NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

An Algorithm for Replicated Directories

Dean Daniels and Alfred Z. Spector

May 31, 1983

Abstract

This paper describes a replication algorithm for directory objects based upon Gifford's weighted voting for files. The algorithm associates a version number with each possible key on every replica and thereby resolves an ambiguity that arises when directory entries are not stored in every replica. The range of keys associated with a version number changes dynamically; but in all instances, a separate version number is associated with each entry stored on every replica. The algorithm exhibits favorable availability and concurrency properties. There is no performance penalty for associating a version number with every possible key except on Delete operations, and simulation results show this overhead is small.

Copyright © 1983 Dean Daniels and Alfred Z. Spector

Technical Report CMU-CS-83-123

This work was sponsored in part by: the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

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

1 Introduction

Object replication on distributed computing systems has the goals of increased parallelism, reduced communications costs, and increased resilience to 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 increased performance and reliability while ensuring that the semantics of replicated data objects are identical with their non-replicated counterparts.

This paper presents a scheme for replicating directories that permits concurrent operations and arbitrarily high data availability. The semantics of the replicated directory are typical of directories that are stored on a single site. Briefly, directories contain a collection of *entries*, each of which contains a (*key*, *value*) pair with a unique key. The replicated directory has operations similar to the following: **Lookup(K:Key) Returns(Boolean, Value)**, **Insert(K:Key, V:Value)**, **Update(K:Key, V:Value)**, and **Delete(K:Key)**. Trivial modifications of this algorithm may be used to implement sets or similar abstractions.

The replication algorithm that we present is similar to Gifford's weighted voting algorithm [Gifford 79, Gifford 81], and thus, has the same performance and reliability advantages. However, unlike Gifford's algorithm, our algorithm uses a new technique to associate a version number with each possible key at every replica. This technique 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 a larger research project studying distributed systems that use a transaction facility to support operations on shared abstract data types [Schwarz 82, Spector 83]. The replicated directory described in this paper is an example of a distributed abstract data type whose construction is facilitated by having a flexible underlying transaction mechanism available. Additional components of our research address synchronization, recovery, and communication issues. Groups at MIT and Georgia Institute of Technology are also investigating the wider use of transactions [Liskov 82, Weihl 83, Allchin 82, Allchin 83].

In the following section of this paper, we survey related replication work and motivate the development of our algorithm. We then describe the algorithm in detail and present performance data that we obtained via simulation. Finally, we discuss additional ways to make the replication algorithm function with greater efficiency and concurrency.

2 Related Work and Motivation

This section discusses the application of existing replication algorithms to the problem of replicated directories, and informally develops the proposed replication strategy. First, unanimous update and primary/secondary copy strategies are briefly discussed. (See Lindsay for a brief survey of these strategies [Lindsay 79].) Then, weighted voting is considered and adapted for use in directory replication.

In the unanimous update strategy, 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. There have been attempts to increase update availability by using the communication system to buffer updates to replicas that are not available. The SDD-1 distributed database system uses an approach like this [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. 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 lessening this problem have been developed; for example, the Locus system uses a synchronization site [Popek 81].

Gifford designed a strategy for replication of files, which is based on a scheme called *weighted voting* [Gifford 79, Gifford 81]. This algorithm assigns some number of votes and a version number to each *representative* (or replica) of a replicated *file suite*. Write operations modify each representative in a *write quorum* of W votes and increment the version number of each representative in the quorum. Read operations read from each representative in a *read quorum* of R votes and return data from the representative with the largest version number. The sizes of the read and write quorums are chosen so that $R+W$ is greater than the sum 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.

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. A unanimous update strategy may be specified if desired. Second, representatives with zero votes may be used as hints [Lampson 79]. Third, consistency and recovery are mainly the responsibility of transactional storage systems, which are assumed to hold each representative. Because concurrent operations are synchronized by the transaction system storing each representative, there can be considerable flexibility in the specification and implementation of concurrency control.

While weighted voting is an appealing approach to directory replication, the basic algorithm can not be applied to directories without undesirable concurrency limitations. Even though the semantics of directory operations permit concurrent modifications to different entries, only a single transaction could modify the directory at any time if a directory were stored as a replicated 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. However, with such an approach, representatives might not have a version number for an entry that is stored on other representatives. Because of this, it may not be possible to examine an arbitrary read quorum and determine whether an entry for a particular key exists.

