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

RECONCILING PARTIALLY REPLICATED NAME SPACES

Richard G. Guy
Gerald J. Popek

April 1990
CSD-900010

Reconciling Partially Replicated Name Spaces

Richard G. Guy and Gerald J. Popek

Department of Computer Science
University of California Los Angeles

March 30, 1990

Abstract

This paper considers the problem of *reconciling* (keeping consistent) partially replicated name spaces used by distributed file systems. The solution presented here supports independent (unsynchronized) update of non-communicating filesystem replicas by adopting an optimistic, non-serializable, *ad hoc* correctness definition. The algorithms have been implemented in the FICUS replicated file system. (companion paper)

The ability to reconcile replicated name spaces is fundamental to providing a highly available distributed file system service. Our novel policy, *one-copy availability*, provides equivalent read and update availability: if any replica is physically available, it is available for any purpose; no artificial restrictions are imposed. The definition of correctness we adopt is sufficiently strict, though not serializable, to make the reconciliation task decidedly non-trivial.

The algorithms here originated to support a replicated filing service, and we therefore draw terminology and examples from that particular domain. The general results, however, apply to arbitrary replicated DAGs with similar (or weaker) consistency constraints and requirements.

1 Introduction

As networked computer systems increasingly dominate the landscape and cooperating users work on separate machines, the importance of assuring access to shared information in an efficient manner increases. Automatic and selective replication of files on storage near expected users is a potentially attractive aid to both increase availability in the face of failures and improve performance by increasing the ratio of fast local to slower remote access.

Typically, a replicated file system strives to deliver to its clients the image of a single logical copy of each file. So long as all copies are stored at communicating sites, this goal is easily achieved through a variety of means. When a new version of a file is created, it is transmitted to all storage sites, and some form of a synchronization protocol is used to assure that all requests for the file see the most recent copy.

Management of replication is complicated by failures, and in many common installations occasional loss of immediate access to copies is a fact of life. Such failures can occur in numerous ways. Locally, a storage site may fail or a shared LAN may become inoperative. When a user is geographically distant from the data, the problem becomes much more difficult. A wide area network is subject to multiple sources of disruption which often occur in third party facilities not under control of the user. Relay services at intermediate points may have priority low enough (or congestion high enough) to induce delays that are equivalent to failure. Communications tariffs that vary widely with time of day may effectively require that transmission be delayed until a lower cost period, creating the effect of a communications outage. Institutional boundaries often dictate that communications be batched, disallowing data exchange in the intervals between batches. We call such communications outages *partitions*, in the sense that when such a failure occurs only subsets of sites may be able to communicate with one another.

1.1 Partitioned update

Problems arise in a replicated file system when any of these failure modes prevent storage sites for a data object from communicating. In particular, if updates to an object are permitted during partition, then conflicting versions of the object will exist and confusion results when communication is reestablished. On the other hand, if updates are blocked, then either the user cannot perform his work, or he must make private copies of information, perform the update in a customized way, and remember to merge the new version back into its proper place later when update is re-enabled. None of these situations is especially attractive.

A variety of systems architectures [Stone79, Bloch84, Noe86] have used very simple replication disciplines among their copies to maintain mutual consistency and avoid the conflicts that can occur when, by chance, two users wish to update the same logical data item in disconnected partitions. In contemporary systems, significant restrictions are usually imposed. For example, one copy may be designated as primary: all updates must be done to it, or it must at least be present in the updating partition [Alsb76]. Updates under other conditions are disallowed. Alternatively, each copy has a weight, and the majority of weights must be present for an update to be allowed [Thom78, Giff79, Herl86]. In any case, update availability can be seriously reduced or entirely eliminated under such conditions. In all of these cases, one generally gets *reduced* update availability as the number of copies is increased.

Consider the situation when the primary site fails, or when a LAN outage prevents many sites from communicating. Then, even when a copy of the replicated data is local to the user's workstation (where the update originates), the update cannot proceed. That situation is especially unfortunate, for example, when the object being updated is merely a directory and the changes are simply the insertions of two different file names. One could easily anticipate that a more resilient name service would be important, whether contemplating a nationwide file system or a

small, private work group, one portion of which was located in Los Angeles and the other in London.

