Consistency Algorithms for Optimistic Replication*

R. G. Guy G. J. Popek† T. W. Page, Jr.

Department of Computer Science
University of California
Los Angeles, CA 90024-1596

Abstract

The need for replication of information storage, with system support for maintaining consistency among the various copies, has been widely recognized. However, earlier work guaranteed that copies would be consistent by assuring that conflicting updates could not occur. Such an approach decreases availability for update, often at precisely the point when availability is desired. Further, there is increasing data that conflicting updates are extremely rare in many situations of interest.

This paper presents a practical set of algorithms for maintaining the consistency of a replicated file system with an optimistic update policy. These algorithms permit a system which allows updates to an object so long as any copy is available; the algorithms then return the various copies to consistency at their first opportunity. These algorithms have been used to build the Ficus replicated file system.

1 Introduction

Replication of important information can serve to improve both the performance and fault tolerance of a distributed system. Locating replicas of data "close" to the location where it will likely be used can minimize the impact of network latency. The existence of multiple copies has the potential to improve data availability in the face of network or server failures by offering alternative sources for data resources.

Redundant storage, however, requires mechanism to maintain the consistency of replicas. Traditional distributed data management prevents inconsistent updates by any of a variety of means. These techniques (such as quorum consensus, weighted voting, etc.) have the unifying characteristic that they achieve consistency by requiring synchronous communications, and by restricting updates when such communications is unavailable. This yields the undesirable result that availability for update actually decreases as the number of replicas increases, the reverse of what is desired.

The optimistic approach, by contrast, detects and repairs, rather than prevents conflicts. This approach may be preferred in cases where conflicts are very rare, preventing conflicts is inordinately expensive (in terms of delay or availability), and the rare occurrence of a conflict can be tolerated or even repaired automatically. These conditions characterize a very large set of important environments, including much of today's use of distributed general purpose file systems [7, 4].

There are numerous environments for which replicated storage is quite valuable. In some of these, rapid communication among sites is not always possible. Interesting examples include conventional high availability systems using redundant hardware, significant numbers of workstation users collectively engaged in a large software development project, an office workgroup composed of several widely geographically separated workgroups, large numbers of laptops operating while disconnected, and military systems subject to communications silence. These examples share several common characteristics:

- low latency communications on demand cannot be guaranteed, either due to failures or policy decisions (such as not keeping a line in operation during high tariff periods);
- updates to data and meta-data (directories) are important to allow and occur from sites whose identity could not be specified in advance;
- concurrent updates of a given data item or directory entry are quite unlikely, and in those rare

*This work was sponsored by DARPA under contract number F29601-87-C-0072.

†This author is also affiliated with Locus Computing Corporation.

0-8186-3670-X/93 $03.00 © 1993 IEEE

cases where a conflict does occur, subsequent reconciliation is feasible. Strict transaction semantics are not required.

Optimistic replica management is certain to play an important role in in these environments. Optimism permits a ***one-copy availability*** concurrency control policy for both read and update: if any copy is physically accessible, read and update are permitted. With this policy, availability improves as the number of replicas increases. Optimistic replication should further guarantee a ***no lost updates*** semantics, so it is incumbent upon the system to detect conflicting updates and manage the mutual inconsistency until it is repaired.

Conventional replica management schemes implicitly or explicitly always have the property that a set of up-to-date "authority" replicas exists. No such authority is present in optimistic replication, short of a consensus reached by all replicas—a consensus not easily obtained when a complete communications graph between all replicas is unattainable.

For example, consider the problem of creating and deleting objects under optimistic replication. Object creation can be effected by causing a single replica to exist at one node; another node may then notice that an object exists for which it lacks a replica, and it will proceed to establish one of its own. But how is an object deleted? Simply deleting a replica will not do, since ***that is indistinguishable from object creation***. When comparing two replicas, one of which knows about an object and the other does not, it is impossible to determine whether the object is newly created in one replica and hence should propagate to the other, or whether the object has been deleted from one, and should therefore now be deleted from the other.

Attempting to determine whether an object is newly created or recently deleted is futile in the absence of additional information. This ***create/delete ambiguity*** (first noted by Fischer and Michael [3]) is resolved in conventional replication schemes by appealing to an authority; in optimistic replication, some other means must be used. In this paper, we provide solutions for this and other problems encountered in optimistic replication, including global inaccessibility and remove-update conflicts.

1.1 File systems

The algorithms presented in this paper provide for management and garbage collection of distributed, selectively replicated graph structures with associated resources. In practice, they are motivated by, and have been extensively applied to, the support of an optimistically replicated hierarchical filing system and accompanying name (directory) service for UNIX[4].

Consider the primary components of a typical UNIX file system. Files are hierarchically organized, with designated files (directories) containing the structural details (pathname components) which indicate a file's place in the hierarchy. The hierarchy is usually a restricted form of a directed acyclic graph. Two types of "objects" are present, files and file names. For replication management purposes, each object can be treated independently, including independent consideration of a file and its names. In the model below, a UNIX file corresponds to a multiply-named logical object, while the file's names are considered to be singly-named objects in their own right.