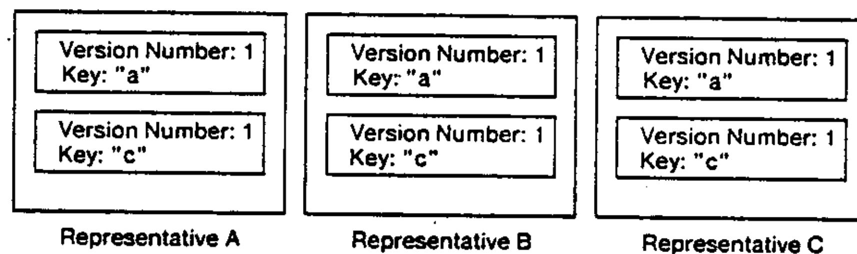

Figure 1: A 3-2-2 Directory Suite - Initial Configuration

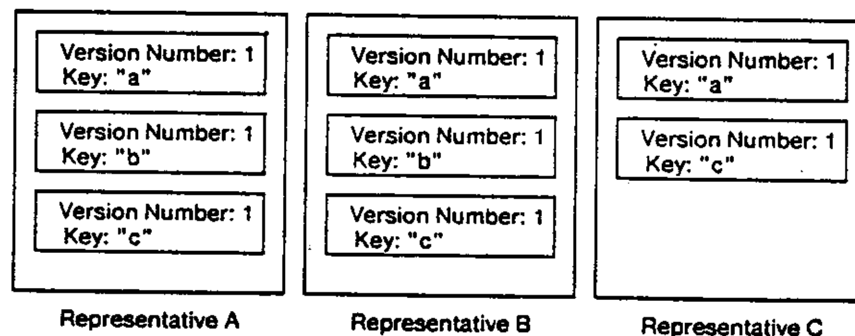

Figure 2: Directory Suite After Inserting "b"

For example, 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.[1] Initially, each representative in the suite contains entries "a", and "c", and

[1] The notation x-y-z will refer to a directory having x representatives, a read quorum of y and a write quorum of z. For simplicity, all examples in this paper assume that each representative is assigned one vote.

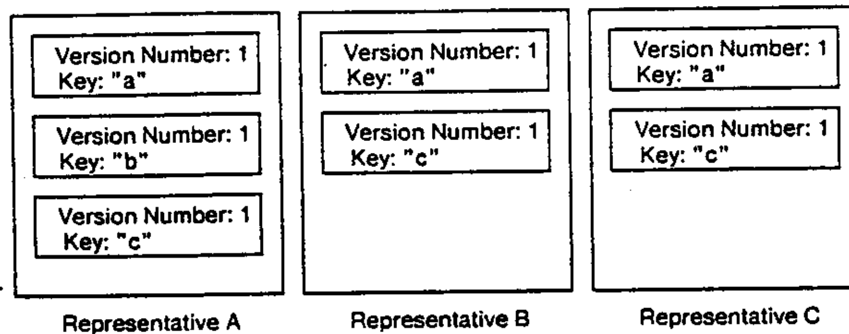

Figure 3: Directory Suite After Deleting "b"

each entry has version number 1 as in Figure 1[2]. Subsequently entry "b" is inserted into representatives A and B with version number 1 (Figure 2). If a "Lookup("b")" request is sent to representatives A and C at this point, representative A will respond with "present with version number 1", and representative C will reply "not present". If entry "b" is then deleted from representatives B and C (Figure 3), "Lookup("b")" requests to representatives A and C will still elicit "present with version number 1", and "not present" responses. 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 there is an entry in the directory suite for a given key.

The ambiguity demonstrated above is associated with deletions and will not occur if deletions are not permitted. Entries could be updated to indicate that they are "deleted", but the space occupied by "deleted" entries could not easily be reclaimed. An alternative strategy is to eliminate the ambiguity by consulting an additional representative whenever one representative replies "present with version number x" and another representative replies "not present." This approach may be applied to any directory suite configuration, but it results in reduced availability.

As has been demonstrated, associating a version number only with existing entries fails to capture important information about the version numbers of keys for which there are not entries. If, however, a single version number per representative is used, concurrency is limited. A solution is to partition the space of possible keys and to associate a separate version number with each partition.

A directory could be partitioned 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. Such a directory keeps a version

[2]The *value* field is omitted from all figures to save space.

number with each entry and keeps an additional version number for use with "not present" responses. Under such a partitioning, deletions must 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 single version number per representative approach. Alternatively, deletions could be implemented by marking entries to be deleted and then performing a "garbage collection" operation periodically. However, that operation is complex and would itself be a concurrency bottleneck.

This paper will consider partitioning the key space into a set of disjoint ranges by imposing an ordering relation on the keys. The simplest approach is to use a static partitioning; however, the additional concurrency that is achieved might be less than expected. If a small number of ranges were used, then at most that number of transactions could modify a directory concurrently. Also, if transactions modify entries in more than one range, concurrency will be further limited. Even if a large number of ranges were used, an uneven distribution of accesses could limit concurrency.