When selecting an appropriate solution to management of replicated files, it is useful to have a view of typical usage patterns. It is widely recognized that in a general filing environment a statistically large proportion of data in use among cooperating work group members remains read-mostly, even though that data is subject to update and those updates may be quite important to accomplish. Often the updates are originated by one or a few parties, rather than coming from many sources and needing active database-style synchronization. Further, many of the "conflicts" that occur in such a replicated filing environment may not really be conflicting updates at all, such as when a shared directory is independently updated by users creating different files.

Interestingly enough, for certain classes of updates it is possible to reconcile automatically the differing versions of the data object, generating a result which satisfactorily captures the changes that took place during partitioned operation. An example is a mailbox file of electronic messages, into which messages are delivered by the mail system and removed by the user. It is easy to specify the desired result of merging two versions of such a mailbox, even if each version had been subject to arrivals and removals: the result should be the union of the two versions minus any messages that had been removed from either version.

In our view, it would be quite attractive if the basic directory system and supporting data structures of a general purpose filing environment could be selectively replicated, and automatically reconciled when partitioned updates occur. This would allow distributed filing systems to operate in a far more flexible and transparent manner than they do today. In fact, in order for a common, global name service in a very large scale filing environment to operate effectively, replication of the service is essential. So called "lazy" update schemes such as Clearinghouse [Schr84] which eventually come to agreement, but which may lose updates and may remain out of synch for some time even though communication has been reestablished, are not suitable for computer based directory services which programs as well as people interrogate and update. Other work addressing the replicated directory problem [Fisc82, Allc83, Wuu84] fails to exploit the full range of directory structure and semantics.

A resilient service is also very useful in the small, when a private work group is spread geographically. Since most files can often be expected to be updated from a single location, automatic propagation and reconciliation of updates in the directory system is quite convenient.[1]

This paper describes a selective replicated file system service which automatically propagates and reconciles directory updates. A companion paper [Page90] describes the file system implementation in which the reconciliation algorithms are embedded. The focus here is on the robust reconciliation algorithms we have developed to support a novel optimistic concurrency control policy, *one-copy availability*. Although our discussion is motivated by file system directory services, the algorithms are applicable to graph structures that share similar consistency constraints.

The next section presents filesystem, reconciliation, and graph models, including discussions of robustness and correctness. Section three describes the directory reconciliation algorithm, in terms of edges in a graph. Section four presents a distributed garbage collection algorithm for general graph structures that is used to reclaim file storage, and lays out how conflicting updates to files are detected. The method is applicable to updates to any vertex in a graph. Concluding remarks comprise section five.

2 Model

In this section we present our models of filesystems, replication, reconciliation, and our definition of correct operation in the face of partitions.

[1]Commercial PC work group packages such as Lotus Notes contain a simple version of such a facility.

2.1 File system

In our model, a filesystem client sees a connected, singly-rooted, directed acyclic graph of *directories* and *files*. All interior vertices are directories; each file is a leaf. Both files and directories may have more than one name (labeled edge).[2] A client "names" a file by specifying a path of names from the root to the file.

Directory updates are limited by several semantic constraints. A name may be added to a directory provided cycles are not introduced and the name does not already appear in the directory. New files and directories may be created and named in any existing directory. A name for a file may be deleted at any time; a file with no names is permanently inaccessible.[3]

Specific filesystem operations exist for file and directory creation, and name creation and deletion. A generic operation for updating a file is assumed to exist; a directory is updated only by the name creation and deletion operations. No explicit file removal operation is required; removal (via garbage collection) is simply a potential side-effect of name deletion.

