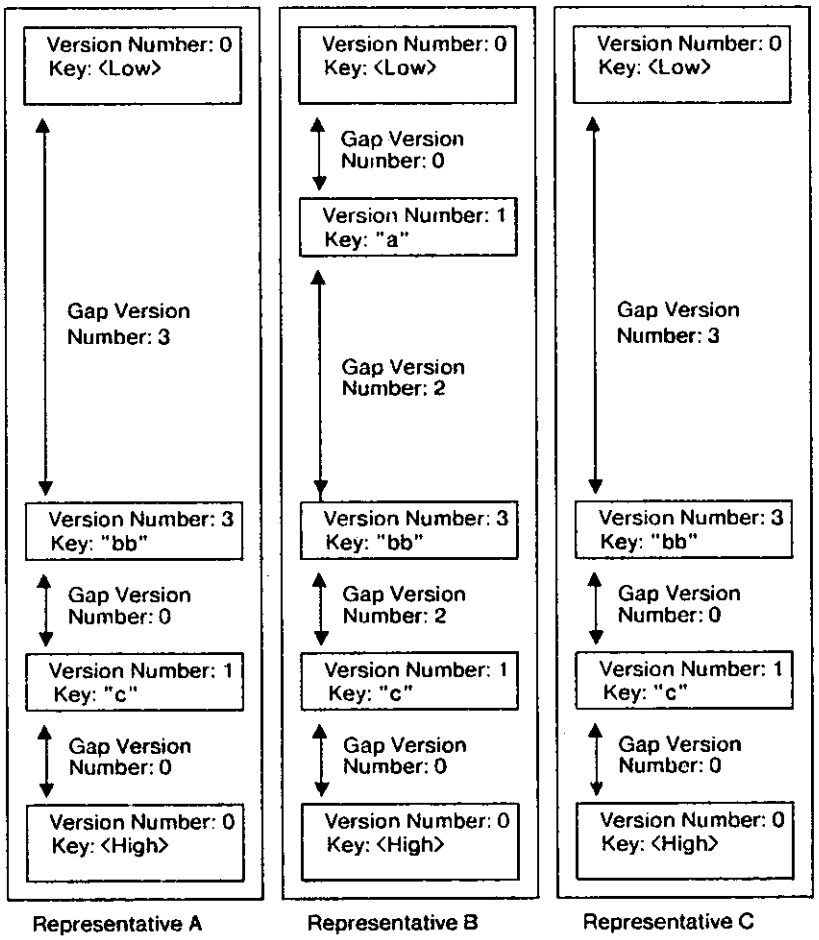

Figure 11: Directory Suite from Figure 10 After Deleting "a"

A gap between entries having keys k_1 and k_2 in a representative is said to *cover* the region (k_1,k_2) and all of its subregions. 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) subset 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.

A gap's *region of currency* is the entire region over which it is current.[6]

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

```
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, sver, repver, ver: version;

begin
  { find a write quorum }
  quorum := CollectWriteQuorum();

  { Find the predecessor and successor of delk }
  succ,sval,sver,ver := RealSuccessor(delk);
  pred,pval,pver,repver := RealPredecessor(delk);

  { The version number of the coalesced gap
    must be higher than the maximum of any
    version numbers in the range coalesced }
  ver := Max(repver, ver);
  isin,repver,val:=Lookup(delk);  { isin, val ignored }
  ver := Max(repver, ver);

  { Ensure the predecessor and successor
    exist in every member of the quorum)
  for i := 1 to W do
    begin
      isin,repver,val:= Send(DirRepLookup(succ)) to(quorum[i]);
      {repver,val ignored}
      if not isin then
        Send(DirRepInsert(succ,sver,sval)) to (quorum[i]);

      isin,repver,val:= Send(DirRepLookup(pred)) to(quorum[i]);
      {repver,val ignored}
      if not isin then
        Send(DirRepInsert(pred,pver,pval)) to (quorum[i])
    end;

  { coalesce the range in each member }
  for i:= 1 to W do
      Send(DirRepCoalesce(pred,succ,ver+1)) to (quorum[i])

end
```

Figure 12: Delete Operation

For example, consider the suite in Figure 13. Gap g covers ("c",HIGH) and all of its subregions, e.g. ("d","f"). Gap g is current over ("g","k"), for example. (Alphabetical ordering on the keys is assumed.) Gap g's region of currency is ("c","d")$\cup$("e",HIGH). We are now ready to state and prove the property.

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

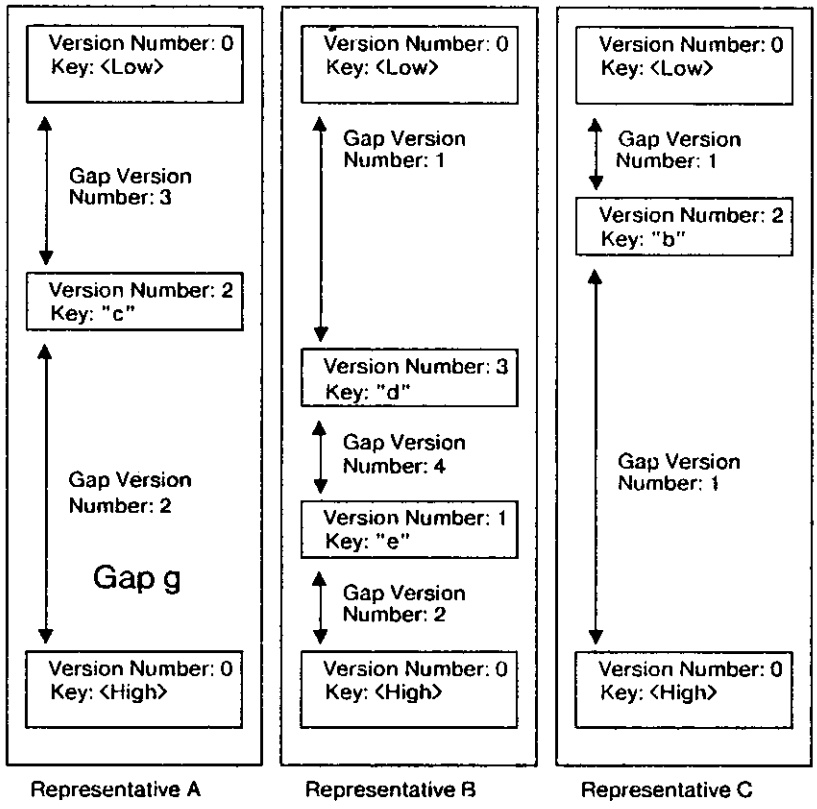

Figure 13: Suite for illustration of *region of currency* and related terminology