Below, we concentrate on a technique in which the ranges of keys associated with version numbers change dynamically. A dynamic technique such as this might be desirable for directories having sizes or access patterns that vary widely over time. In this dynamic approach, each directory entry, and, consequently, its key, is in a range by itself with its own version number. Each range of keys between directory entries, called a *gap*, is a separate range with a separate version number.

Because each entry in a directory representative is in a range by itself, lookup operations on such entries return the version number associated with the entry. Lookup operations on keys not in a directory representative return the version number of the gap in which the key appears. Update operations increment the version number of the range containing the entry being updated; insertion operations split a gap; and deletions coalesce the gaps and entries in a range of keys into a single gap. For example, using this approach, entry "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 "Lookup("b")" request were sent to representatives A and C at this point, representative A will respond with "present with version number 1," and representative B will reply "not present with version number 0." Using these responses, a client may determine that there is an entry for "b" since that response has the larger version number. If "b" is subsequently deleted from representatives B and C, then the two gaps on either side of "b" on representative B are coalesced; then on both representatives, the gap between "a" and "c" is assigned version number 2. (Figure 5).

---

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

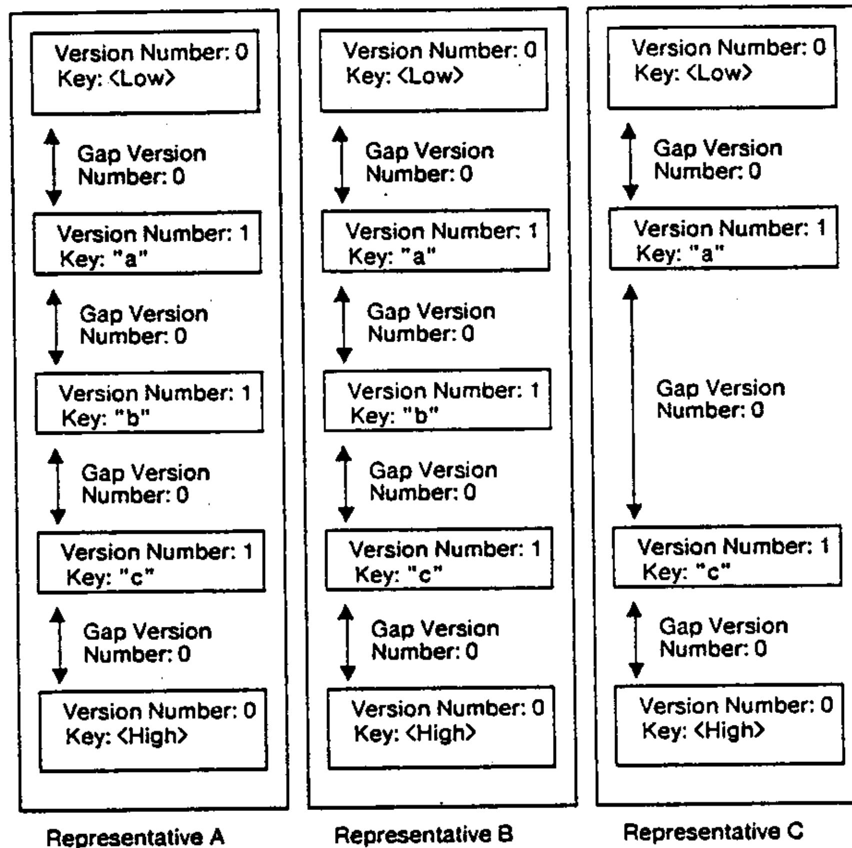

Figure 4: Directory Suite After Inserting "b"

The following section discusses this replication algorithm in more detail.

3 Details of the Algorithm

This section presents the details of the approach to directory replication sketched in the previous section. The descriptions given here 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 are similar in semantics to those of ARGUS [Liskov 82], except that error responses, such as timeouts, are not considered in these examples. Clarity is emphasized over performance in these descriptions and an inventive reader will find many improvements.

There are three parts to the descriptions given here. First, the operations on directory representatives are