2.2 Replication and reconciliation

The *logical* ("one-copy") filesystem model presented to a client is supported physically by one or more filesystem *replicas*, each containing the root directory and a subgraph of some or all of the files and other directories of the logical filesystem. An operation (e.g., create a name) is applied to a single physical replica and then followed by the execution of a *reconciliation algorithm* to propagate the effects of the operation to other replicas.[4]

Each filesystem replica is maintained by an instance of a filesystem reconciliation algorithm.[5] A reconciliation algorithm consults with other filesystem replicas regarding their current state and (partial) histories of the operations that have been applied to their replica, and attempts to determine the "reconciled union" of the replicas. If no conflicting replica states or operation histories are discovered, the (single) replica being maintained by the algorithm instance is "brought up to date" if necessary — "up to date" being the reconciled union of the replicas just consulted. The replica is not brought up to date if a conflict among replicas is discovered that requires additional semantic context to resolve. A conflict resolution procedure must be employed (perhaps at a later time) to resolve the conflict and establish a reconciled descendant of the conflicting replicas.

A filesystem reconciliation instance decides on its own how much of the filesystem is to be reconciled. As little as a single file or directory may be reconciled in response to a notification of update activity at another replica, or the entire directory graph may be reconciled when disrupted communications have been restored.

2.3 Robustness

Reconciling an entire filesystem directory hierarchy (containing perhaps 10,000 or more files and directories) is likely to span a significant period of time. Unfortunately, as the distribution of failures tends to be bursty, the point at which a full filesystem reconciliation is most likely to be initiated (restoration of communication) falls within a period characterized by a high probability of further failures. In order to guarantee termination, a reconciliation algorithm must therefore be extraordinarily tolerant of mid-reconciliation failures.

---

[2] One distinguished directory is called the *root* and has no labeled edges pointing to it.

[3] Exceptions to this are noted in section 4.

[4] Our algorithms are orthogonal to synchronization. A set of replicas within a LAN could be tightly synchronized such that a reconciliation algorithm considers them to be a single replica. The reconciliation algorithms could then be used simply to propagate updates between clusters of replicas, rather than within a cluster.

[5] A filesystem replica is always accompanied by an instance of the reconciliation algorithm: they "fail" together.

Tolerance of failures takes several forms. First, reconciliation work accomplished up to the point of a failure should, for the most part, not be lost as a result of the failure. In terms of directories and files, this suggests that each file and directory should be (largely) separately reconcilable. Second, filesystem reconciliation should be able to make progress in between multiple errors during a complete filesystem reconciliation, since a sufficiently long failure-free period may never occur.[6] The reconciliation algorithm should be able to start reconciling anywhere in the filesystem hierarchy, and proceed.

A reconciliation algorithm should also be robust in terms of the requirements it places on underlying communications services. For example, an algorithm that must communicate with other replicas in a specific order may have great difficulty making progress if communications links are not available in the order and at the times the algorithm desires. Or, an algorithm that expects direct communication with any particular (or all) replicas requires either very powerful store-and-forward support, or very robust communications links between itself and other replica(s).

In our view, a robust reconciliation algorithm should tolerate the permanent absence of direct communications links between arbitrary replicas. Links must still, of course, form a connected graph of communicating replicas over time, but the connections may be disjoint in time; simultaneous full connectivity is not required. The reconciliation algorithm must, therefore, provide any necessary intermediate communications between replicas.

A filesystem reconciliation algorithm should coexist satisfactorily with normal filesystem service to applications and users. Normal service should not be blocked while a filesystem is reconciled, nor should reconciliation activity need to wait for a quiescent filesystem. Reconciliation algorithms should also be sparing in their use of resources such as cpu time, disk space, and network bandwidth.

Our algorithms are robust in all of these senses. Each file and directory is atomically reconciled, and the algorithm may be started (or restarted) at any point in the directory graph. The graph may be traversed in any order (even random), although some orders make progress faster than others. Our algorithm also tolerates any underlying communications service that provides a connected topology of replicas over a finite period of time. No ordering of filesystem replicas is assumed or required.

2.4 One-copy availability

In the literature, serializability is often assumed to be the "only" correctness definition worth considering. We propose that a weaker criterion, which we call *one-copy availability*, is sufficient for managing concurrent access to, and updates of, a filesystem directory structure.

OCA guarantees that the *effects* of every operation are incorporated by every replica, and not that every operation is *applied* to every replica. For example, suppose the same name is concurrently deleted in separate partitions. OCA accepts this situation, based on its effects: the name is deleted. (The situation is clearly not serializable, as one of the deletions would have been rejected because a single name cannot be deleted twice.)

