(1)

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Ma. 02139

Programming Methodology Group Memo 48

Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection

Barbara Liskov
Rivka Ladin

1986

(Note: To appear in the Proceedings of the 5th ACM Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada, August 1986)

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125 and in part by the National Science Foundation under grant DCR-85-03662.

May 1986

Abstract

This paper describes two techniques that are only loosely related. The first is a method for constructing a highly available service for use in a distributed system. The service presents its clients with a consistent view of its state, but the view may contain old information. Clients can indicate how recent the information must be. The method can be used in any application in which the property of interest is stable: once the property becomes true, it remains true forever.

The paper also describes a fault-tolerant garbage collection method for a distributed heap. The method is practical and efficient. Each computer that contains part of the heap does local garbage collection independently, using whatever algorithm it chooses, and without needing to communicate with the other computers that contain parts of the heap. The highly available central service is used to store information about inter-computer references.

1. Introduction

This paper describes two techniques that are only loosely related. The first is a method for constructing a highly available service for use in a distributed system. The service presents its clients with a consistent view of its state, but the view may contain old information. Clients can indicate how recent the information must be. The method was invented as a way of optimizing the orphan detection strategy developed for the Argus language and system [16], [17]. However, the technique appears to be applicable to a wide range of applications. We have found three others so far—garbage collection of objects in a distributed heap, locating movable objects in a distributed system, and deletion of unused versions in a hybrid concurrency control scheme [21]. The method requires that the property of interest to the application is *stable* [4], i.e., once the property becomes true, it remains true forever. For example, once an object in a heap becomes inaccessible, it will always be inaccessible.

The second contribution of the paper is a fault-tolerant garbage collection method for a distributed heap. The method is practical and efficient. Each computer that contains part of the heap does local garbage collection independently, using whatever algorithm it chooses, and without needing to communicate with the other computers that contain other parts of the heap. The highly available central service is used to store information about inter-computer references. The computers containing the heap communicate with the central service periodically, to inform it about their references to objects at other sites, and to inquire about the accessibility of any local objects that might be referred to at other sites. The method propagates information about accessibility quickly and makes that information highly available. In addition, it off-loads the work of distributed garbage collection from the computers storing the heap, and does not delay them in carrying out user computations.

The methods are intended to run in an environment in which individual computers, or *nodes*, are connected by a communications network. Both the nodes and the network may fail; the methods tolerate these failures. The nodes are failstop processors [18]; we assume they can crash, but Byzantine failures are not expected. We assume that nodes do eventually recover from crashes, and that each node has access to a stable storage device that (with very high probability) preserves the information entrusted to it [11]. After a crash, a node can recover the portion of its state that was written to its stable storage device before the crash.

The network connecting the nodes can have an arbitrary topology. For example, it might consist of a number of local area nets connected via gateways to a long-haul network. Again we rule out Byzantine failures, but otherwise the network can behave arbitrarily badly. For example, it can partition. Therefore messages can be lost, delayed, duplicated, and delivered out of order.

We assume that the nodes are loosely synchronized: each node has a local clock, and the skew of these clocks is bounded by some ε. Since our algorithms do not require a small value for ε, this assumption is reasonable [14], [13]. This assumption permits us to discard delayed messages and thus treat them like lost messages.

We begin in section 2 by describing our method for constructing highly available distributed services. Then, in section 3, we describe our garbage collection technique. We conclude with a summary of what we have accomplished.

2. The Highly-Available Service

We are concerned here with a distributed implementation of a logically centralized service. Information stored by the service is replicated at different nodes of the network. Reasons for implementing a service in this way are improved availability and response time. However, in spite of its distributed implementation, the service is logically centralized in the sense that it appears as a single entity to clients. The fact that its implementation is distributed is not visible at the client level.

When data of a service is replicated at several nodes in this way, the information at the different replicas may not be identical. Clients can observe the information at the replicas only by making operation calls. A useful distinction among replication schemes is whether operation calls expose inconsistent information to clients.