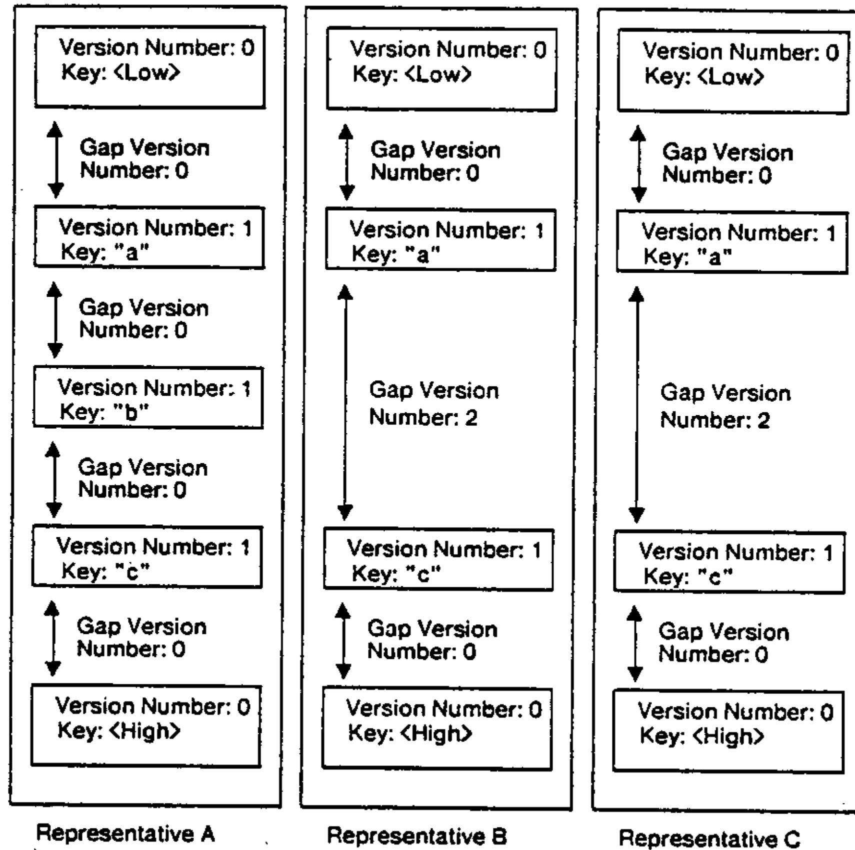

Figure 5: Directory Suite After Deleting "b"

identified. Second, the operations on directory suites are described and illustrated and finally, some correctness arguments are given.

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 on its file representatives.

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 successor and real

```
DirRepLookup(x:key) Returns(boolean,version,value);
{ If there is an entry for x return TRUE, the
  version number of the entry, and its
  value; otherwise return 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);
{ Returns the key and version number of the
  entry with smallest key greater than x.
  Also returns the version number of the gap
  between x and its successor.  There need
  not be an entry for x.

  Locks RepLookup(x,y) where y is the key
  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

predecessor in the directory. *Real predecessor* and *real successor* have an intuitive meaning, but are defined precisely in Section 3.2.

Directory representatives provide typical directory primitives: **DirRepLookup** and **DirRepInsert**. In addition, directory representatives provide specialized operations that are used to implement the directory

suite deletion operation: DirRepPredecessor, DirRepSuccessor, and DirRepCoalesce. 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. Figure 6 gives sample procedure headings for each of these operations.

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 [Schwarz 82]. 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 analogs of the lock classes for a single-copy directory (given by Schwarz [Schwarz 82]). 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, and DirRepSuccessor) 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.

As specified, the lock compatibility relation is sufficiently strong to guarantee that the actions of transactions operating on a directory representative are serializable [Traiger 82], providing that two phase locking is used. This form of synchronization simplifies correctness arguments given in Section 3.3.

Lock Requested	Lock Held		
	None	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

3.2 Directory Suites

Directory suites consist of a set of directory representatives, a distribution of votes, 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. A Directory suite implements the operations DirSuiteLookup, DirSuiteInsert, DirSuiteUpdate, and DirSuiteDelete.

The DirSuiteLookup operation sends **DirRepLookup** requests to a read quorum of representatives and returns the results[4] of the reply with the largest version number. Code for this operation is given in Figure 8.

Directory suite modification operations must ensure that the version number of the modified entry is higher than any version number that had been previously associated with the entry's key. In addition, the DirSuiteDelete operation must exercise care so that it does not inadvertently give a higher version number to non-current data.

The DirSuiteInsert operation is quite simple. DirSuiteInsert first looks up the key to be inserted in a read quorum and uses one greater than the highest version number as the version number for the new entry. The entry is then inserted in a write quorum of representatives. Figure 9 illustrates this operation. The DirSuiteUpdate operation is analogous.

DirSuiteDelete must delete an entry from a write quorum by coalescing a range of keys that includes the entry to be deleted and assigning a higher version number to the resulting gaps. To avoid assigning higher version numbers to data that is not current, the range to be coalesced may not contain directory suite entries

[4] Figure 8 shows DirSuiteLookup returning a version number as well as a boolean and the value of the entry. The version number is used by the procedures RealPredecessor, DirSuiteInsert, and DirSuiteModify. A user would ignore this number.