The proof 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 region of currency of each of these gaps is expressible as a union of ranges whose endpoints are keys in the directory. One of these ranges must contain k. Let us call this range (k_1,k_2). (Of course the values of k_1 and k_2 may be different for each such gap.) When $\{k\}$ is deleted from such a gap's region of currency, the resulting region will be equivalent to the original region, with (k_1,k_2) replaced by $(k_1,k)\cup(k,k_2)$. But k, k_1 and k_2 are all 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,[7] no gap's region of currency will be affected by the operation. If no entry for k exists, then the gap 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, if k is in the region of currency of the gap being split, it falls in some range that is bounded by keys in the directory and is contained entirely in the region of currency. Let us call this range (k_1,k_2). Then g_1's region of currency will consist of the union of all of the ranges in the original gap's region of currency before k_1 and (k_1,k), and g_2's region of currency will consist of the union of all of the ranges in the original gap's region of currency after k_2 and (k,k_2). (Figure 14) If the key being inserted falls outside of the original gap's region of currency, g_1's region of currency will consist of the ranges in the original gap's region of currency before k and g_2's region of currency will consist of the all such ranges after k. Thus, the induction hypothesis is preserved in all representatives for **Insert** operations.

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.

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

---

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

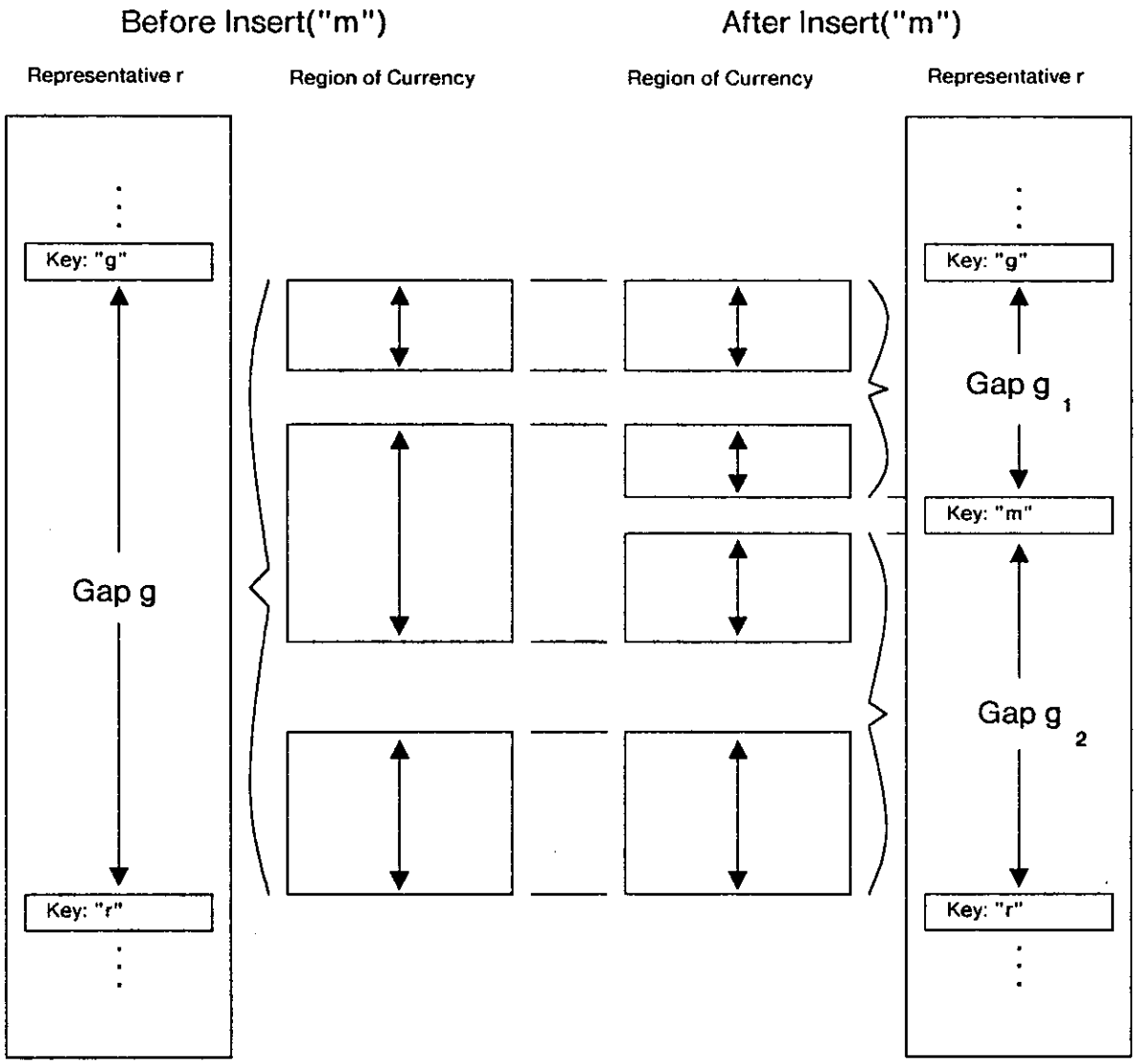

Figure 14: Effect of the **insert** operation on regions of currency, within write quorum

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 upwards from p consists of the ranges before p previously in the region of currency of the gap from which the new gap was split off. Similarly, if s is inserted, the gap extending downwards from s will have as its region of currency the ranges after s previously in the region of currency of the gap from which this gap was split off. (Figure 15). The keys p and s are, by definition, currently in the directory, so all of the gaps whose regions of currency are modified satisfy the induction hypothesis. Furthermore, all gaps whose region of currency previously contained a range bounded by k were modified in the fashion described above, so none of the gaps whose region of currency remains unchanged relies on the fact that k is in the directory in order to satisfy the induction hypothesis. Thus, the induction hypothesis holds within the write quorum.