In reading the following sections, it may be helpful to keep in mind the role the algorithms play in the architecture of a real optimistic replicated system such as Ficus. Updates are initially applied to any one replica which is accessible. An update notification mechanism may then attempt to spread knowledge of the update to the other replicas in a best effort manner; that is, no attempt is made to ensure that the notification eventually gets to every replica. The system does not depend on update propagation for correctness. It is the role of the reconciliation algorithms which run on behalf of each replica to poll periodically other replicas looking for information which is needed to promote consistency and detect conflicts.

The next section specifies the problem to be solved in more formal terms and presents a family of algorithms to address it. We then comment further on the algorithms' utility, consider other related research, and conclude. The appendix provides correctness arguments which aid understanding of the algorithms.

2 Algorithms

The role of these algorithms is to restore mutual consistency to objects and their name space which are subject to unsynchronized creation, deletion and update, and to recover all resources related to an object for which the last name reference has been deleted.

2.1 Model

Our model provides clients with a persistent storage service for a collection of *objects*. A *logical object* is represented by a finite set of *physical object replicas*. Each object has a *replication factor* which defines the intended quantity and placement of replicas.[1] Clients access an object via a *logical name*, which is represented by a finite set of *physical name replicas*. Each name has a replication factor separate from the object it names and from other names for the object.

2.1.1 Object and Name Creation

The system creates a new logical object by establishing a single physical object replica and a single physical name replica. Additional name and object replicas for this object are established asynchronously. The first physical name replica to be established for a logical name is tagged with a unique value that distinguishes this particular usage of the name from all others; all physical name replicas for this logical name carry this same tag. An object is initially created with one (logical) name. Some objects may be restricted to only the original name; other objects allow additional names to be added or removed at will. Each object replica maintains a reference count, indicating the number of (local) name replicas referring to it.

2.1.2 Name Removal

In order to resolve the insert/delete ambiguity, name removal is effected by setting a flag to mark a name replica 'deleted', preventing its use in accessing an object. This indelible mark eventually propagates to the other replicas of the name. Until then, however, the object may be accessible via the remaining unmarked replicas of the name.

An object replica's reference count is decremented atomically with marking a name replica deleted. Name removal leaves an object inaccessible when no physical name replicas for the object remain (the reference count is zero). New names can only be added to accessible objects, so once the last name for an object is removed, it is permanently inaccessible. Resources devoted to an inaccessible object are subject to reclamation. That is, garbage collection of an object occurs as a side-effect of the removal of its last name, but a local zero name count does not necessarily imply global inaccessibility.

2.1.3 Communications Assumptions

We make only minimal and realistic assumptions about the available communications environment in order to assure successful operation of the algorithms in practice. All we require is that information be able to flow from any node to any other in the network over time if relayed through intervening nodes. More formally:

> nodes N_1 and N_m are *time flow connected* if there is a finite sequence of nodes $N_1, N_2, \ldots, N_m$ such that for $1 \leq i < m$, a message can successfully be sent from N_i to N_{i+1} at time t_i, and $t_i < t_{i+1}$.

We require that every pair of nodes is time flow connected starting at any time. This property does not require, for example, that any pair of nodes be operational simultaneously, but it does mean that no relevant node can be down indefinitely.

We also assume that nodes are truthful: Byzantine behavior does not occur. Finally, history only moves forward: a node must never "roll back" from a reported state, so stable storage of any reported state must precede that report. In practice, this affects how replicated data may be restored from backups.

Several issues determine (in part) the complexity of replica management: static versus dynamic naming, object mutability, equivalence of name and object replication factors, and static versus dynamic replication factor. The algorithms presented in the next several subsections vary in their ability to handle these issues, ranging from the simplest combination (fixed name, immutable object with equivalent static replication factors for both name and object) to the most difficult (dynamically named, mutable objects, with non-equivalent name and object replication factors). Further, to aid clarity of discussion, we assume that no more than one replica is stored by any given node. The algorithms generalize directly to multiple replicas per node.

2.2 Basic two-phase algorithm

The basic two-phase algorithm is appropriate for the simplest kind of replicated object: static single name, immutable object, and equivalent name and object replication factors. Subsequent more difficult types of management tasks adapt this algorithm to accomplish their goals.

By using the technique of marking a name deleted in order to resolve the insert/delete ambiguity, we have simply transformed the problem into an issue of garbage collection. That is, when, if ever, is it safe to completely remove all resources devoted to a name which is marked deleted? The answer is that a node, A, cannot remove all record of a name replica until it

[1] We use the term replica to include all of the resources at a node which are devoted to the (logical) object. Typically, this includes a copy of the object's "client data," as well as any replication or other bookkeeping meta-data associated with the object. Resources consumed by meta-data must be reclaimed as well as resources used by client data.

determines, not only that all replicas of the name are marked deleted, but that every replica knows that A knows that all replicas have marked the name deleted.