```
DirSuiteLookup(x:key) Returns(boolean,version,value)

var
   { read quorum has R members }
   quorum : array[1..R] of DirRep;
   v, bestv : version;
   val, bestval : value;
   isin, bestisin : boolean;
   i : integer;

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

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

Figure 8: DirSuiteLookup Operation

other than the one to be deleted. To possess this property, the range must extend from the *real predecessor* of the key to be deleted to its *real successor*. The real predecessor of a key, *x*, is the entry with the largest key less than *x* that appears in a write quorum of representatives. The real successor of a key is defined similarly.

Locating the real predecessor and real successor of an entry that is to be deleted is complex. There may be *ghosts* of entries located between the deleted key 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 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.

```
DirSuiteInsert(x:key,z:value);

var
  { write quorum has W members }
  quorum : array[1..W] of DirRep;
  i : integer;
  k : key;
  v : version;
  val : value;
  isin: boolean;

begin
  { first, lookup the key to find the          }
  { current version number                     }
  isin,ver,val:= DirSuiteLookup(x);
  { 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 DirSuiteLookup call }
  ver:=ver+1;

  { insert the entry in each quorum member }
  for i:= 1 to W do
    Send(DirRepInsert(x,ver,z))
      to(quorum[i]);

end; {of DirSuiteInsert}
```

Figure 9: DirSuiteInsert Operation

A straightforward implementation of the procedure RealPredecessor, which locates the real predecessor of a key, is shown in Figure 12. Because of ghost entries, this procedure may have to examine many keys before finding the real predecessor. However, measurements reported in Section 4 indicate that this is not a problem in practice. The DirSuiteDelete operation uses this procedure and the analogous procedure: RealSuccessor. DirSuiteDelete locates the real successor and real predecessor of an entry to be deleted, and inserts entries for the real successor and real predecessor into any member of the write quorum where they do not appear. It then determines the version number to be assigned to the new gap and coalesces the range in each member of the write quorum. DirSuiteDelete is illustrated in Figure 13.

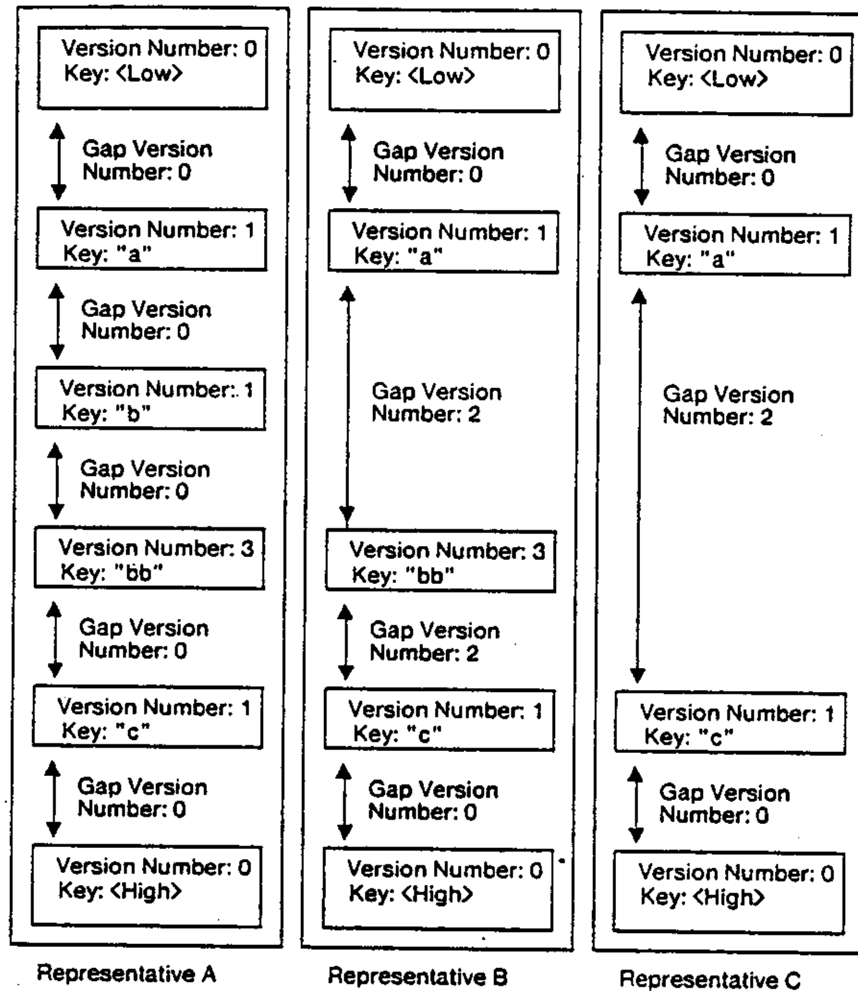

Figure 10: Directory Suite from Figure 5 After Inserting "bb"

3.3 Correctness Arguments

The correctness of a directory suite's operations depends on **DirSuiteLookup** always returning current information about a key. Because every read quorum intersects every write quorum, **DirSuiteLookup** 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 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.