File updates are managed by OCA in an optimistic, serializable fashion. In the absence of knowledge about file semantics, OCA allows concurrent partitioned update of file replicas and *detects* non-serializable updates.[7] Upon detection of (non-serializable) concurrent updates, the file's owner is notified and is expected to take action to resolve the conflict. Non-serializable file updates result in an *update/update* conflict.

The order of directory updates is not preserved by OCA: each replica learns of every update, but in an arbitrary order. Immediate propagation is used to assist other

[6] Imagine reconciling two replicas connected through several dozen gateways and low-bandwidth, high-delay intermediate links. The probability of some form of communications failure during full filesystem reconciliation is rather high.

[7] This applies to a single file; multi-file transaction support could be provided by a higher level mechanism.

communicating replicas in becoming aware of updates from a particular replica in the order originally applied.

Concurrent creation of the same name in distinct partitions is allowed by OCA; the (similar) names are propagated to each replica even though the internal consistency of a directory has been violated. The existence of a *name conflict* is indicated by marking the non-unique names to block translation of the name until the conflict is resolved.

Filesystems that use OCA detect and tolerate *remove/update conflicts*, which occur when a file is concurrently updated in one partition, and removed in another. Our filesystem reconciliation algorithm is able to detect and handle remove/update conflicts even though the filesystem model does not include an explicit file removal operation.

A file with no names in any filesystem replica is involved in a remove/update conflict *unless* some name for the file was deleted in a filesystem replica containing the latest version of the file. If such a name deletion operation has not occurred, or there is no latest version of the file (because of an update/update conflict) a remove/update conflict exists.

One-copy availability (OCA) provides equivalent read and update availability: if any copy of a file or directory is present, an update proceeds in accordance with the concurrency control policy being enforced among communicating replicas. Since conflicts are likely to be rare, a considerable improvement in availability is achieved at low cost.

2.5 Graph model

Describing filesystem reconciliation algorithms is aided by transforming the discussion into graph terminology and working with graph formalisms, while translating back into filesystem language as needed to maintain an intuitive feel for what is happening and why.

The filesystem model of section 2.1 described a rooted DAG of directories and files. The graph model contains a rooted DAG of vertices and edges: directories correspond to interior vertices, files to leaves, and names to edges.

A directory vertex is interesting only in that it provides a source vertex for edges; it is the edges themselves (singly or grouped by source vertex) that are important to us. In the sequel, a "directory" is the collection of edges emanating from a particular vertex.

Restating from section 2.2, a *logical* graph is represented by one or more *physical* graph replicas, each containing the root vertex and a subgraph of some or all of the other vertices and edges of the logical graph. A graph operation (e.g., insert_edge or update_leaf) is applied to a single physical graph replica, and followed at some point by the execution of a *graph reconciliation algorithm.*

Our graph reconciliation algorithm is composed of *edge reconciliation* and *leaf reconciliation* algorithms. The edge algorithm is a distributed *two-phase* algorithm that reconciles the effects of edge replica creation and removal operations. The leaf reconciliation algorithm is composed of several algorithms, most of which are two-phase and may be executed in parallel.

2.6 Two-phase algorithms

Distributed two-phase algorithms (without single coordinator sites) are at the heart of our reconciliation algorithms. Each graph replica "executes" an instance of an edge (or leaf) reconciliation algorithm for each edge (leaf) it contains. The collection of algorithm instances for a particular logical edge (leaf) operate together in phases, such that a) all instances have either not yet begun the algorithm or are in phase one; or, b) all instances are in phase one or two; or c), all are in phase two or have completed execution of the algorithm.

The first phase of our two-phase algorithms typically ascertains that some interesting global stable state [Chan85] exists, e.g., "all file replicas are inaccessible." The second phase verifies that every replica is informed that the global stable state of phase one has been established, and provides the foundation for ensuring termination of the collection of algorithm instances: the second phase is normally followed by destruction of

a replica, which could otherwise confuse algorithm instances for other replicas.