A good way to keep inconsistencies hidden from clients is to use a voting scheme as described in [6]. In such a scheme, the nodes visited by operations that read the service state must intersect those visited by operations that modify the state. There is some freedom in choosing how to implement the operations; for example, if there are three nodes, information could be written to three nodes and read from only one, or information could be both read and written at two nodes. The former choice enhances the availability and response time of reading at the expense of writing; the availability of writing is a problem since all three nodes need to be up and accessible, although this problem can be mitigated to some extent by using the technique described in [19]. Any choice made for reading and writing results either in some operations having availability problems (the write operations in the first choice) or in all operations requiring more message passing than is needed if the service is not replicated.

An attractive alternative to voting schemes is to have both reading and writing take place at only one of the replicas, and have the replicas communicate new information among themselves in background mode by exchanging "gossip" messages. Such a scheme is described in [5]. As long as things are running well, the information will propagate quickly, so the amount of time the replicas contain inconsistent information will be small. If there are crashes and partitions, however, inconsistencies can persist for long periods of time, and there is no guarantee that information obtained from the service is accurate. Inaccurate information is not a problem in some cases. For example, in a mail system (e.g., [7]), a request to read mail need not produce all messages that have been sent; a promise of timely delivery is sufficient. Other applications require accurate information. For example, a service that could only tell you that an object *might* be inaccessible would be useless for garbage collection.

The scheme to be described below performs operations at a single replica and propagates information in background mode. It retains the availability and response time advantages of the gossip scheme, but can be used by clients, such as a garbage collector, that require accurate information.

The remainder of this section presents our approach. We begin by describing a particular service to be implemented by the technique. Then we describe the processing that takes place at the replicas. Next we discuss the performance of our method. We conclude by discussing the general properties of applications that can use the method.

2.1. Our Approach

This section describes our approach in detail. To make the discussion concrete, we descibe a particular service, which allows clients to map unique names (uids) to integers. This *map* service stores part of the information needed for orphan detection; in particular, it is used to detect orphans that are caused by crashes (see [20] for a discussion of orphan detection). The uids are the names of guardians, which are the entities that can crash in Argus. Associated with each guardian name is its *crash count*, which records the number of times the guardian has recovered from crashes.

Since we intend to execute operations at just one replica, and to propagate the new information (e.g., increased crash counts) in background mode, it is possible that the replica consulted by an operation to look up a crash count may contain out-of-date information. To allow clients to make sense of a response based on partial information, we mark each response with a timestamp [10]. The timestamp identifies a particular state of the service, and the response is guaranteed to be accurate for that state. The timestamps are partially ordered; larger timestamps are associated with more recent states. It is up to the client to determine how recent a state is of interest to it; the client can request information that is at least as recent as a given timestamp. Timestamps can be merged; merging t_1 and t_2 produces a timestamp t_3 that is $\geq$ both t_1 and t_2. Clients merge timestamps to request information that is at least as recent as the states associated with the two merged timestamps.

A specification of the service is shown in figure 1. In this specification, the service is modelled as a set of states, each one marked with a timestamp, and all operations return the timestamp of some state. Later states (those with larger timestamps) map uids to larger integers. Operations *enter* and *delete* cause new states to be added to the set, while *lookup* observes existing states. (See [12] for a discussion of such specifications.)

Whenever information about an association is entered (by calling the *enter* operation), the server returns the timestamp of a state that results from entering the new association. This state associates the uid with a value at least as large as the one passed as an argument to *enter*. The *enter* operation can be used to change the associated value many times. An association can be deleted by calling the *delete* operation; this associates the uid with a value larger than any integer (∞). The *lookup* operation takes a timestamp as an argument; it returns information based on a state with a timestamp at least as large as that timestamp. *Lookup* must wait until a state with a large enough timestamp exists; in practice, however, clients of the map service make calls with timestamps for which states already exist.