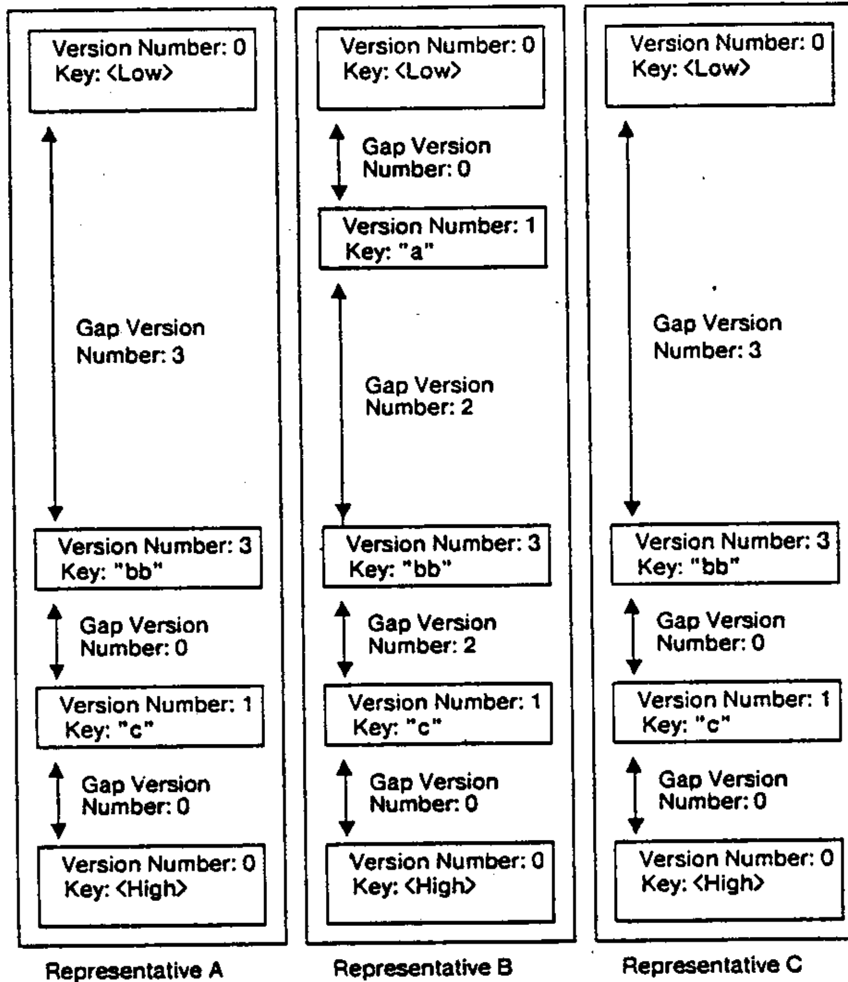

Figure 11: Directory Suite from Figure 10 After Deleting "a"

The DirSuiteInsert and DirSuiteUpdate 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.

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

```
RealPredecessor(x:key)
  Returns(key,value,version,version);
  {returns key, value, and version number }
  { of x's real predecessor, and the largest}
  { gap version encountered while searching }

var
  {read quorum has R members }
  quorum array[1..R] of DirRep;
  pred, k, pk: key;
  pver, tv, v, vt, maxv: version;
  pvalue: value;
  i: integer;
  isin: boolean;

begin
{ collect a read quorum }
  quorum:=CollectReadQuorum();
  k:=x;
  isin:=false;
  maxv:=LowestVersion; {a constant}
  while not isin do
    begin
      pred:=LowestKey; {a constant }
      for i:=1 to R do
        begin
          pk,tv,v:=Send(DirRepPredecessor(k))
            to(quorum[i]); { tv, ignored }
          pred := Max(pk, pred);
          maxv := Max(v, maxv);
        end; {of for i}
      isin,pver,pvalue:=DirSuiteLookup(pred);
      if not isin then
        k:=pred;
    end; {of while do }
  Return(pred,pvalue,pver,maxv);
end; {of RealPredecessor }
```

Figure 12: RealPredecessor Operation

4 Performance Characterization

This section presents the results of simulations of this directory replication strategy. There are many statistics that characterize the performance of this algorithm, but only three were selected for the measurements presented here.