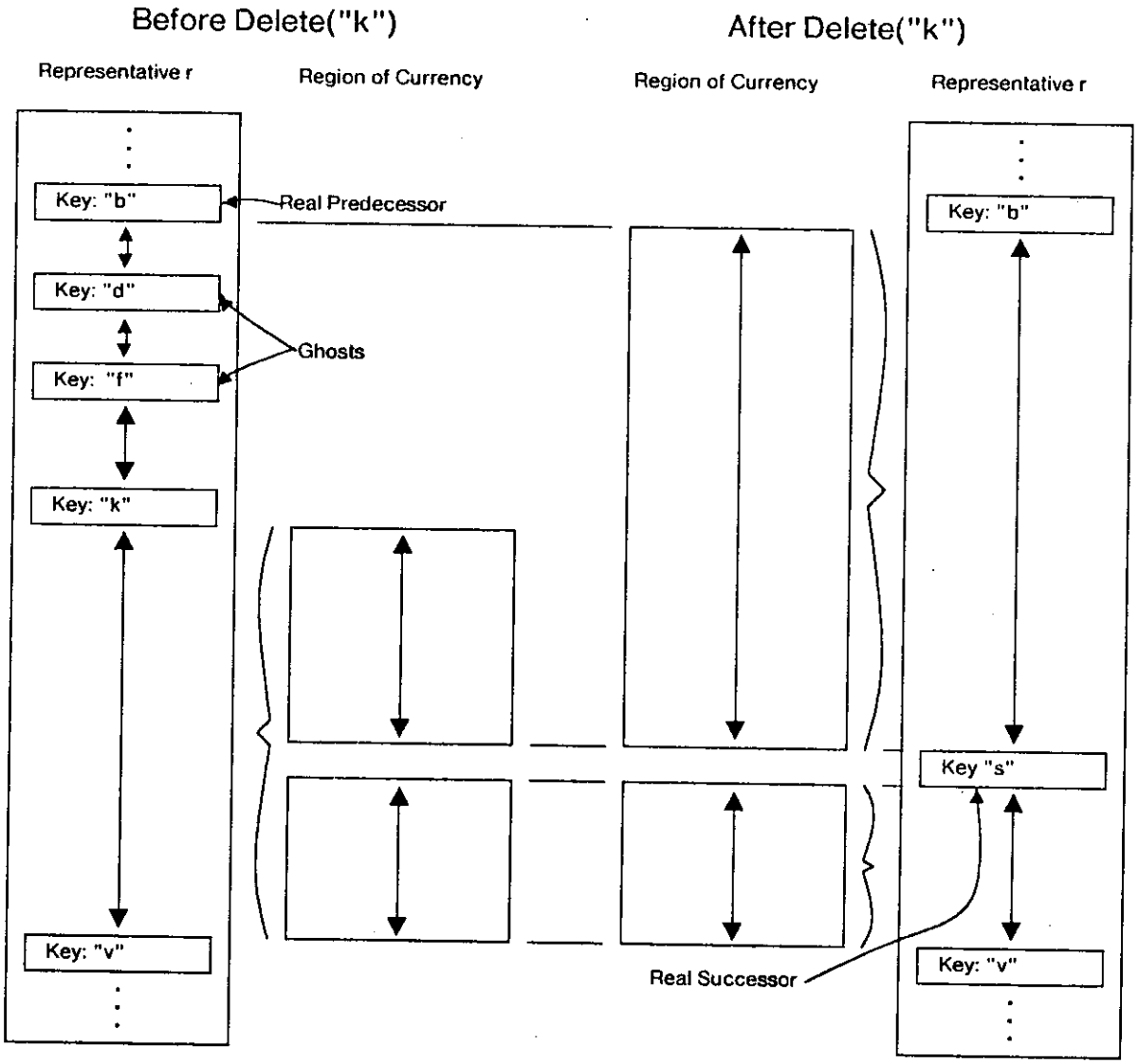

Figure 15: Effect of the **delete** operation on regions of currency, within write quorum

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. Any portions of other gaps' regions of currency that lie in this region are deleted from the gaps' regions of currency. This deletion has the effect of removing ranges entirely contained within (p,s). Any range that had k as one endpoint must have had p or s as its other endpoint, and must have fallen into this category. Thus the gaps outside of the write quorum in a **Delete** operation satisfy the induction hypothesis. This completes the proof.

We are now ready to describe the real predecessor algorithm. First, the node doing the real predecessor

determination for the key k asks each representative in the read quorum for the version number of the gap covering the key p immediately preceding k in the key space, and the entry delimiting the gap on the low side. The gap returned in response to this request with the highest version number is current over the region $\{p\}$. Let us call this gap g_{curr}. By the theorem, g_{curr} must be current over some range containing p that is bounded by keys currently in the directory. Thus, this "current range" must extend from p's real predecessor to p's real successor, exclusive. But p's real predecessor is also k's real predecessor, so we know that k's real predecessor lies in g_{curr} or on its boundary. Since any key (or gap) that intersects g_{curr} and has a higher version number than that of g_{curr} must lie outside of its region of currency, the closest such key (or gap) to k delimits the current range in which k lies, if such a key (or gap) exists. By the theorem, if it exists it is a key (not a gap) and it is k's real predecessor. If no such key exists, the current range extends all the way to g_{curr}'s low boundary, and the key delimiting it is k's real predecessor.

The node doing the determination proceeds by passing both g_{curr}'s version number and the key delimiting g_{curr} on the low side to each representative in the read quorum. The representatives return their entry closest to k that "supersedes" this gap (i.e. lies in the gap and has a higher version number than it does). If they have no entry that supersedes the gap then they check to see if they have an entry for the key delimiting the gap. If so, they return it; if not, they return a message saying that they have no candidate for the real predecessor. Then the node doing the determination merely selects the candidate closest to k. If several entries return candidates equally close, of course the one with the highest version number is selected. This entry is guaranteed to be the real predecessor. A formal statement of the algorithm is given in Figure 16.

3.4 Enhancements to the Real Predecessor Algorithm

As in the other procedures presented, efficiency is sometimes sacrificed for clarity in the **RealPredecessor** procedure of Figure 16. There are several additional improvements that would be made in any practical implementation of the algorithm. Firstly, the procedure would check if the second round of information exchange were necessary before doing it. If the closest 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 more by having each representative return several gaps and entries preceding the key being deleted rather than just one. The procedure would check if any entry for which it had information (entry or covering gap) from all representatives had a higher version number than any covering gap. If this were the case, then the closest such entry would represent the real predecessor, and no second stage would be 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

```
RealPredecessor(k:key) Returns(key,value,version,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,
    GapVer: array[1..R] of version,
    PredKeyVal: array[1..R] of value,
    PredKey: array[1..R] of key,
    MaxGapRep,i: integer,
    MaxGapVer,RealPredVer,CandVer: version,
    MaxGapKey,RealPredKey,CandKey: key,
    RealPredVal, CandVal: value;
begin
  quorum := CollectReadQuorum();
  {Collect info on predecessor gaps in each rep in the read quorum
   & find out which rep has the gap w/ the highest version number.}
  MaxGapVer :=  -1;  {Lower than any real version number}
  for i := 1 to R do
    begin
      GapVer[i],PredKey[i],PredKeyVer[i],PredKeyVal[i] :=
                        Send(DirRepPredecessor(k)) to quorum[i];
      if GapVer[i] > MaxGapVer then
        begin
          MaxGapRep := i;
          MaxGapVer := GapVer[i]
        end
    end;

  {Key delimiting Max gap is our initial candidate for real pred}
  RealPredKey := MaxGapKey := PredKey[MaxGapRep];
  RealPredVer := PredKeyVer[MaxGapRep];
  RealPredVal := PredKeyVal[MaxGapRep];
  {Find closest entry which supersedes Max gap in any rep in the
   read quorum. This will be the real predecessor. }
  for i := 1 to R do
    if i <> MaxGapRep then
      begin
        CandFlag,CandKey,CandVal,CandVer :=
          Send(DirRepSuperseder(k,MaxGapVer,MaxGapKey)
            to quorum[i];
        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[i] > RealPredKey
             or (CandKey = RealPredKey and CandVer > RealPredVer)) then
          begin  {Tentatively select the candidate}
            RealPredKey := CandKey;
            RealPredVal := CandVal;
            RealPredVer := CandVer
          end
      end;
  {Selected candidate is real predecessor. Return it.}
  Return(RealPredKey,RealPredVal,RealPredVer,MaxGapVer)
end
```

Figure 16: Real Predecessor Operation

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 it real predecessor will on average be half of the key's delete list size. Thus, the formula developed in Section 4.2.4 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 4.2.5 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 has no more information to add and need not participate in the second round.