The basic reclamation algorithm proceeds in two phases at each node. The first phase begins executing at a node when the node learns the object is to be reclaimed, that is, when its (single) name replica is marked deleted. This mark is then also placed on the object replica. Actual physical reclamation of the object replica (and name replica) will not occur until after the node completes its second phase of the algorithm. Figure 1 lists the basic two-phase algorithm in pseudo-code.

Each node concurrently executes the algorithm, and shares its progress with other nodes. Sharing improves the algorithm's efficiency, but more importantly, it enables the algorithm to cope with pathological communications failures.

2.2.1 Phase one

The first phase proceeds by composing a list of nodes that have their object replica marked deleted. In effect, each node is engaged in the same activity: collecting information about the deletion status of each object replica. A node completes its first phase when every replica is listed as marked deleted[2].

Indirect communication is an integral part of the algorithm. When a node inquires about a peer's status, the list of replicas marked deleted maintained by that node is shared with the inquirer, who in turn incorporates the information into its own list. Thus a node, A, may find out indirectly from B that C knows an object is deleted, without A and C ever communicating directly.

A node that has completed phase one knows that all replicas have marked their name and object replicas deleted, and thus have themselves begun executing phase one of the algorithm. However, a node at this stage makes no assumptions about the knowledge other nodes have of *it*. It is possible that no other node is aware that the node in question has marked its replica deleted, as the flow of information is not guaranteed to be a two-way exchange at any step. This necessitates a second phase to guarantee termination.

2.2.2 Phase two

Immediately upon completing phase one, a node begins executing phase two. In this phase, a node com-

[2] In practice, the list of nodes is a bit vector, with a component for each replica. Adding a replica to the list is accomplished by turning on its bit in the vector. Phase one is complete when all of the bits are turned on.

```
/* variables and data structures:

   Let set R := replication factor,
     r, s, self  elements drawn from R,
     P1, P2, R  binary vectors of size |R|
*/
begin:
  while (my name replica not marked deleted)
    { donothing; }

  mark my object replica deleted;

  P1[r] := 0, for all replicas r;
  P2[r] := 0, for all replicas r;

phase1:
  P1[self] := 1;
  while (P1[r] == 0 for any r) {
    R[r] := 0, for all r;
    choose some r and ask for its P1 vector;
    if r responds {
      R[] := r's response;
      foreach (R[s] == 1)
        { P1[s] := 1; }
    }
  }

phase2:
  P2[self] := 1;
  while (P2[r] == 0 for any r) {
    R[r] := 0, for all r;
    choose some r to query;
    ask r for its P2 vector;
    if r responds {
      R[] := r's response;
      foreach (R[s] == 1) { P2[s] := 1; }
    }
  }
postlude:
  reclaim object replica resources;
  reclaim name replica resources;
```

Figure 1: Basic two-phase algorithm

piles a list of nodes that it learns have finished phase one. The first node placed on this second list is, of course, itself. As with the first phase, phase two at this node is complete when all nodes are listed. The same style of list sharing also occurs in phase two.

Nodes placed on a phase two list are those that know that all replicas are marked deleted. A node with a complete phase two list therefore knows that all nodes know all replicas are marked deleted. This global state is vital to providing "once reclaimed, never re-established" behavior: it allows a node that has finished phase two to reclaim all local resources devoted to the replicated object and to forget about it entirely, secure in the knowledge that the replica will never be re-established in response to a query from another node about the object.[3]

A node in phase two might query a node which has already reclaimed its resources and forgotten about the object. The query response will indicate that no such object is known, which the inquirer will (correctly) interpret to mean that the queried node has completed phase two. The state of having finished phase two can be distinguished from never having even known about the object in the first place because the presence of the complete phase one list at the inquirer guarantees that the inquiree once knew about the object and its deletion. The inquiring node uses this inferred status to complete its own second phase immediately, and proceed with reclaiming its object and name replicas' physical resources.

2.3 Intermediate algorithm

The basic algorithm in the previous section applies to fixed name, immutable objects with identical static name and object replication factors. In this section we relax the first two constraints to allow dynamic naming and object updates, while continuing to require name and object replication factors to be both identical and static.

Dynamic naming and object mutability each complicate the reclamation problem, and the combination of the two is especially difficult. Dynamic naming introduces a global stable state detection problem, while object mutability requires special mechanisms to prevent inadvertent data loss.

[3] "Once reclaimed, never re-established" behavior is important for both practical and theoretical reasons: resource allocation and deallocation is costly and should be done only when necessary; and, removing the possibility of re-allocation greatly simplifies algorithm termination arguments.