The first statistic is labeled "Entries in ranges coalesced" and is the average number of entries (per representative) that lie between the real predecessor and real successor of a deleted key. This statistic counts the entry to be deleted, if it appears in a representative, and any ghosts that may be in the range to be

```
DirSuiteDelete(x:key);

var
  { write quorum has W members }
  quorum : array[1..W] of DirRep;
  i : integer;
  isin: boolean
  succ, pred, k: key;
  pval, sval, val: value;
  pver, sver, v, ver: version;

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

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

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

  { make sure the predecessor and successor }
  { exist in every member of the quorum)
  for i := 1 to W do
    begin
      isin,v,val:= Send(DirRepLookup(succ))
                        to(quorum[i]);
      {v,val ignored}
      if not isin then
        Send(DirRepInsert(succ,sver,svalue))
          to (quorum[i]);
      isin,v,val:= Send(DirRepLookup(pred))
                        to(quorum[i]);
      {v,val ignored}
      if not isin then
        Send(DirRepInsert(pred,pver,pvalue))
          to (quorum[i]);
    end; { for i }

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

Figure 13: DirSuiteDelete Operation

coalesced. Entries for the real predecessors and real successors are not included. This statistic reflects the number of entries that must be examined when the DirSuiteDelete operation is locating the real predecessor and real successor of a entry.

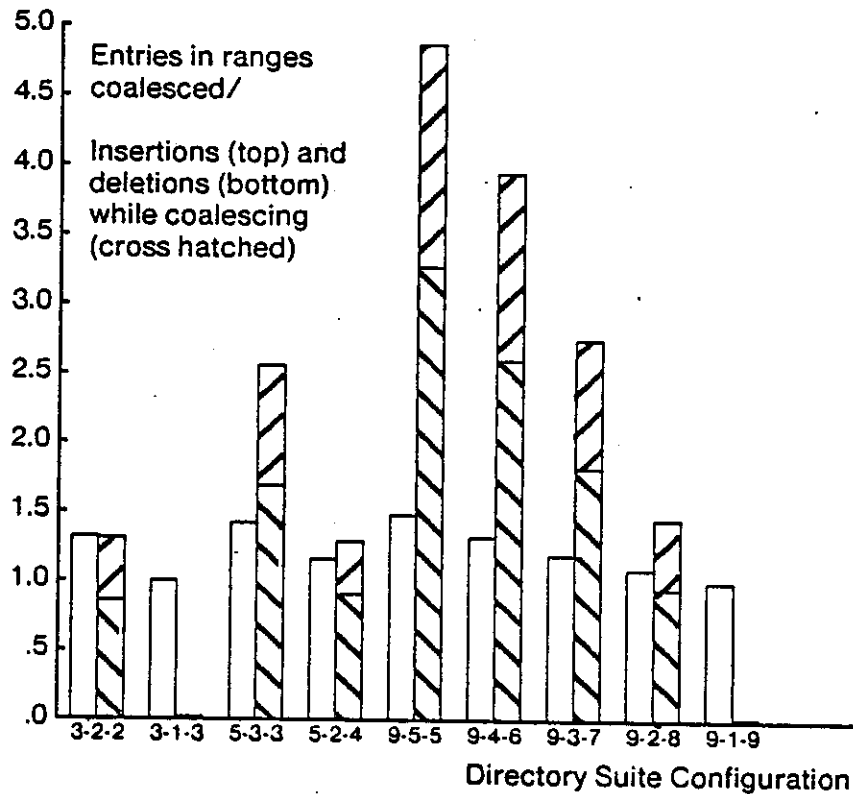

Figure 14: Simulation Results for Various Directory Suites

The second and third statistics, labeled "Insertions while coalescing," and "Deletions while coalescing," are the average numbers of insertions and extra deletions (per suite) performed during each DirSuiteDelete operation. The insertion statistic counts the number of real predecessors and real successors that must be inserted on representatives, and the deletion statistic counts the number of ghost entries that must be deleted. These statistics reflect the extra work done by DirSuiteDelete in addition to the work that would be done by the deletion operation of a unanimous update strategy having the number of replicas in a write quorum.

Figure 14 shows the average results of simulations using directory sizes of approximately one hundred entries with varying numbers of directory representatives and varying sizes of read and write quorums. The duration of each simulation was ten thousand operations, and the members of quorums and the keys to insert, update, or delete were selected randomly from a uniform distribution.

More detailed results for 3-2-2 directories with one hundred, one thousand, and ten thousand entries are shown in Figure 15. The duration of each of these simulations was one hundred thousand operations. The

maximums and standard deviations that are shown indicate the statistics do not vary significantly with directory size.[5]

100 Entries			1000 Entries			10000 Entries		
Entries in ranges coalesced								
Avg	Max	Std Dev	Avg	Max	Std Dev	Avg	Max	Std Dev
1.33	9	0.87	1.32	12	0.86	1.20	9	0.76
Deletions while coalescing								
Avg	Max	Std Dev	Avg	Max	Std Dev	Avg	Max	Std Dev
0.88	8	1.05	0.87	11	1.04	0.67	9	0.90
Insertions while coalescing								
Avg	Max	Std Dev	Avg	Max	Std Dev	Avg	Max	Std Dev
0.44	2	0.59	0.45	2	0.59	0.53	2	0.64

Figure 15: Detailed Simulation Results for three 3-2-2 Directory Suites

The measurements of the first statistic indicate that the real predecessor and real successor of a key to be deleted will be located quickly if the simulation assumptions hold. For instance, if each member of a read quorum sends the results of three successive **DirRepPredecessor** and **DirRepSuccessor** operations in a single message, the real predecessor and real successor will often be located using one remote procedure call to each member of the quorum. The results for the second and third statistics indicate that the weighted voting algorithm does little extra work during deletions, compared with a unanimous update strategy.

5 Discussion