Note that all the operations are non-deterministic. For each call there are many states whose map = **data type is** enter, delete, lookup

Overview

The map service associates uids with integers. It consists of a set σ of states, each marked with a unique timestamp. Each state maps uids to integers. *Enter* and *delete* may cause new states to be added to σ. States with larger timestamps associate larger values with the uids: If t_1 and t_2 are timestamps and s_1 and s_2 are their associated states, respectively, then

$$t_1 < t_2 \Rightarrow \forall\, u: \text{uid}\ [s_1(u) \leq s_2(u) \vee (s_1(u) \text{ is undefined } \&\ s_2(u) \text{ is defined})]$$

Operations

enter = **proc** (u: uid, x: integer) **returns** (timestamp)
modifies σ
effects May return the timestamp of a state s such that $s(u) > x$. In this case, s is already in σ. Otherwise, returns the timestamp of state s such that $s(u) = x$. In this case, $\exists\, s' \in \sigma$ such that $s(v) = s'(v)$ for all $v \neq u$ & $(s'(u) < x \vee s'$ is not defined on u) Adds s to σ if it isn't already there; many such states may be created and added to σ. For each state added, the associated timestamp is created in a way that satisfies the invariant.

delete = **proc** (u: uid) **returns** (timestamp)
modifies σ
effects Returns the timestamp of a state s such that $s(u) = \infty$ and $\exists\, s' \in \sigma$ such that $s(v) = s'(v)$ for all $v \neq u$ Adds s to σ if it isn't already there; many such states may be created and added to σ. For each state added, the associated timestamp is created in a way that satisfies the invariant.

lookup = **proc** (u: uid, t: timestamp) **returns** (integer, timestamp)
signals (not_known(timestamp))
effects Returns a timestamp $t' \geq t$ and information obtained from the state s' associated with t'. If u maps to ∞ in s' or u is not defined in s', signals *not_known*; otherwise returns s'(u).

end map

Figure 1: The service operations.

timestamp could be returned; the operation simply returns one of these timestamps, possibly with some associated information. However, we guarantee that the state corresponding to the returned timestamp contains the specified information.

The service is modeled as an abstract data type. The idea is that clients call operations of the type. The called operation then sends an appropriate message to a replica; there is a message type corresponding to each operation. The replica responds by sending back a reply message. If the response is slow, the operation may send the message to a different replica. Therefore a single operation call may result in messages being sent to several different replicas. As mentioned above, we assume messages can be lost, delayed, duplicated, and delivered out of order.

2.2. Processing at the Replicas

The next two sections describe the processing at the replicas. In what follows we describe the processing and the stored information in a general way, without considering possible optimizations.

Each replica maintains a state and the timestamp of that state. The state maps uids either to integers or to a special value representing ∞. Replicas respond to messages sent by operations by either looking up information in the state, or by generating a new local state and timestamp. They communicate with one another periodically by exchanging gossip messages.

The main problem that must be solved to make this work efficiently is the generation of the timestamps. Replicas must be able to generate new timestamps independently, or this service will be dependent on a timestamp service posing exactly the same problems we are trying to solve. Furthermore, we must preserve the invariant mentioned above, namely, that later timestamps must not be associated with states containing older information.

We solve this problem by using *multipart timestamps*, where there is one part for each replica. Thus if there are n replicas, a timestamp t is

$$t = \langle t_1, ..., t_n \rangle$$

where each part is a positive integer. Since there will typically be a small number of replicas (e.g., 3 to 7), using such a timestamp is practical.

A replica generates a new timestamp by incrementing its part of its timestamp by one, while leaving all other parts unchanged. Since each part can be advanced by only a single replica, we guarantee that the resulting timestamps are unique. Some timestamps can be compared: For two timestamps t1 and t2, t1 $\leq$ t2 provided $t1_i \leq t2_i$ for each part i of the timestamp. Other timestamps are incomparable. Two timestamps t1 and t2 are merged by retaining the larger value for each part; as required, the result of the merge is $\geq$ t1 and t2.

The messages are processed at the replicas as follows: If the replica receives an *enter*(*u*, *x*) message, it looks in its local state to see if there is an association for u. If there is no association, or if u is mapped to a value less than x, then it associates x with u in its local state, advances its timestamp, and returns the new timestamp in a reply message. Otherwise it simply returns its current timestamp.

When a replica receives a *delete*(*u*) message, it looks up u in its local state. If u already maps to ∞ it returns its current timestamp; otherwise it associates ∞ with u in the local state, advances its timestamp and returns the new timestamp.