2.3.1 Dynamic naming

A necessary, but not sufficient, condition for object reclamation is that the object have no names. In our model, 'no names' means that every name replica referring to an object replica has been marked deleted. In the basic two-phase algorithm, object reclamation inevitably follows name removal; the two phases ensure that all replicas will be reclaimed, exactly once. In the dynamic naming case, it is much harder to determine whether or not reclamation is to occur: names may be added or removed at any node at any time, since optimism allows unsynchronized concurrent updates despite non-communicating partitions. When a name is added at one node concurrently with a name removal at another node, a transient situation may arise in which a node has no names for an object for a time, until the new name propagates to that node. During this time, reclamation of the 'nameless' object replica would be premature, even though, locally, it has a zero-valued reference count.

Premature reclamation must be avoided because of the potential for violating "no lost updates" semantics. Concurrent update, name creation, and name removal together with the non-atomicity of name and object propagation leave open the possibility that the only object replica reflecting an update could temporarily have a zero-valued reference count. Such a replica must not be prematurely reclaimed.

Although a single replica's zero-valued reference count may be a transient condition, when all replicas have zero references a global stable state [2] exists. The task, then, is to detect that all object replicas simultaneously have zero-valued reference counts in an environment where simultaneous or pseudo-simultaneous queries of all nodes are not feasible.

We provide an adaptation of our basic two-phase algorithm which exploits the rules governing name additions to achieve a relatively inexpensive, fully distributed mechanism for determining the global zero-valued reference count stable state. The adaptation requires that each object replica maintain a monotonic counter in parallel with the reference counter, and that the algorithm compile and distribute a vector of these new counters. The new counter is incremented atomically with the reference counter, but it is never decremented. It functions as a 'total name counter' to reflect the number of name replicas at this node which have referred to the object replica. The total name counter from each replica is used to determine that a zero-valued reference counter has been quiescent between interrogations.

2.3.2 Algorithm for dynamic naming

This intermediate two-phase algorithm is triggered at a node when the object replica's reference count becomes zero. In the first phase, two parallel vectors are maintained. One vector indicates which replicas have reported a zero-valued reference count, and the other contains the total name counter value reported by those replicas. A node has completed its first phase when all replicas are listed with total name count values recorded in the parallel vectors. This also implies that each replica reported a zero-valued reference counter. These parallel vectors are shared in gossip fashion as before.

The second phase proceeds similarly, with two parallel vectors of total name count values and report indicators. In this phase, the total name count values recorded reflect a replica's total name count value at some point after the queried replica has finished phase one. As a node is collecting values in the second phase, it compares the newly reported values with those recorded in its phase one vector. If any discrepancy is discovered (i.e., the corresponding values are not identical), the algorithm aborts, re-initializes its vectors, and restarts phase one. This abort occurs when the transient behavior described above occurs. A node finishes its second phase when all replicas have reported values to it, and the values are identical to those collected in the first phase. At this point, all object replicas are guaranteed to be permanently inaccessible.

2.3.3 Mutability

As presented, the intermediate algorithm will determine that an object is globally inaccessible. A further condition is necessary to allow physical reclamation to proceed: we must insure that the object for which the last name was removed was not concurrently updated (a *remove/update conflict*). Reclaiming such an object risks violating the 'no lost updates' guarantee. Note, we are not concerned here with the kind of 'inadvertent loss' that results when a client mistakenly removes a name, but only with loss which results as a consequence of optimism.

We assume that there may be a connection between the state of an object and the decision to remove a name for it, so we invest the name removal operation with the additional semantics that a client wishes object reclamation (when no names exist) if no other object replicas are newer than (or in write-write conflict with) the object replica which is initially affected by the name removal.

To accommodate the additional semantics, the reclamation algorithm must determine, without synchronized clocks, which of the object versions represented by the replicas is the latest, and which is the latest version to provide a context for name removal. (Optimism also introduces the possibility that no 'latest' version exists, such as when unsynchronized concurrent updates occur to distinct replicas, generating an update/update conflict.) After identifying the latest object version and removal context version, it is trivial to detect a remove/update conflict.

Version identification and context recollection can be readily accomplished with *version vectors*, which provide a multi-dimensional version numbering scheme for replicas [8][4]. We augment the object replica model with two data items: a 'current' VV, and a 'removal context' VV. The current VV always identifies the set of updates that have been applied to the object replica. The removal context vector is replaced by a copy of the current VV when a name removal operation is issued with this object replica providing context. Each replica's removal context vector is checked to see that no remove/update conflict exists.

2.3.4 Remove/update conflict algorithm

Recall, a remove-update conflict exists if the last name for an object is removed while that object is concurrently updated. The last name is gone, but the object should not be garbage collected lest "no lost updates" be violated.