Though the previous sections motivate and describe the basic replication algorithm, there are many performance issues worthy of mention. First, it is interesting to note that if the memberships of write quorums change infrequently, coalescing during deletions will not be costly. Thus, the statistics presented in the previous section are worse than could be achieved, because quorum members were selected randomly. In some ways, the algorithm behaves similarly to a moving primary update strategy [Alsberg 76] when write quorums change infrequently.

If transactions that operate on a directory exhibit locality of reference with respect to keys, quorums can be

[5] We believe that the statistics for the ten thousand entry directory do not reflect steady state behavior.

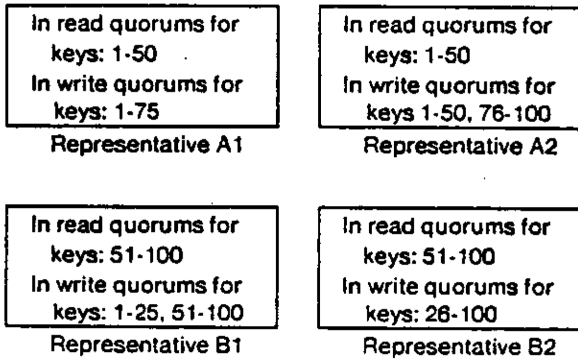

Figure 16: A 4-2-3 Directory Suite Partitioned for Locality

If transactions that operate on a directory exhibit locality of reference with respect to keys, quorums can be chosen that permit reads to be done locally and non-local writes to be distributed among all the non-local representatives.[6] 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 16, 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 similarly. 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.

With respect to the implementation of the replication algorithm, the sketches we have provided are pedagogically sound, but not the most efficient. Locking rules can be modified to permit greater concurrency without sacrificing serializability. Additionally, inter-representative message traffic can be reduced by combining certain remote procedure calls. We envision that directories could be represented as B-trees [Comer 79]. Version numbers for gaps could be stored in fields in their bounding entries. For some applications, version numbers containing 48 or more bits may be required to prevent version numbers from cycling.

The performance characterizations presented in this paper are based on simulations, however initial work on an analytical treatment indicates that we can obtain similar results from simple analytic models. Further simulations and practical experience are needed in order to quantify the additional concurrency permitted by this directory replication algorithm. We plan to implement this algorithm as well as Gifford's weighted voting

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

algorithm for files using a prototype transaction-based system we are constructing on a modified version of the Accent kernel [Rashid 81].

In summary, this paper has 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 directory by range and associating a version number with each range. Simulation results show the extra costs associated with maintaining the consistency of a directory replicated using our algorithm is low.

Acknowledgments

James Driscoll suggested improvements to our initial dynamic partitioning algorithm that resulted in the algorithm presented in this paper. These improvements simplified the algorithm and reduced the amount of overhead for insert and update operations. Joshua Bloch worked on the analytic model for this algorithm and made other helpful suggestions. Daniel Duchamp, Dave Gifford, Cynthia Hibbard, Robert Sansom, and Peter Schwarz have read and commented on drafts of this paper.

References

[Allchin 82] James E. Allchin, Martin S. McKendry.
Object-Based Synchronization and Recovery.
Technical Report GIT-CS-82/15, Georgia Institute of Technology, September, 1982.

[Allchin 83] 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.

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

[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.

[Lampson 79] Butler W. Lampson, Robert F. Sproull.
An Open Operating System for a Personal Computer.
In *Proc. Seventh Symp. on Operating System Principles*, pages 98-105. ACM, 1979.

[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 82] Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Carnegie-Mellon Report CMU-CS-82-128, Carnegie-Mellon University, Pittsburgh, PA, September, 1982.

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

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