Finally, 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 "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.

The algorithm, with the improvements described, is extremely fast in the average and worst cases. In fact, under an appropriate model it is optimal with respect to the number of fixed length messages required. However, the notion is somewhat difficult to formalize and a proof would be tedious. The average performance of this algorithm is close enough to the trivial lower bound of one exchange of messages with each member of a read quorum that there is no practical reason to attempt to prove optimality.

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.

3.5 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 and as long as there are no concurrency anomalies. 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 3.1 are strong enough to guarantee the serializability of transactions at any single representative. Traiger et al. [Traiger 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 is free from concurrency anomalies.

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.

3.6 More on Synchronization and Recovery

Directory representatives, as described in Section 3.1, are synchronized to ensure that all transactions using their operations can be made serializable.[8] 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 anomalies 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 are *non-serializable* [Schwarz 83a].

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

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 read 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 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 **RepModify** locks must be acquired temporarily to make certain that no active transactions have read (and therefore set **RepLookup** locks on) the old version numbers of the gaps being coalesced. 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. While it is an important example of the use of a non-serializable object, this change in locking rules increases concurrency only slightly because the only keys made accessible to concurrent transactions are those for which there are no entries in the directory. This change in locking rules also increases the chances of deadlock.

Finally, **RepLookup** locks on data beyond the real predecessor and 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. It should also be noted that the operations of inserting the real predecessor and real successor into representatives are additional examples of copying current data and therefore only **RepLookup** locks need be temporarily obtained for these insertions.

4 Performance Characterization

In this section, we present the results of simulations and construct and analyze a model of the directory replication algorithm. The system studied in both the simulations and the model consists of a directory suite initially containing a certain number of keys into which inserts, updates and deletes occur at regular intervals 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.

We concentrate on two performance measures. The first, 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.

The second performance measure, *delete list length*, is the average number of ghost entries in the range to be coalesced during a deletion on a representative. This measure indicates the amount of work that must be done at each node while searching for the real predecessor and successor, and while performing the coalesce operation during deletes. These two steps are the only parts of any of our procedures that do not run in constant time. Thus the delete list size characterizes the only non-obvious component of the time requirement of our algorithm.

4.1 Simulation Results

Figures 17 and 18 show the size ratios and delete list lengths measured in simulations for a variety of directory configurations. 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.

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 19. The duration of each of these simulations was two hundred thousand operations, with performance data gathered during the final one hundred thousand operations. These additional simulations indicate that none of the results depend on the initial number of keys in the directory suite; Thus, time and space requirements are proportional to the number of keys in the directory, just as in a single site directory.