It is easy to modify the intermediate two-phase algorithm to collect and compare the various vectors and determine if a remove/update conflict exists: each instance of the algorithm can collect (and share) sets of vectors, and perform the appropriate comparisons when the sets are complete. This approach, however, imposes quadratic storage and message size complexity upon each instance of the algorithm.[5]

Linear storage complexity can be achieved by exploiting the (partial) ordering of version vector values. Instead of retaining each replica's version vector

[4]A version vector (VV) is a tuple of integer counters with a component for each replica of an object. When an object is updated, the component of the vector corresponding to the replica to which the update was first applied is incremented. When an object replica propagates, its VV propagates with it. Thus a VV is a summary of all updates that have been applied to a given replica. Two VVs for different replicas of an object may be compared. If one component-wise dominates the other, then it reflects every update the other one does, plus some; it is strictly newer. If the vectors are pair-wise equal, the the replicas are identical. If they are not equal and neither is dominant, then each reflects at least one update that the other does not and an update/update conflict is indicated

[5]Each VV is of length n, of which n must be collected in each set ($n = |\text{replicas}|$).

values, an algorithm instance can remember only the greatest vector value encountered, along with a list (vector) indicating which replica's vectors have been consulted, and whether the vectors conflict with the greatest values seen to this point in the algorithm's execution.

The linear optimization is not free, however. A two-phase consultation scheme is required to collect the vectors and correctly assert that a particular vector value is greatest, or that no value is greatest due to conflicting versions. As it happens, these two phases can be executed in parallel with the two-phase algorithm that determines global inaccessibility, so the cost is effectively eliminated.

Once global inaccessibility and remove/update conflict status are determined, a decision can be made whether to reclaim an object replica's resources. If a remove/update conflict is discovered, reclamation will not occur; proper action at this point is application dependent. (An example is described in a later section.) Figures 2 and 3 show the intermediate algorithm.

2.4 Advanced two-phase algorithm

The previous algorithms each assume that object and name replication factors are fixed at creation time, and are identical. In practice, these constraints are not attractive. Changing circumstances of network behavior or object usage may necessitate adding, deleting, or moving replicas, which cannot be usefully predicted when an object is created. It should also be possible to change an object's replication factor without directly affecting object names. Note that an object (or name) replication factor is itself a replicated data structure which is used to manage other replicated data structures. The version vector technique used to manage updates to replicated data can not easily be applied to managing updates to version vectors themselves.

Our scheme supports an approximation to an ideal flexible replication factor mechanism: a replication factor can grow to be very large (2^{32} replicas), with masks used to 'shrink' a replication factor. One mask is used to indicate that particular replicas should be (irrevocably) ignored during algorithm execution. The second mask permits an object replica to avoid the expense of storing the object itself any further, but the 'skeletal' replica must continue to participate in algorithm execution. In short, a replication factor monotonically increases in physical size, with adjustments available to reduce the actual number of physical copies of a client's data which are maintained.

Increasing a replication factor is straightforward. Any replica's replication factor can be augmented simply by adding a (globally unique) replica identifier to its list of replicas. A replica can form a new replication factor while executing one of the two-phase algorithms by taking the union of its replication factor and that reported by another replica.

A replication factor's 'ignore' mask provides a way for a replica to be forever ignored. This is especially useful when recovery of a destroyed replica is impossible or too expensive. Indicating that a replica is to be ignored is an irrevocable action. Like an increase in replication factor, a new ignore mask is formed by taking the union of the local mask and one reported by another replica. The 'skeletal' mask indicates which object replicas don't actually store any client data. This mask is maintained in an optimistic fashion, but without conflict detection: mask updates cause a new timestamp to be generated for the mask; the mask with the latest timestamp is deemed to be correct.

2.4.1 Algorithm

Very few changes need to be made to the intermediate two-phase algorithm to support dynamic name and object replication factors. Each replication factor must support two additional parallel data structures (the masks), and the algorithm must check reported replication factors for changes. Care must be exercised, when increasing a replication factor, not to violate the semantics of an in-progress reclamation algorithm. Our two-phase algorithms have two critical points: when a node finishes phase one, it believes that all replicas have been consulted; and when a node finishes phase two, it believes that all replicas have finished phase one. While a node is currently in phase one, its replication factor can be augmented safely because every other node must consult it at least once more, during phase two. When this node is consulted, other nodes will learn about the additional replica(s). But a replication factor must not be augmented to create a new replica when the 'source' replica's node is in phase two. For brevity, we do not show these minor algorithm modifications in a separate figure.

3 Applications and observations

Which two-phase algorithms are appropriate for managing a replicated network file system? UNIX files are mutable, dynamically named objects, so at least the intermediate algorithm should be used for them. File names (directory entries) are simple objects which can be managed with the basic two-phase algorithm. However, the additional cost of implementing and using the advanced algorithm (with flexible replication

```
/*
 * = major changes to basic algorithm
  new vectors:
 NCR  total name count response
 NC1  total name count, phase1
 NC2  total name count, phase2
 NV   total name count validation
 VV   replica's version vector
 VVR  version vector response
 SVV  saved version vector response
 RC   removal context vector
 RCR  removal context response
  new scalars:
 C    reference count response
 RU   remove/update conflict flag
*/

begin:
  while (my ref-counter non-zero)
    { donothing; }
  reset all elements of vectors
    P1, P2, NCR, NC1, NC2, NV;
  RU := 0;

phase1:
  P1[self] := 1;
  while (P1[r] == 0 for any r) {
    NCR[r] := 0, for all r;
    choose some r to query;
*     ask r for its C, NC1, P1, VV, RC;
*     if r responds with C==0 {
*     NCR[] := r's NC1;
*     NV[] := r's P1;
*     foreach (NV[s] == 1) {
*       NC1[s] := NCR[s];
    P1[s] := 1;
    }
*     VVR := r's VV;
*     RCR := r's RC;
*     if (VVR >= VV) { SVV := VVR; }
*     if (RCR >= RC) { RC := RCR; }
  }
  }
```