A *lookup*(u, t) message causes the replica to compare t with its timestamp, t_r. If $t > t_r$, then the replica needs more information. It either waits for gossip messages from the other replicas or sends a query to another replica to elicit the information. Communication between replicas is discussed below. As soon as $t \leq t_r$, it looks up u in its state. If u is mapped to an integer $< \infty$, it returns the associated integer and t_r; otherwise it returns t_r with the information that there is no association for u in the t_r state.

Periodically, a replica sends a gossip message containing its timestamp and its state to other replicas. When a replica receives a gossip message, it proceeds as follows: If the timestamp in the message is less than or equal to its timestamp, it discards the message since it is old. Otherwise it merges the timestamp of the message with its timestamp. The merged timestamp becomes the new timestamp of the replica. Then it merges the state of the message with its own by retaining the larger value for each association that is present in both states, and by retaining all associations that are present in only one state.

Replicas produce new states in response to gossip messages. The specification of the map service says nothing about gossip messages; instead, it indicates that states are created only when *enter* and *delete* operations are executed. However, clients cannot tell when the state associated with the timestamp returned by a call of *lookup* was created. Therefore, creating states late (in response to gossip messages) causes no harm.

Note that our timestamps do satisfy the required invariant that larger timestamps are not associated with older information:

1. The condition holds initially.

2. Each individual replica preserves the invariant whenever it receives an update message. A replica advances its timestamp only when it receives new information, and the new timestamp it generates is larger than its old one. Therefore it preserves the invariant locally. It also preserves it globally. Its new timestamp is either larger than that at another replica or incomparable with it. In the former case, its information was no older than that at the other replica before the update, so its information is still no older. In the latter case the invariant is satisfied trivially.

3. The exchange of a single gossip message preserves the invariant. The replica receiving the message produces a new timestamp that is larger than its previous timestamp, and a new state containing information no older than its previous state. Thus the invariant holds locally. The new timestamp is either greater than the timestamps of other replicas or incomparable with them; in either case the invariant holds.

2.3. Eliminating Deleted Entries

The implementation described above requires that entries mapping uids to ∞ be retained forever for the following reason. Suppose we discarded information about deleted uids. Then an association that has not yet been entered is identical to one that has been deleted. Therefore, when we perform an *enter* operation, we would not know whether to associate the uid with the integer argument or not.

Note that we would have no problem if clients never entered new information about a uid after it has been deleted. This constraint holds for the map service as used for orphan detection; here a single client is responsible for updating information about a particular uid, and once it has deleted the uid (meaning that the guardian named by the uid has been destroyed), it never updates information about that uid again. Therefore we will make this assumption about the clients. Other applications of the highly available service impose similar constraints on clients, and in all cases the clients satisfy these in a natural manner since the stored information typically involves unique names that are managed by a single client.

Even if clients never call *enter* after deleting an entry, there is still a problem with late messages. If an *enter* message is delivered late, it may arrive after the deleted entry has been discarded. To solve this problem, we have each message contain the time τ at which it was sent and we impose an upper bound δ on message delay. τ is the time of the local clock at the sending node; it should not be confused with the timestamps discussed above. Recall that we assume the clocks of all the nodes are synchronized with a maximum skew of ϵ [13], [14]. When a message arrives whose time τ is older than the time of the clock of the receiving replica by $\delta + \epsilon$, it is discarded. To handle late *enter* messages that are not discarded but arrive after the *delete* message, we retain information about deletes at least $\delta + \epsilon$ more than the time in the *delete* message.

Waiting $\delta + \epsilon$ is not sufficient however. Suppose replica r waited this long and then discarded an association for uid u. Then it sent a gossip message to replica s, which still had an association stored for u. There would be no way at s to decide whether the association for u should be retained or not.

To solve this problem, we proceed as follows. We continue to store information about deleted entries in the state. However, in such an entry, e, we store two additional pieces of information: *e.time*, the time of the delete message, and *e.ts*, the multipart timestamp generated when the delete message was processed. In addition, we maintain a replica table, *ts table*, containing a multipart timestamp for each replica. When a gossip message is processed, its timestamp is stored in ts table in the entry for the replica that sent the message. Notice that the real timestamp of the sending replica must be at least as large as the one stored for it in ts table.