One should note that guaranteeing correct operation of these algorithms is far more difficult that it may at first appear. For example, it is not sufficient for a replica to be deleted as soon as it knows that all replicas have been notified of the delete operation; there are conditions under which the algorithm would never terminate.

In some of our algorithms, "phase one" is itself a two-phase algorithm. This is necessary when the collected state information has the potential to become out-of-date and must be reconfirmed by a second phase, as in the case of file replica inaccessibility. A zero reference count for a vertex replica is not a stable state: additional vertex names which exist in other graph replicas may yet be propagated into a replica currently containing no names.

The instances of an algorithm for a particular edge (leaf) cooperate largely by sharing global state information. The state data is usually a bit vector (for edges) or an integer vector (for leaves) with a component corresponding to each replica. The data in a vector component is initially supplied by the corresponding replica algorithm instance, and then is freely passed among the entire set of replicas.

Sharing state vectors both promotes efficient reconciliation and is necessary to tolerate (i.e., make progress despite) the permanent absence of communications links. For example, suppose a set of replicas is connected in a linear topology, in which each replica can only communicate directly with its (one or two) neighbors. To make any progress at all in reconciling replicas, state data must be shared.

3 Directory Reconciliation Algorithm

The directory reconciliation algorithm is responsible for reconciling and propagating the effects of `insert_edge` and `delete_edge` operations. Such effects include creating and deleting edge replicas, and incrementing and decrementing reference counts of the target directory or file. A new file or directory replica may also be created in the process of creating an edge replica.

3.1 Log-less reconciliation

A reconciliation algorithm that purports to understand the semantics of operations and reconcile updated replicas must somehow determine what operations have actually occurred. Rather than maintain a log of operations for each replica, our algorithm compares two replicas and *infers* what operations have occurred and what action should be taken to make the local replica reflect remote activity.

A fundamental difficulty of the "log-less" approach is the *insert/delete ambiguity* problem noted in [Fischer82]: simply comparing two object replicas whose operations are insert and delete does not yield sufficient data to correctly determine if an item present in just one of the replicas has been newly created or recently removed. We address this difficulty by *logically*, but not *physically*, deleting an edge in a graph replica when a `delete_edge` operation occurs. The edge reconciliation algorithm is then able to distinguish a logically deleted edge from a nonexistent one, and so the insert/delete ambiguity is eliminated.

3.2 Garbage collection

The differentiation between logical and physical deletion substitutes a *garbage collection* problem in place of the ambiguity problem. While several distributed garbage collection algorithms have appeared, none is both inexpensive and sufficiently robust for our requirements. We therefore developed a low-overhead, robust, two-phase distributed garbage collection algorithm to dispose of logically deleted edge replicas.

The algorithm has two important properties: monotonicity and low-cost indirect communication. The monotonicity property is both global (to ensure termination) and local (to prevent repeated sequences of deallocation/allocation of a physically deleted edge replica). Indirect communication is a necessary property for distributed algorithms in

systems in which all-pairs communication is not guaranteed.

The two-phase distributed garbage collection algorithm is executed by each logically deleted edge replica. Each instance of the algorithm maintains two bit-vectors (V1 and V2) of length equal to the replica list attribute. A bit vector component V[i] corresponds to the i-th edge replica.

Vector V1 is used by the first phase of the algorithm. A bit set in this vector indicates that the corresponding edge replica is marked logically deleted. Vector V2 is used in the second phase to indicate with a set bit that the corresponding edge replica has completed phase one.

3.2.1 Phase One

The first phase of the algorithm begins when a `delete_edge` operation occurs. The edge reconciliation algorithm instance for the replica to which the operation was applied sets the bit in the V1 vector corresponding to its local replica. Further bits in vector V1 are set as the algorithm consults with other replicas, obtains a copy of their phase one vectors, and pair-wise logically ORs the remote vector components into the local vector. Phase one is complete when V1 has all bits set.

At this point in the execution of an instance of the edge reconciliation algorithm, the local edge replica knows that *all* edge replicas have been marked logically deleted. This fact will be used later to prevent non-monotonic re-creation of edge replicas. Note that all other edge replicas have at least begun phase one of the algorithm, but little is known about their progress.