Figure 2: Intermediate algorithm, phase one.

```
phase2:
  P2[self] := 1;
  while (P2[r] == 0 for any r) {
    NCR[r] := 0, for all r;
    choose some r to query;
*   ask r for its C, NC2, P2, VV, RC;
*   if r responds with C==0 {
*   NCR[] := r's NC2;
*   NV[] := r's P2;
*   foreach (NV[s] == 1) {
*     NC2[s] := NCR[s];
*     P2[s] := 1;
*     if (NC1[s] != NC2[s])
*       goto begin;
    }
*   VVR := r's VV;
*   RCR := r's RC;
*   if ((VVR conflicts SVV) or
*        (RCR conflicts RC)   )
*     { RU := 1}
*   if (VVR >= VV)
*     { SVV := VVR; }
*   if (RCR >= RC)
*     { RC := RCR; }
      } else if (C > 0)
    { goto begin; }
  }

postlude:
  if (RU == 0) {
    reclaim object replica resources;
    reclaim name replica resources;
  } else {put object into orphanage}
```

Figure 3: Intermediate algorithm, phase two.

factors) is negligible. The Ficus optimistic replicated file system, described in [4, 7], incorporates the advanced algorithm to manage its files.

The advanced algorithm is also used to support the name service that connects subtrees together to form a large connected hierarchical filing environment. This name service plays a role similar to NIS (Yellow Pages) in NFS, or volume support in AFS. The implementations of these two applications (file hierarchy and volume hierarchy) are common, so multiple facilities were not required[7].

3.1 Directed acyclic graphs

The UNIX filing environment is a simple directed acyclic graph (dag) structure. These algorithms may be applied to an arbitrary graph structure as well, so long as there are no disconnected self-referential subgraphs. Additional mechanism is needed to handle that case.

In fact, modest mechanism beyond that discussed in this paper is required even to handle dags. This is because the discussion was cast in terms of a single logical object. The additional facilities are simple, and discussed in [4].

3.2 Communications cost

Performance of these algorithms is, of course, important. A suitable measure is the number of messages that must be exchanged in order to cause a set of n nodes with replicas to reach agreement. One would expect that the worst case could be expensive, since the underlying minimum communications assumptions do not allow a stylized pattern of interaction always to be employed. The worst case indeed requires $O(n^2)$ messages, as most nodes talk to most of the other nodes to complete each phase.[6] However, in practice the situation is far better, since we can communicate in a stylized manner most of the time. As a simple example, if the nodes order their communications in a ring, then a total of $3n - 1$ messages are used.[7]

[6] In each phase, in the worst case, a first node pulls from the $n - 1$ other nodes to become knowledgeable. A second node then pulls from the remaining $n - 2$ unknowledgeable nodes, and then the first, knowledgeable one. The third node pulls from the remaining $n - 3$ unknowledgeable nodes, and then one of the knowledgeable ones. Thus each phase requires $(n-1) + (n-1) + (n-2) + (n-3) + \ldots + 1 = \frac{(n+1)n}{2} - 1$ pulls, and there are two phases. Thus, $n^2 + n - 2$ pull messages are required.

[7] Assume that a single message is active in the ring at any time. This ever-changing message flows around the ring three times. Phase one of the algorithm begins for all nodes in the first round trip. Phase one completes and phase two begins for all nodes during the second round trip. Phase two completes for all nodes during the third round trip.

4 Related work

Our work is related to several areas of research: the "gossip" problem, which has received substantial formal treatment; optimistic file systems, including LOCUSand Coda; optimistic "dictionaries" (directories) in file systems; and, distributed garbage collection.

In the gossip problem, each node in a graph must communicate a unique item to every other node in the graph. A variety of papers have appeared in the twenty years of its study [6], yielding complexity results under varying communications assumptions. Heddaya, Hsu, and Weihl [5] used a two-phase gossip protocol to manage distributed event histories of updates to object replicas. A timestamp vector is used to determine when history elements may be safely discarded. Their solution does not address the problem of completely forgetting that a history exists, but only forgetting items in the history.

LOCUS [9], an intellectual ancestor of the Ficus file system, incorporated more limited replica management algorithms, from which the algorithms presented here are descended. The version vector scheme utilized here is borrowed directly from LOCUS[8].