It is safe to remove the entry e for a uid when:

1. its time e.time $+ \delta + \epsilon$ is less than than the time of the local clock of the replica, and
2. its timestamp e.ts is less than or equal to all the timestamps stored in the replica's ts-table.

The first condition takes care of late *enter* messages. The second guarantees that the entry is retained until we can be sure that a state mapping the uid to a normal value will never be received in a gossip message. Since the timestamps stored in ts-table are less the timestamps of their associated replicas, each replica has a timestamp $\geq$ e.ts, and therefore it has heard about the delete, and has stored this information in its state. Therefore any new gossip messages it generates will contain either information about the delete or no information for the entry at all. Its old, delayed gossip messages (containing a timestamp less than that of the receiving replica) might contain bad information, but they will be discarded.

One final point concerns duplicate *delete* messages processed at different replicas. In this case the states at the two replicas will contain slightly different information for the deleted uid. When such states are merged, the timestamp e.ts stored for the uid will be the merge of the two stored timestamps, and the time e.time will be the time of the later *delete* message. This ensures that the delete information remains long enough to eliminate any problems.

2.4. Performance

When this method can be used by an application, it has better response time and availability than a voting scheme. When the system is working well, i.e., replicas are up and accessible, then clients need to wait for a response from just one replica in carrying out any operation. In a voting scheme, at least some operations require responses from several replicas. Since in general not all replicas are equally close, this will slow down the clients.

When the system is not working well because of crashes and partitions, the performance of any replication scheme degrades. Our method allows updates to proceed but slows down queries; voting schemes typically do the opposite. However, the effect on clients is really the same, assuming that updates must happen. For example, for orphan detection to work the guardian must record its new crash count at the service; not allowing the update means the guardian cannot recover from the crash.

Our scheme provides better availability than a voting scheme because clients can specify that old information is acceptable. This increases the probability that the replica processing a query has the needed information.

It is important that the probability of loss of information by the service due to a crash of one of the replicas be acceptably low. An obvious way to achieve this is to have the replicas log new information on stable storage. Information received in both update and gossip messages must be logged; the stored information must include the replica's timestamp and the changes to its state. After a crash, a replica would restore its state from the log and then communicate with other replicas to get up to date.

One problem with this scheme is that there is a period of time during which just one replica knows a new piece of information. If the replica crashes during this period, the new information becomes inaccessible to other replicas until it recovers. The longer this period, the higher the probability of the information being inaccessible when needed by a client. To reduce this period, gossip about new information should be sent out frequently. For example, a replica might gossip about the new information to another replica at the same time that it replies to the client. Another possibility is for the client to send an update message simultaneously to several replicas; this would not slow the client down since it need wait for only one response.

Since replicas contain copies of one another's information, it is tempting to consider eliminating stable storage. Instead, before responding to an update message, a replica would communicate the new information to one or more other replicas and wait for them to respond. Then it could send the reply back to the client. The number of replicas involved would depend on the acceptable probability of information loss. Although this approach avoids the use of stable storage, it is likely to degrade system performance; it is similar to a voting scheme in which updates must happen at two copies.

2.5. Discussion

This section has described a method for constructing highly-available central services for use in a distributed system. The method was discussed with respect to a particular example, the map service. Below we describe the service in a more abstract way.

The service provides its clients with update and query operations. *Update* operations modify the service state; they return a timestamp of a state guaranteed to contain the new information or later information. *Enter* and *delete* are the update operations for the map service. *Query* operations take a timestamp as an argument and return some information and a timestamp as a result. The returned information is guaranteed to come from the state associated with the returned timestamp, and the returned timestamp is $\geq$ the argument timestamp. *Lookup* is the (only) query operation for the map service.

The implementation of the service must guarantee the invariant that new timestamps do not correspond to older information. This implies that there must be a way for the service to distinguish newer from older information. In the case of the map service, larger crash counts were more recent than smaller ones.