3.2.2 Phase Two

An instance of the edge reconciliation algorithm begins the second phase upon its completion of phase one by setting V2[self]. The remaining unset bits in the second vector are set by ORing the contents of remote V2 vectors into the local vector. Phase two is complete when all bits in V2 are set.

When V2 is complete, the local instance of the edge reconciliation algorithm knows that "every other replica knows my replica (and all others) are marked logically deleted." This sets the stage for physically deleting the local replica, without risking non-monotonic re-creation of the local, logically deleted edge replica.

Upon completion of phase two, the local logically deleted edge replica is physically removed, and the local instance of the two-phase garbage collection algorithm terminates.

Frequently, when a remote replica is consulted during phase two, the remote replica does not physically exist: it has finished phase two quicker than the local replica, and has been physically deleted. The local edge reconciliation algorithm is able to infer this fact immediately because its complete V1 vector guarantees that the remote replica once existed and was marked logically deleted. Since the remote replica does not currently exist, it must have already completed phase two. In response, the local edge algorithm declares itself completed with phase two, and physically removes the local edge replica.

3.3.3 Discussion

The quantity of information maintained by each edge reconciliation algorithm instance is linear in the number of replicas, as is the amount of data passed between replicas. The low storage overhead is achieved by carefully structuring the *meaning* of the information: the context is always implicit. For example, a bit set in V1 means "edge replica i is logically deleted" and a bit set in V2 means "edge replica i knows all replicas have been logically deleted."

An alternative approach is to maintain (and share) a *bit matrix*, in which the meaning of a set bit is "replica i knows that replica j is logically deleted." In some circumstances, the more detailed information would enable an edge replica to be physically deleted more quickly than with the bit vector approach. The tradeoff is quadratic versus linear storage and message length costs, although fewer messages may be exchanged.

In this algorithm, logical deletion is a local stable state: there is no way to reverse or undo the logical deletion of an edge replica. The first phase determines that logical deletion is a global stable state, and the second phase ensures that storage allocation and reclamation occurs only once per edge replica.

In other applications, such as file storage reclamation, the global stable state of interest (zero-valued reference counts) does not have a local analog. A local zero reference count is not a stable state, as another graph replica may well contain new edges (references) about which the local replica as yet has no knowledge. For this kind of situation, a more complex method is required. The next section presents a solution to the file reclamation problem which is based upon a two-phase algorithm.

4 File reconciliation algorithm

The file (leaf) reconciliation algorithm is actually a composite of three algorithms: leaf version reconciliation, global inaccessibility determination, and remove/update conflict detection. Although the algorithms logically must execute in the order listed with no overlap, in practice they tend to execute in parallel.

The parallel execution occurs even though it does not seem necessary at the time. For example, a remove/update conflict cannot be determined until after global inaccessibility is established, but the data necessary to declare a remove/update conflict can be collected while data for global inaccessibility is being obtained.

This overlap is essential for reasonable termination of the leaf reconciliation algorithm, as the component algorithms are mostly two phase, and in one case composed of two-phase sub-algorithms.

4.1 Leaf version reconciliation algorithm

The leaf version reconciliation algorithm is based upon the *version vector* concurrent partitioned update detection scheme of [Parker81]. A version vector is a multidimensional version number, with one component for each dimension (replica). Each leaf replica maintains a complete version vector as a mutable attribute.

When an `update_leaf` operation occurs on a replica, the corresponding version vector component is incremented. The leaf reconciliation algorithm compares two version vectors to determine if they are equal (no concurrent updates occurred), dominant/subordinate (one, but not both, of the replicas was updated), or in conflict (both replicas were updated).

If the remote replica's version vector dominates the local one, the local replica is made to be equivalent to the remote replica, including the version vector. If the vectors are equal, or the local vector dominates, no action is taken. If the version vectors conflict, the local replica is marked in conflict, and the client is notified. Pending conflict resolution by the client, access to a conflicted replica is denied. Conflict resolution sets the local version vector to be strictly greater than the version vectors of all replicas consulted in the resolution activity.