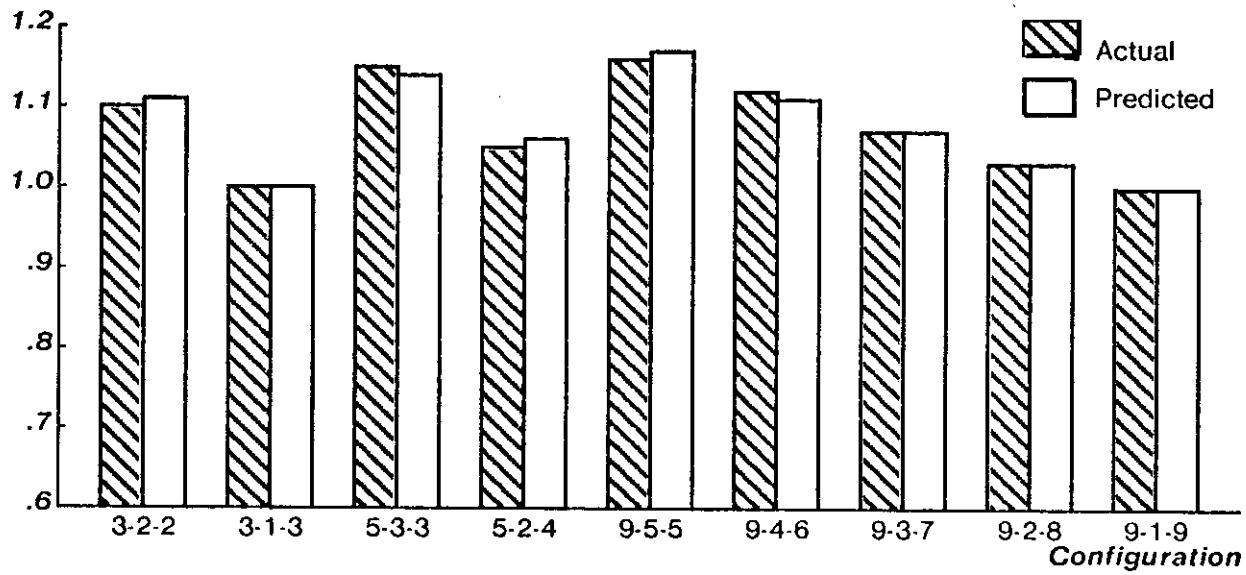

Figure 17: Size Ratios for Various Directory Suites

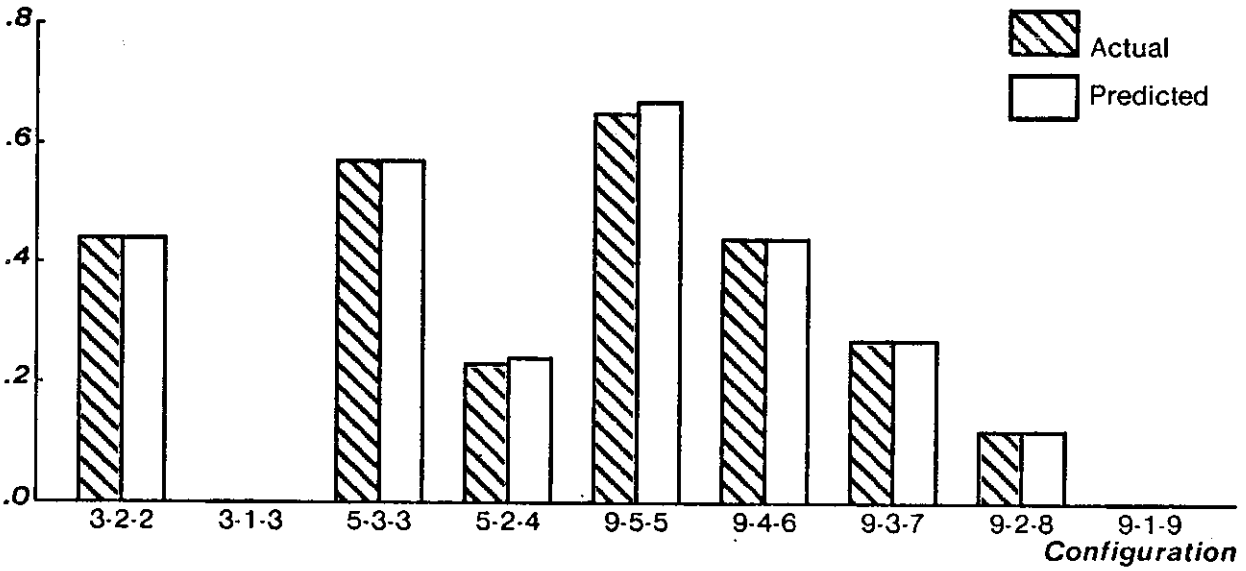

Figure 18: Delete List Lengths for Various Directory Suites

100 Entries			1000 Entries			10000 Entries		
			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 19: Detailed Simulation Results for three 3-2-2 Directory Suites

4.2 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 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.

4.2.1 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.

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.) It is desirable that our model belong to this class.

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 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. Attempts to lump states based on order relations between version numbers 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 completely, 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 classes that correspond to terms introduced previously. 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 extremely 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. The proof of this fact 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.

2. No useful information is "thrown away" in going from a system state to its concise representation. All of the system attributes we care about are fully determined by the concise representation of a system state.

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 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.

Finally, the model is irreducible. 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 state reachable from the initial state can be reached from every state.

The model achieves stochastic equilibrium, because it is Markovian, finite, and irreducible. There is one other property that the model must have in order to fulfill our requirements: all system states represented by each state must be "functionally identical" in the sense that they coincide in all attributes for which we wish to utilize an equilibrium distribution. However, this is precisely what property 2 tells us. In fact the attributes in question are for the most part aggregate information concerning the composition of a representative in terms of class. (The reader can easily check that each attribute for which we eventually require an equilibrium distribution is fixed over single states in our model.)

4.2.2 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 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.

These balance equations are naturally constructed in terms of three variables c', o' and d, and the system parameters N and W, defined in Section 4.2.3. 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. The desired performance measures can be derived from the variables, though we need to make a simplifying approximation in one derivation.

4.2.3 Formulation of Balance Equations

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

$\mathcal{C}$ — The number of current entries in an arbitrary (but fixed) representative.

$\mathcal{O}$ — The number of outdated entries in an arbitrary (but fixed) representative.

$\mathcal{G}$ — The number of ghost entries in an arbitrary (but fixed) representative.

$\mathcal{E}$ — The total number of entries in an arbitrary (but fixed) representative. 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 an arbitrary (but fixed) representative. The delete list of a key consists of all of the ghost entries between the *real predecessor* and *real successor* of the key in the representative.

$\mathcal{D}$ — $(\sum_{k \in \text{Suite}} \mathcal{D}_k)/\mathcal{K}$. $\mathcal{D}$ is the average delete list size in an arbitrary (but fixed) representative. 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:

$$(1+\tfrac{W}{N})c' + \tfrac{W}{N}o' = 2\tfrac{W}{N}$$

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

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

4.2.4 Solution of Balance Equations

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

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

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

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

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:

$E[\mathfrak{S}/\mathfrak{N}]$
$= E[(\mathfrak{C}+\mathfrak{O}+\mathfrak{G})/\mathfrak{N}]$
$= E[\mathfrak{C}/\mathfrak{N}] + E[\mathfrak{O}/\mathfrak{N}] + E[\mathfrak{G}/\mathfrak{N}]$
$= c' + o' + E[\mathfrak{G}/\mathfrak{N}].$

The three terms of this expression ($E[\mathfrak{C}/\mathfrak{N}]$, $E[\mathfrak{O}/\mathfrak{N}]$ and $E[\mathfrak{G}/\mathfrak{N}]$) 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:

$$2\mathcal{G} = \sum_{k \in \text{Suite}} \mathcal{D}_k.$$

Dividing both sides of this equation by $2\mathcal{H}$ and taking expected values we get:

$$\begin{aligned} E[\mathcal{G}/\mathcal{H}] &= E[(\sum_{k \in \text{Suite}} \mathcal{D}_k)/2\mathcal{H}] \\ &= \tfrac{1}{2} E[\mathcal{D}] \\ &= \tfrac{d}{2}. \end{aligned}$$

Substituting back, our formula for the size ratio becomes:

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

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

4.2.5 Results

Figure 17 (p. 28) compares the average size ratios observed in the simulations with predictions obtained from the formula developed in the previous section. Figure 18 (p. 28) 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 20 shows the predicted average composition ratios in a $10 - (11-W) - W$ suite, for all possible values of W. Figure 21 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 while decreasing the read quorum size increases the cost of the write operation and the availability of the system and decreases the cost of the read operation. 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 ($w=10$) there is the universal update strategy; at the other ($w=1$), there is a strategy where only a single representative is written and all are read. While the latter strategy would never be used, because <u>all</u> directory operations require reading from a read quorum, it is interesting to see how performance varies over the spectrum. From these graphs, one can see that even when the system is stretched to an unreasonable extreme, performance does not degrade very much.

Figures 22 and 23 show respectively the predicted average composition ratios and delete list lengths in $(2i-1) - i - i$ suites. Increasing read quorum, write quorum and suite sizes simultaneously, as illustrated in these graphs represents a fairly straightforward performance tradeoff: As the sizes increase, the availability of the suite 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 of a $(2i-1) - i - i$ suite is $i-1$. The graphs show that the amount of work at each node for a **Delete** operation, and the size and makeup of each representative do not vary appreciably over the spectrum.