The method of distinguishing information comes from the application domain, and limits the set of applications that can use the technique. For example, consider a naming system in which clients generate user-friendly names that may conflict. Suppose two clients enter the same name at approximately the same time. To make use of the service for storing the names, these enters must be ordered: the higher ordered one is the one that will persist. Although the naming system could impose some ordering, e.g., based on a ranking of the person entering the name, this is probably not a good idea.

Query operations do not return until they have recent enough information. Clients must define what is recent enough; there must be something in the application that makes this possible. In addition, clients must be able to act safely on information that is out of date, which is true if the property of interest to clients is stable. The stable property for the map service is that crash counts never decrease; the meaning of "recent enough" is difficult to explain without going into the details of orphan detection. We will examine these properties for garbage collection later in the paper.

3. Garbage Collection

Programming languages such as Lisp, SmallTalk and CLU use a model of computation in which objects reside in a heap and storage management is automatic. In particular, objects are not deallocated explicitly. Instead, they are garbage collected at some convenient time after they become inaccessible. An interesting way to use a distributed system is to provide direct support for this model. An implication of such an approach is that the heap now has parts residing at all the different nodes.

This section describes a way of doing garbage collection of a distributed heap in an efficient manner. Our method is independent of the particular garbage collection technique in use at the nodes, and in fact different nodes can use different techniques. Nodes do not communicate with one another. Instead information about inter-node references is stored at a highly available, logically centralized *reference* service; nodes communicate with this service periodically, with a result that fewer messages need be communicated. The reference service is constructed using the technique described in the preceding section. Finally, the technique is tolerant of all the faults discussed earlier: node crashes, and lost, duplicated, delayed and out-of-order messages.

Several papers on distributed garbage collection [1], [15], [9], [8] have been published. Our method is an improvement on these techniques; none of them is fault tolerant, and all involve more message passing than ours.

The algorithm preserves the same correctness condition as in a non-distributed heap, namely an object remains in existence until all references to it are gone. This requirement must be satisfied even if the node that contains the only reference to an object is down or inaccessible. The requirement is sensible since we assume a stable heap in which all objects are persistent, i.e., survive crashes.

Our approach uses the fact that an inaccessible object stays inaccessible, and so it is not necessary to recognize the inaccessibility of an object immediately. .Our algorithm is "lazy" about reclaiming objects that were referred to at other nodes, but we guarantee that all inaccessible objects are reclaimed eventually.

We assume objects can be referred to uniformly regardless of their location by using a unique name, and that there exists a method for locating objects efficiently given their names. The node at which an object is located is called the *owner* of the object; the object is *local* to that node. We assume that objects do not move; only slight extensions to our algorithm are needed to handle objects that move, but we do not discuss them in this paper. An object is *public* if its owner has sent its name to some other node; once an object is public, it continues to be public even though other nodes no longer refer to it.

A distributed heap can be viewed as a heap whose root is distributed and consists of the union of the roots at all nodes. In such an environment, an object is accessible if it is accessible from one of the roots. In additi[illegible], however, we must consider the special problem of objects that are "in transit." For example, suppose object x is owned by node B, and that node A has a reference to x. Suppose A sends the reference to node C, deletes its own reference (perhaps after having received an acknowledgement from C), and performs a garbage collection. If a garbage collection were done at B using information from A after its garbage collection and information from C before it received the reference to x, x could be destroyed by mistake. This example reflects a case in which x is not accessible from any other object in the heap but a reference to it is in a message that is in transit. To handle in-transit references correctly, we define accessibility as follows: An object is *accessible* if either it is accessible from one of the local roots or a reference to it is in transit.

As was the case in the preceding section, we assume a bound δ on message delay. Messages containing external references also contain the time at which they were sent. If a message arrives at a node when its time $+ \delta + \epsilon$ is less than the time of the node's local clock, it is discarded. This assumption allows us to bound the time the references remain in transit.

The remainder of this section describes our garbage collection method. In the presentation, we descibe the processing and stored information without considering the many optimizations that would be done in a real implementation. We begin by describing the processing at the nodes. Then we describe the interface to the highly-available service, and how the nodes interact with the service. Next, we give an implementation of the service using the replication technique discussed in the preceding section. We conclude with a discussion of the performance of our algorithm.