The Coda project [10] has similar goals to our own Ficus work and bases its replica management on the LOCUS version vector mechanism and utilizes, to some extent, an earlier draft of this work.

Fischer and Michael [3] proposed recasting the replicated directory maintenance problem as a replicated "dictionary" problem, with slightly (but significantly) different semantics. A timestamp vector was used to infer from a comparison of two dictionary replicas which entries had been inserted and which had been deleted. Allchin [1] and Wuu and Bernstein [12] expanded upon Fischer and Michael's approach to use two-dimensional timestamp matrices to reduce the number of messages exchanged, with small variations in semantics. None of these works addressed the general problem of reclaiming resources of named replicated objects; they were concerned with "dictionary entries" as isolated entities.

Wiseman's survey [11] of distributed garbage collection methods includes several techniques based on reference counting, but none are designed for use on replicated objects, and none are directly applicable to imperfectly connected networks.

5 Conclusions

We believe that optimistic replication underlies a number of important distributed systems problems, and so have labored to develop relatively general solutions. The advanced algorithm described here has been used in the Ficus replicated file system with excellent results.

It is worth noting that a successful solution to the problems addressed here is more difficult than it may at first appear. The numerous pathological cases that can occur in practice were sobering and instructive.

There are a number of useful directions for future work. Further reductions in communications costs may be achievable. We are working on adaptive methods of determining the topology by which sites reconcile with each other in order to reduce the total overhead required to complete the two phases of garbage collection. Incorporation of support for general graph structures is an obvious extension. Further, many data structures have characteristics very similar to file system directories; sequences of records with update transactions consisting only of insertion and delete. Examples include relational databases, and most replicated tables. Very similar algorithms should be capable of optimistic management of this class of data. The ability to freely intermix optimistically replicated components with others which are kept strictly consistent could be important in practice. Nevertheless, the algorithms as presented appear quite useful in their current form.

We conclude from the experience of designing these algorithms and using them in Ficus, that optimism is not only feasible, but likely to play a substantial roll in the replicated network file systems of the future.

Appendix

A Correctness discussion

The following informal argument asserts that the algorithms are *correct*. The basic two-phase reclamation algorithm is correct if and only if these conditions are satisfied:

- object reclamation occurs iff the object is globally inaccessible
- for each replica of an inaccessible object, reclamation occurs exactly once, in finite time
- all algorithm executions terminate in finite time
- all algorithm executions are free from deadlock

We first show that reclamation occurs *if* an object is inaccessible, followed by the *only if* direction. We then show that reclamation occurs exactly once in finite time by proving that it occurs at least once, and at most once.

A.1 Reclamation *if* inaccessible

The *time flow connected* assumption ensures that it is possible for each node to learn of status changes at every other node. Since each node periodically uses the propagation protocol to incorporate other replicas' status changes into its own replica, and since all replicas are guaranteed to be available at some time, each node will, in fact, learn in finite time of the status of every other replica. Therefore, every logical name deletion will eventually be reflected at every node, as each name replica will be indelibly marked deleted.

Following the deletion of every name for an object, in finite time all name replicas will be marked deleted. Each object replica will, in turn, have a zero-valued reference count, and be inaccessible.

The first phase of the algorithm simply collects the information that, when consulted, each replica was inaccessible. The second phase similarly collects information from each node. By the previous argument, each node is guaranteed to learn the desired information. At the conclusion of executing its second phase, a node reclaims its resources. Since each node is guaranteed to finish its phases if the object is inaccessible, each node will reclaim the resources consumed by the object.

A.2 Reclamation *only if* inaccessible

We argue by contradiction. Suppose reclamation of an object replica occurred without the object being inaccessible. Therefore, some object replica must have a non-zero reference count at the end of a node's second phase.

But, the algorithm's first phase demonstrated that each replica had a zero-valued reference count (though not necessarily simultaneously), and the second phase ensured that each replica's reference count had not changed between the first and second reference count queries. Since the second set of queries strictly followed the first set, a point in time must exist at which all replicas were simultaneously inaccessible. Global inaccessibility is a global stable state, by the restrictions placed on additional name creation. Therefore, a non-zero object replica reference cannot exist, which contradicts the hypothesis.

A.3 Reclamation *exactly once*

If the object is inaccessible, each replica will be reclaimed at the end of its node's execution of the algorithm, as per the above arguments. Therefore, each replica is reclaimed at least once.

Multiple reclamation requires multiple establishment of a replica. Replica establishment occurs when a node without a replica receives a message that indicates that the receiver is intended to have a replica and there is no indication in the message of the replica's prior existence. Therefore, to re-establish a replica, a node which has already reclaimed its replica must receive a message about the object which does not indicate that the replica is known to have existed.

It suffices to hypothesize that such a message is received, and then prove that such a message cannot arrive. We do so by classifying all messages and showing that none of the types which could cause replica establishment will be received after reclamation.