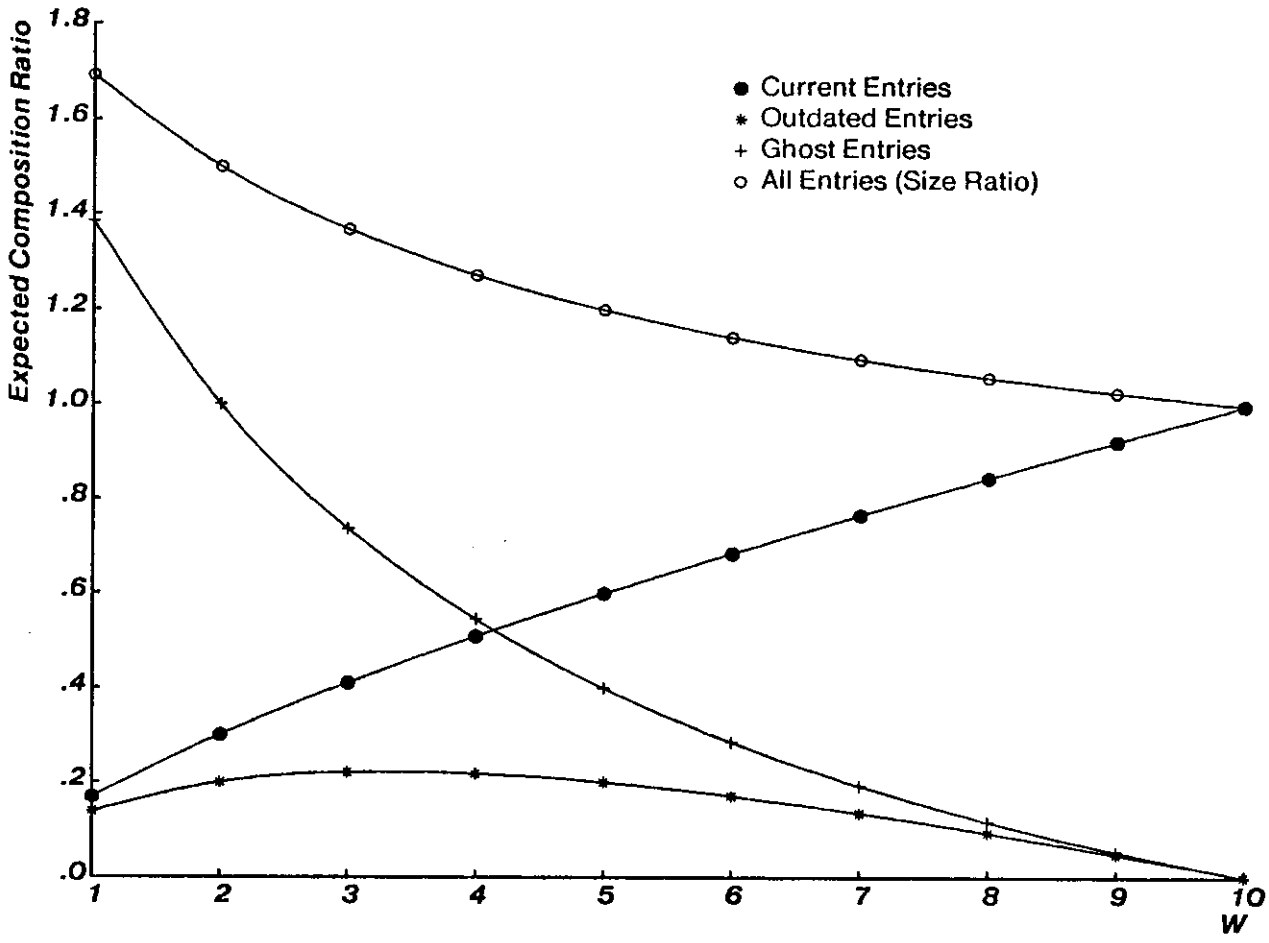

Figure 20: Expected Composition Ratios in a 10 - (11 – W) - W Suite

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 $1 \leq W \leq N$. As we would expect, this expression grows when the suite size increases and when the write quorum decreases. Thus the expression approaches its maximum when N tends to infinity and W is set to 1, its lowest permissible value. So:

$$d < \lim_{N \to \infty} \frac{4N-4}{N+3} = 4.$$

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

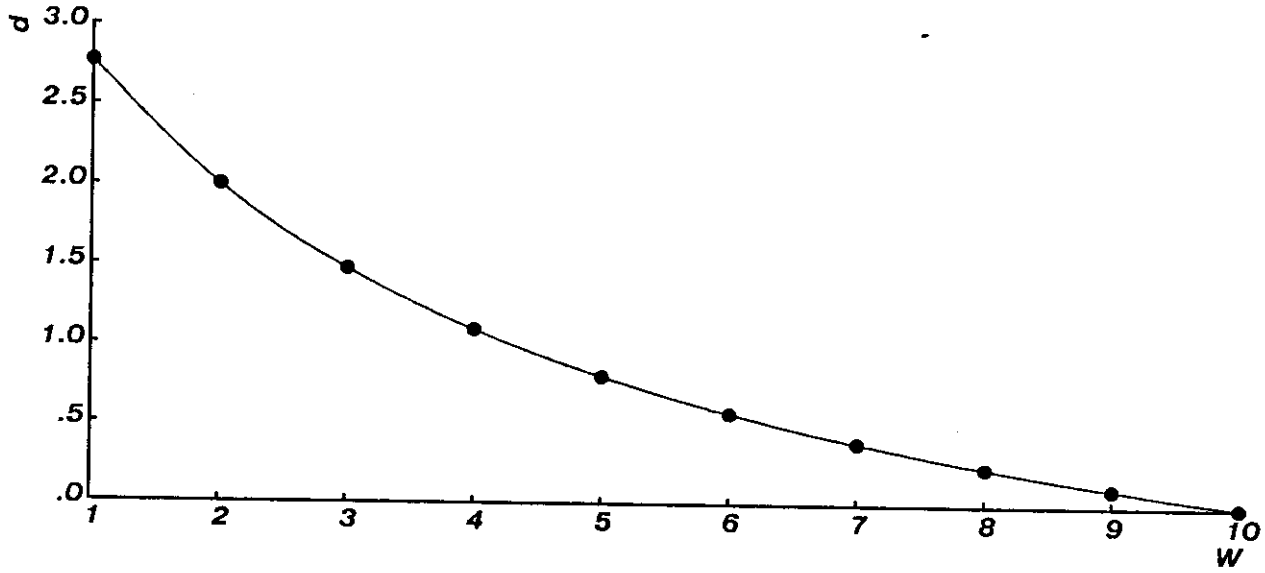

Figure 21: Expected Delete List Lengths in a 10 - (11 – W) - W Suite

A similar result holds for the size ratio ($E[\mathcal{S}/\mathcal{K}]$). 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 approaches its maximum when $W=1$ and N tends to infinity. Thus its value is bounded by:

$$\lim_{N \to \infty} \frac{2N+2}{N+3} = 2.$$

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

4.2.6 Discussion of the model

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. 44)

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. 45)