3.1. Information at nodes

Our strategy for garbage collection is to have the nodes do as much work as possible. Nodes are responsible for doing local garbage collection and managing the storage for local objects that are not public. The reference service is responsible for determining the accessibility of public objects. Nodes must provide it with enough information so that it can do this correctly. Nodes must ask the service about the accessibility of their local objects that are public and must not destroy them until informed by the service that they are inaccessible.

Each node maintains a list, called the *inlist*, of its public, local objects. The *inlist* is kept on stable storage so that it is recoverable should the node crash. An entry is added to the list the first time a reference to a local object is sent to another node. The objects listed in the *inlist* may be a superset of those actually used by other nodes. For example, suppose that object x in node B contains the only reference to object y in node A, and that x is deleted at B. Object y is now inaccessible, yet it has an entry in A's *inlist*.

A node also maintains *trans*, a list of the references it has sent in messages. This list enables us to avoid the mistake of regarding an object as inaccessible while a reference to it is in transit. An entry is added to the list whenever a reference is sent in a message. The list must be recoverable and therefore each entry is written to stable storage before the message containing the reference is sent. Each eniry in *trans* is a triple

<obj-ref, target node, time>

where *time* is the local time stored in the message containing *obj-ref*. For example, if *trans* contains <(x, A, t_1), (y, B, t_2)>, then a reference to object x was sent to node A at time t_1 and a reference to object y was sent to node B at time t_2.

To determine accessibility of public objects correctly, the reference service needs to know *trans* for each node. It also needs to know which public objects are accessible from each node. Objects are accessible if there is a path to them from the root of some node. Each node knows about the objects accessible from its root, but not about objects accessible from other nodes' roots. However, it does know about its *inlist* and this is a superset of its local objects that may be accessible from some other root. Thus a node treats objects as accessible if they are accessible either from its root or its *inlist*.

An example is given in figure 2. Here x, y, z and w are public objects belonging to node A and u and v are public objects belonging to node B. Objects x and u are accessible from A's root; y, z, w, and v are accessible from its *inlist*. No objects are accessible from B's root, but u and v are accessible from its *inlist*. The only inaccessible object in the figure is w; objects y, z, u, and v are globally accessible even though they are not accessible from the roots of their owners.

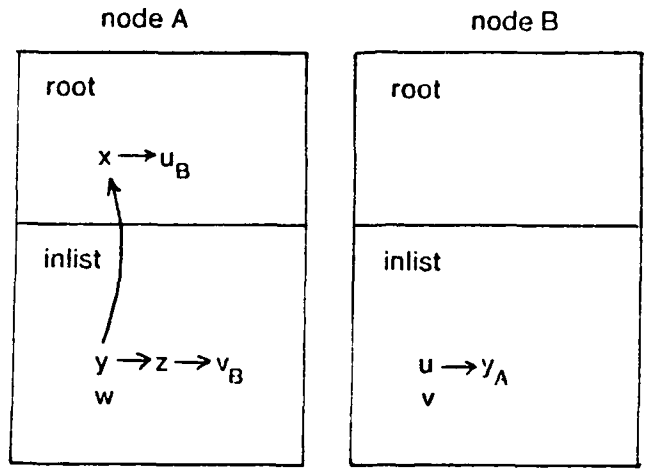

Figure 2: An example of global accessibility.

To inform the service about local accessibility, a node sends two lists, acc and paths. *Acc* lists all objects accessible from its root. We omit local public objects (such as x) from *acc*; these are not needed since their owner will not inquire about their accessibility. *Paths* contains information about what public objects are accessible from locally inaccessible objects in the *inlist*. It is a list of pairs <o, p>, where o is in the *inlist* and is not accessible from the local root, and p is a public object accessible from o. We omit pairs whose second element is local and accessible from the root; these are not needed, again, because their owner will not inquire about their accessibility. Thus the information for node A is:

acc = { u }
paths = { <y, z>, <z, v> }

Note that we do not include pairs in *paths* where the accessibility can be deduced from information already in *paths*; e.g., <y, v> is not included.