4.2 Global inaccessibility algorithm

In our graph model, the absence of edges to a leaf implies that the leaf needs to be garbage collected, as there is no further way for a client to access a label-less leaf. Replication introduces several complications not present in traditional graph management: local inaccessibility does not imply global inaccessibility, and a significantly greater opportunity exists for updates to be inadvertently lost as a result of unsynchronized, concurrent `delete_edge` and `update_leaf` operations.

A leaf replica becomes locally inaccessible when no available graph replica contains a logically existing edge to the leaf. It is possible that additional edges have been created in a graph replica that is not currently available, or that an unavailable leaf replica has been updated. Either possibility suggests that garbage collecting a locally inaccessible

leaf replica is premature, at least until global inaccessibility has been established. In the first case, extra effort will have to be expended to recreate the leaf replica--burdensome, but not incorrect. In the second case, updates will have been quietly lost, with no mechanism by which the clients involved could possibly have ascertained the impending situation.

Determining global inaccessibility is a *stable-state* detection problem [Chan85]. We exploit knowledge of how edges are created to obtain a lower-cost algorithm than the general algorithms previously published.

Our model utilizes the current reference counter attribute (*cref*) of a leaf replica to determine whether the replica is locally accessible. In addition, the history reference counter attribute (*href*) is maintained as a monotonically increasing counter. The href counter is incremented in parallel with the cref counter. The global inaccessibility task is to determine that every leaf replica's current reference counter is simultaneously zero, even when not all graph replicas can be simultaneously available.

The globally inaccessibility algorithm is executed on behalf of each leaf replica when its *cref* drops to zero. Two vectors (V1 and V2) are maintained, each containing an *href* for each replica. Vector V1 is used in the first phase to record the history reference counter of each replica that, when consulted, also has a zero current reference count. The first phase is complete when vector V1 has an entry for each replica in the replica list.

When phase one is complete, the local instance of the global inaccessibility algorithm knows that every other leaf replica has had a zero current reference count at some point in the past, and it knows what each replica's href value was at one of those times. Note that the local algorithm has no knowledge of the current state of affairs at any other replica; the V1 vector is strictly information about the past.

The second phase verifies the validity of vector V1 by checking to ensure that none of the history counters have changed. The new *href* values are stored in V2. However, values are only obtained from leaf replicas that have completed phase one. Phase two is complete when V2 is complete, and so the global inaccessibility algorithm is complete.

If at any time during phase one a replica is encountered whose reference count is non-zero, the algorithm initializes itself and begins anew. During the second phase, any non-zero reference count or changed history count causes the algorithm to initialize itself, and phase one (not two) begins anew. And, of course, if the local reference count becomes non-zero the algorithm terminates immediately.

The historical reference counter values collected in phase one establish a landmark by which the creation of new edges to the leaf can be detected. Creation of a new edge causes some leaf replica's href attribute to be incremented; the new value is guaranteed to be noticed during the second phase, which (possibly indirectly) re-consults each href value.

The creation of a new edge requires the existence of another edge replica to the target leaf. This condition allows the second phase of the global inaccessibility algorithm to steadily narrow down the set of graph replicas that might be capable of creating new edges and thereby prevent the determination of global inaccessibility.

As with the previous algorithms, the global inaccessibility algorithm instances exchange data. In this case, the *href* vectors are shared.

4.3 Remove/update conflict detection algorithm

The remove/update detection algorithm is executed in parallel with the global inaccessibility algorithm, in the hope that global inaccessibility will be established. The remove/update algorithm attempts to determine the greatest version vector of all leaf replicas, and the greatest *removal vector* amongst all replicas.

The removal vector is a copy of a replica's version vector made at the time of a `delete_edge` operation. (It is overwritten by a succeeding `delete_edge` operation on

the same leaf replica.) The removal vector records the version visible to the initiator of the `delete_edge` operation.

If the dominant removal vector is equivalent to the dominant version vector, no remove/update conflict exists; otherwise, a conflict exists, and an edge from the *orphanage* directory to the leaf will be created.