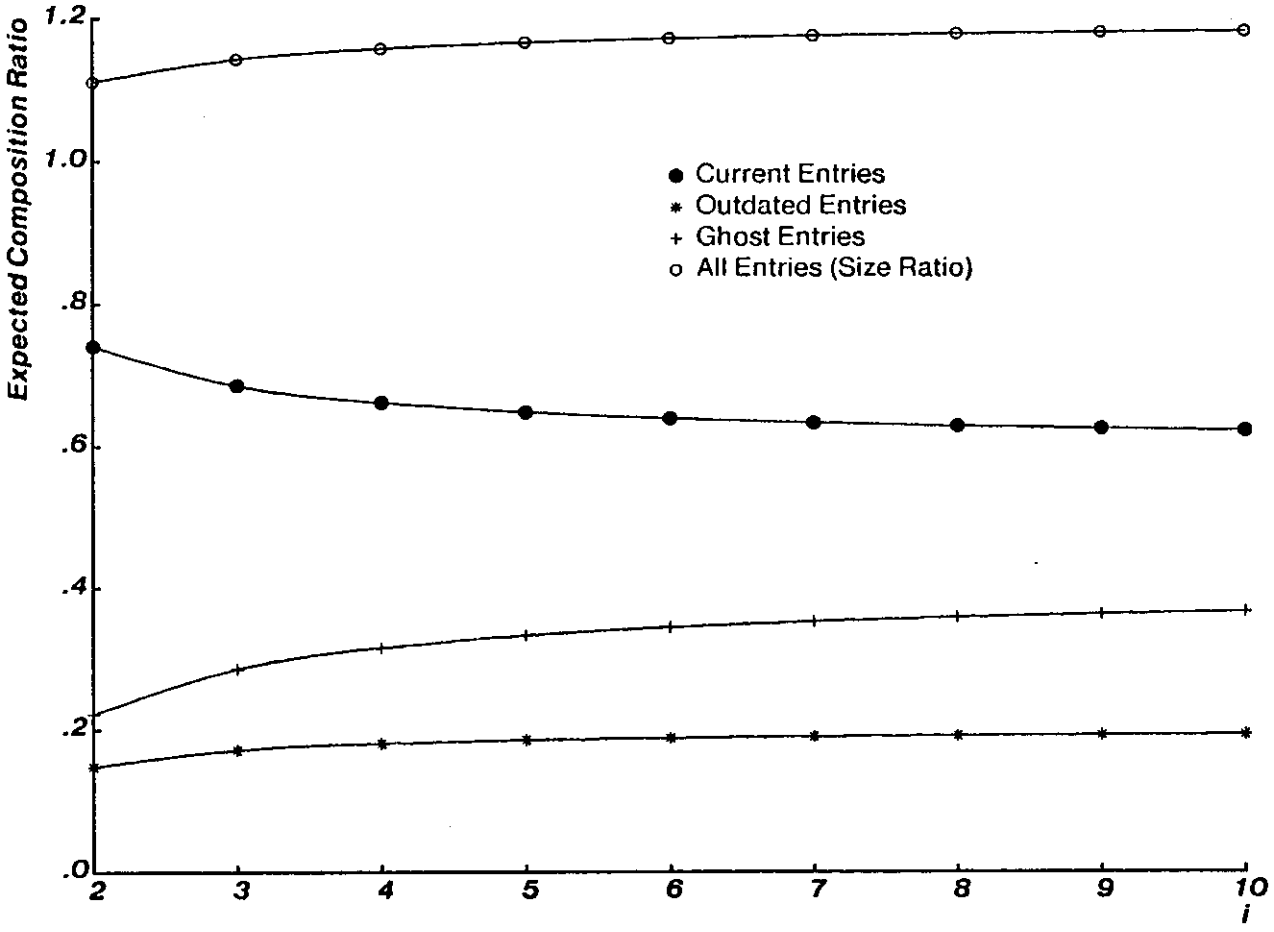

Figure 22: Expected Composition Ratios in a $(2i-1)$ - i - i Suite

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. 47, 48)

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. 34)

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.

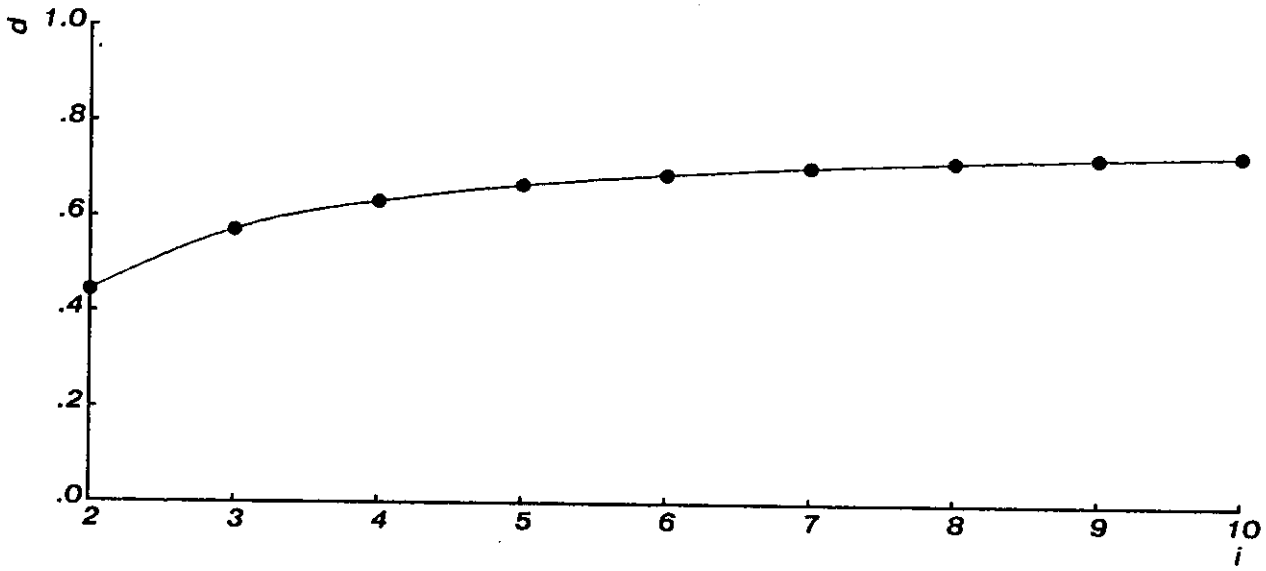

Figure 23: Expected Delete List Lengths in a $(2i-1)$ - i - i Suite

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 has an entry in this representative differs by at most $1/\mathcal{K}$ 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, $\mathcal{K}$ 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. In fact, 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.