Every message about an object replica implicitly indicates a "phase" of algorithm execution. In addition to phase one and phase two messages, nodes routinely send status query and response messages to learn of object updates when the algorithm is not executing. For convenience, consider these routine messages to be "phase zero" messages.

When a node without a replica receives a message, its decision whether to create a replica is based on the phase of the sender:

zero A phase zero message contains no indication whether the receiving node ever had a replica. Therefore, a replica must be established.

one A phase one message may or may not indicate that the replica has ever existed. If it does not indicate that the replica existed, a replica must be established. If it indicates that a replica once existed, an anomalous condition has been encountered (see discussion below).

two A prerequisite for entering phase two is that all replicas have been consulted, which implies that all replicas exist. Therefore, the replica has previously existed, been reclaimed, and must not be re-established.

A node which has already reclaimed its replica normally expects to receive only phase two messages, because a condition of phase two completion is to determine that all other nodes have finished phase one. Since phase two messages cannot cause a replica to be re-established, only the receipt of phase zero or phase one messages after reclamation might cause a replica to be established again.

Phase zero or phase one messages received by a node which has completed phase two and reclaimed its replica could only have been sent *before* the sender began phase two. Such messages have been delayed in transit long enough for the sender to finish phase one and the receiver to finish phase two.

The algorithm is resilient to delayed messages which are received within the next phase: phase one messages received by a node in the midst of phase two are quite normal, as are phase zero messages received during phase one. It is only when message delay exceeds one phase that replica re-establishment might occur.

We assume that message delays do not exceed the time required for one complete phase. If this bound is invalid, algorithm execution can be artificially slowed to increase the length of a phase until a valid bound is achieved. It is, therefore, feasible to prevent phase zero or phase one messages from arriving after reclamation occurs.

The hypothesized message received after a replica has been reclaimed must be from one of the three phases, but since delayed phase zero and phase one messages can be prevented and phase two messages do not cause replica establishment, no message which could cause replica establishment will be received. This contradicts the hypothesis that such a message might be received, and so replica re-establishment (and subsequent reclamation) after an initial reclamation is not possible.

A.4 Termination

We show that the algorithm terminates by defining a partial order on the possible states of a node during the algorithm's execution, and showing that all state transitions are monotonic with respect to this order. (We showed above that sufficient transitions will occur, based on the finite time information flow assumption.)

A node's algorithm execution status is primarily determined by the list of replicas consulted, compiled in each phase. The set of valid list values comprises all subsets of the (finite) set of replicas indicated in the object's replication factor. A partial order based on cardinality can then be defined over these subsets.

A state transition (list change) is defined in the algorithm to be a set union operation, which is monotonic over the partial order. A partial order is acyclic, so all algorithm state transitions are acyclic. Progress towards termination is guaranteed, unless deadlock occurs.

The intermediate algorithm occasionally aborts and restarts. The only circumstance in which abort occurs (a mismatch of total name count vectors) is bounded in occurrence by the product of the number of names and the cardinality of the object's replication factor. Since the number of aborts at a node is bounded, some algorithm execution will not abort, and so the above termination argument holds.

A.5 Deadlock-free

We show that the protocol is free from deadlock by developing a *waits for* graph model and proving that it is acyclic for all algorithm executions.

Recall that the propagation protocol underlying state transitions is non-blocking, so a node is never blocked waiting for a particular response from another node. It therefore suffices to consider the algorithm's behavior at the higher level of phase transitions, where 'waiting' does occur.

Define a total order over the states 'accessible', 'phase one', 'phase two', and 'reclaimed' such that accessible < phase one < phase two < reclaimed. A node transitions from accessible to phase one when its replica becomes inaccessible, and from phase one to phase two when it learns that all nodes have transitioned to phase one. It transitions from phase two to reclaimed upon learning that all nodes have transitioned to phase two.

With the exception of the initial transition from accessible to phase one, a node waits for all other nodes to reach the same state as itself, before transitioning to a later (fully ordered) state. Therefore, a node only waits for "lesser" nodes; since "lesser" is acyclic, no cycles can occur in the waits-for graph and so the protocol is deadlock-free.

References

[1] James E. Allchin. A suite of robust algorithms for maintaining replicated data using weak consistency conditions. In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, October 1983.

[2] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[3] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

[4] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. Ph.D. dissertation, University of California, Los Angeles, June 1991. Also available as UCLA technical report CSD-910018.

[5] Abdelsalam Heddaya, Meichun Hsu, and William Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49:35–57, October 1989.

[6] Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Arthur L. Liestman. A survey of gossiping and broadcasting in communication networks. *NETWORKS*, 18:319–349, 1988.

[7] Thomas W. Page, Jr., Richard G. Guy, Gerald J. Popek, and John S. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report CSD-910005, University of California, Los Angeles, March 1991.

[8] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[9] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture*. The MIT Press, 1985.

[10] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[11] Simon R. Wiseman. *Garbage Collection in Distributed Systems*. Ph.D. dissertation, University of Newcastle Upon Tyne, November 1988.

[12] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, August 1984.