A replica determines the dominant version vector by comparing its version vector with each of the other version vectors, and replacing its local vector with the dominant vector from each comparison. After comparing with each replica, the local version vector is dominant.

Two problems arise: lack of direct communications with each other replica, and the possibility that no dominant vector exists. (Vectors are partially, not totally, ordered.)

Indirect comparison of version vectors is accomplished by maintaining a bit vector indicating which replicas have been consulted. The bit vector and version vector are shared among replicas; when the bit vector is complete, all replicas have been consulted. If a dominant vector exists, the local vector contains that value.

It is possible that no dominant version vector (or removal vector) exists if any update/update conflicts arose in the course of a file's usage. If the conflict still exists, no dominant version vector will exist. Even if the conflict was resolved, the removal vectors may still contain conflicting vectors, since they record past state. In either case, the above algorithm to determine a dominant vector must be augmented to tolerate the nonexistence of a dominant vector.

The "conflict tolerant" algorithm executes in two phases. The first phase operates as indicated above. The local vector is replaced only by a dominant vector; subordinate, equal, or conflicting vectors are ignored. However, the bit vector continues to record which replicas have been consulted.

The second phase consults replicas which have completed the first phase, and compares version vectors again. Two bit vectors are maintained: one indicates which replicas have been consulted in phase two, and the second indicates whether the vectors are equal. At the completion of phase two (the "consulted" vector is complete) the "equality" vector is checked for "unequal" indications. If all replicas have equal vectors, the local vector is already set to the value of the dominant vector. Any "unequal" indication implies that no dominant vector exists.

The above two-phase dominance-detection algorithm is executed in parallel for both the version vector and the removal vector. A remove/update conflict exists if either vector was not determined to be dominant over its replicas, or if the dominant removal vector does not equal the dominant version vector.

5 Conclusion

In this paper, we have motivated a loosely coupled approach to replicated file systems that is decentralized enough to have the potential to scale to very large systems. We have argued that in order for update availability not to decrease as replication increases, a single copy availability policy must be supported. The framework for the algorithms necessary to make such loosely coupled replicated file systems operate successfully has been outlined in terms of a general DAG structure.

We presented algorithms for managing directory replicas, in particular a new distributed garbage collection algorithm for read-only objects. We also described algorithms for managing replicated graph structures.

Some of the algorithms, such as a general distributed garbage collection method, may have applicability outside this particular context.

References

[Allc83] J. Allchin, "A Suite of Robust Algorithms for Maintaining Replicated Data Using Weak Consistency Conditions," Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems, October, 1983.

[Alsb76] P. A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources," Proceedings of the Second International Conference on Software Engineering, October, 1976.

[Bloch84] J. Bloch, D. Daniels, A. Spector, "Weighted Voting for Directories," CMU Technical Report CMU-CS-84-114, 1984.

[Chan85] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," ACM Transactions on Computer Systems, February, 1985.

[Fisc82] M. Fischer, A. Michael, "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," Proceedings of the ACM Symposium on Principles of Database Systems, March, 1982.

[Giff79] D. K. Gifford, "Weighted Voting for Replicated Data," Proceedings of the Seventh Symposium on Operating Systems Principles, December, 1979.

[Herl86] M. Herlihy, "A Quorum-Consensus Replication Method for Abstract Data Types," ACM Transactions on Computer Systems, February, 1986.

[Noe86] J. Noe, A. Proudfoot, C. Pu, "Replication in Distributed Systems: The Eden Experience," Proceedings of Fall Joint Computer Conference, November, 1986.

[Page90] T. Page, G. Popek, R. Guy, J. Heidemann, "The Ficus File System: Replication for NFS via Stackable Layers," submitted to 1990 Symposium on Reliable Distributed Systems.

[Park83] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," IEEE Transactions on Software Engineering, May, 1983.

[Stone79] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," IEEE Transactions on Software Engineering, May, 1979.

[Thom78] R. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM Transactions on Database Systems, June, 1979.

[Wuu84] G. Wuu, A. Bernstein, "Efficient Solutions to the Replicated Log and Dictionary Problems," Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, August, 1984.