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 $\mathcal{D}/2$ will be almost identical to $\mathcal{G}/\mathcal{K}$. 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 key space size enters into our analysis. It was not used explicitly in any of the equations.) None of the assumptions break down when N or W gets 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 ($\mathcal{K}$) from run to run. The observed average values for $\mathcal{K}$ 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 key space size, and the simulations were run for far fewer steps than the key space size 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, it is likely that our model 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.

4.3 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. However, we strongly conjecture that the performance observed under these conditions would 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 **Insert**, **Update**, and **Delete** 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 show that as the value of p decreases, the average delete list size decreases significantly from the value observed under random usage. An average delete list size of 0.44 was observed when the value of p was 0.1, 0.25 when the value of p was 0.01, 0.28 when the value of p was 0.001, and 0.02 when the value of p was 0.0001. 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 tend 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 any realistic usage pattern as it was under the random usage studied in the simulations and analysis.

5 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.

Weighted voting 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 intra-suite parallelism. There are no easy solutions to the problems caused when a large collection of operations simultaneously attempt to update information associated with the same key; however, any directory may be statically partitioned into separate sub-directories in which concurrent operations can take place. Such sub-directories can be represented as directory suites. Terry [Terry 84] has analyzed the performance of various directory partitioning schemes for **Lookup** operations.

Directory suites can also 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.[9] 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 24, 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.

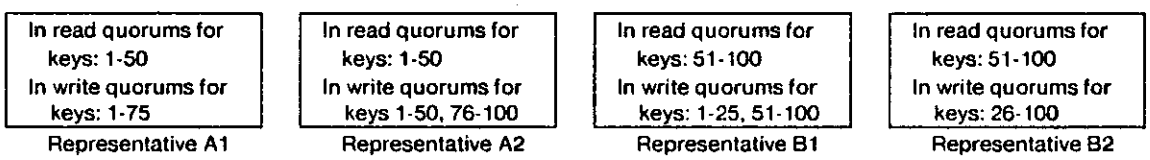

Figure 24: A 4-2-3 Directory Suite Partitioned for Locality

The ways in which this algorithm will actually be used will become known once implementations are available. We have begun an implementation and resolved some details not addressed in this paper. For

[9] Of course, failures that require the quorums to change will result in a performance loss.

example, our implementation stores data for directory representatives as B-trees [Comer 79], and version numbers for gaps are stored in fields in their bounding entries. We envision using version numbers containing 48 or more bits to prevent cycling. When completed, the implementation will run on a transaction-based system that we are building on a modified version of the Accent kernel [Rashid 81, Spector 83b, Spector 83a]. Our transaction manager uses write ahead log protocols described by Schwarz and Spector [Schwarz 83b] for recovery from failures.

In summary, we have presented a replication algorithm for directories that exhibits favorable performance and availability properties. As is the case with Gifford's algorithm, the exact configuration of suites can be tailored to provide higher or lower availability, and higher or lower performance. 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 have proven a property of directory suites that permits all operations, including deletions, to be done in a small number of messages that depends only on the size of the read and write quorums. Both simulation and analytic results show that the time and space costs associated with using our algorithm are low.

Acknowledgments

James Driscoll suggested improvements to our initial dynamic partitioning algorithm that resulted in the algorithm presented in this paper. John Lehoczky provided invaluable assistance in the definition and analysis of our analytic model. 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 suite 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. Of course 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 − P[P]) + (1 − 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 huge 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:

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

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:

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

Simplifying, this becomes:

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

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:

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

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 47 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:

P[The representative is chosen for the write quorum]
× E[The size of the delete list of the the key being deleted]
= P[The representative is chosen for the write quorum]
× E[The size of the delete list of the a randomly chosen key in the directory]
$= \tfrac{W}{N}d$.

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

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

Simplifying:

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

[Allchin 83a] J. E. Allchin, M.S. McKendry.
Synchronization and Recovery of Actions.
In *Proc. of the Second Principles of Distributed Computing Conference*, pages 31-44. August, 1983.

[Allchin 83b] James E. Allchin, Martin S. McKendry.
Facilities for Supporting Atomicity in Operating Systems.
Technical Report GIT-CS-83/1, Georgia Institute of Technology, January, 1983.

[Alsberg 76] P. A. Alsberg, J. D. Day.
A Principle for Resilient Sharing of Distributed Resources.
In *Proc. 2nd International Conf. on Software Engineering*, pages 562-570. October, 1976.

[Bartlett 81] Joel Bartlett.
A NonStop™ Kernel.
In *Proc. Eighth Symp. on Operating System Principles.* ACM, 1981.

[Birman 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 Unviersity, 1983.

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

[Daniels 83] Dean Daniels, Alfred Z. Spector.
An Algorithm for Replicated Directories.
In *Proc. of the Second Principles of Distributed Computing Conference.* August, 1983.

[Gifford 79] David K. Gifford .
Weighted Voting for Replicated Data.
In *Proc. Seventh Symp. on Operating System Principles*, pages 150-162. ACM, 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 81] James N. Gray, et al.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys (2):223-242, June, 1981.

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

[Kemeny 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), Jaunary, 1983.

[Lindsay 79] Bruce G. Lindsay, et al.
Notes on Distributed Databases.
IBM Research Report RJ2571, IBM Research Laboratory, San Jose, Ca., July, 1979.

[Liskov 82] Barbara Liskov and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In *Proceedings of the Ninth ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 7-19. Albuquerque, NM, January, 1982.

[Popek 81] G. Popek et al.
LOCUS: A Network Transparent, High Reliability Distributed System.
In *Proc. Eighth Symp. on Operating System Principles.* ACM, 1981.

[Rashid 81] Richard Rashid, George Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. Eighth Symp. on Operating System Principles.* ACM, 1981.

[Rothnie 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 83a] Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Carnegie-Mellon Report CMU-CS-83-163, Carnegie-Mellon University, Pittsburgh, PA, November, 1983.
Revised edition of CMU-CS-82-128.

[Schwarz 83b] Peter M. Schwarz, Alfred Z. Spector.
Recovery of Shared Abstract Types.
Carnegie-Mellon Report CMU-CS-83-151, Carnegie-Mellon University, Pittsburgh, PA, October, 1983.

[Spector 83a] 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 Carnegie-Mellon Report CMU-CS-82-143, January 1983.

[Spector 83b] Alfred Z. Spector.
Modifying the Accent Kernel to Support TABS Recovery.
November, 1983.

[Terry 84] Douglas B. Terry.
An Analysis of Naming Conventions for Distributed Computer Systems.
In *Proceedings of SIGCOMM 84 Symposium: Communications Architectures and Protocols.* June, 1984.
To appear. Also available as Department of Electrical Engineering and Computer Science Technical Report UCB/CSD/83/156, University of California, Berkeley.

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

[Weihl 83b] William E. Weihl.
Data Dependent Concurrency Control and Recovery.
In *Proc. of the Second Principles of Distributed Computing Conference*, pages 73-74. August, 1983.