The node computes *acc* and *paths* during garbage collection. The lists need not be remembered between consecutive rounds of garbage collection.

The node also maintains a multipart timestamp, which it uses to communicate with the reference service. The timestamp is used to guarantee that the service bases its decision about accessibility of the node's objects on information that is recent enough, as discussed further below. The timestamp is kept on stable storage, so it is remembered when the node recovers from a crash.

Each node does garbage collection independently of the rest of the nodes, using an algorithm of its choice. The algorithm must be extended slightly to take account of the *inlist* and to compute the *acc* and *paths*. It must also construct the *qlist*, the list of objects whose accessibility is questionable. This contains all public, local objects that are not accessible from the node's root, e.g., y, z and w for node A in figure 2.

As an example, we will show the extension of the real-time garbage collector proposed by Baker [2]. This is a real-time copying garbage collector that moves some objects from old space to new space each time it does a CONS (thus creating a new object). When all objects have been copies, it flips the spaces.

The extended version of the algorithm is:

1. Construct an empty *acc*, *paths* and *qlist*.

2. Add to *acc* all references to public, nonlocal objects found in variables or in objects that are moved to new space. This includes both objects that are copied from old space and newly created objects, such as those created when a message is processed.

3. When all accessible objects have been moved to new space, scan the *inlist*. For each object x in the *inlist* that was not moved, do the following:

   a. Add x to the *qlist*.

   b. Move x to new space, and also move to new space all objects reachable from x that are not already in new space and are not in the *inlist*.

   c. For each reference to a public object y found in step b, add <x, y> to the *paths* if either y is nonlocal or y is local but is not accessible from the root.

4. Record local time. This is the *gc time*.

5. Flip the spaces.

Note that the *acc* and *paths* contain a superset of the external references that exist in new space just before the space flip.

The only non-real time processing is due to the scan of the *inlist* in step 3 of the algorithm. If the *inlist* is long and we want to maintain a constant bound on the time for CONS we must allow incremental scanning of the *inlist*. In this case we must add to *acc* any references to public, nonlocal objects found in objects created in new space while the *inlist* is being processed.

The information obtained from running the garbage collector is then conveyed to the reference service as discussed below.

3.2. The Reference Service

The reference service provides two operations, *info* and *query*; a specification is given in figure 3. A node calls *info* when garbage collection is complete to inform the service about its external references and also about objects that are in transit from it to other nodes. One of the arguments to *info* is the *gc-time*; this informs the service about the local time at which *acc* and *paths* were computed, and is used to discard old information about objects in transit to the node. The service records the information in the arguments and responds with the timestamp of a state that contains the information. When the operation returns, the node replaces its timestamp with the returned timestamp and records the result on stable storage. Then it discards *acc* and *paths*, and also the part of *trans* that was passed to the call. (Additional messages may have been recorded in *trans* since the call was made; these new entries are retained.)

Later, the node can call the *query* operation to inquire about which objects in the *qlist* are not accessible elsewhere. When the call returns, the node deletes any inaccessible objects from the *inlist*; this will cause their storage to be reclaimed at the next local garbage collection. The names of these inaccessible objects are recorded on stable storage so that after a crash the *inlist* is recovered accurately. The node can either discard the *qlist*, or retain it (minus the objects known to be inaccessible) for use in further calls of *query* done before the next garbage collection.

One of the arguments to the call of *query* is the timestamp of the node. This argument is used to ensure that the most recent call of *info* is used in processing the query. This is necessary because the *qlist* is defined in conjunction with *paths* of that call of *info* and could be misinterpreted otherwise.

reference = **data type is** info, query

Overview

The reference service maintains a global picture of the inter-node by calling references. Nodes inform it about accessibility periodically the *info* operation. Nodes request information about which local objects can be deleted by calling the *query* operation. The service consists of a set of states σ. Each state records information about the accessibility of public objects and in-transit references for each node. Each state has an associated (multipart) timestamp, and states having larger timestamps record information at least as recent as that in states with smaller timestamps.

Operations

info = **proc** (o: acc, n: paths, t: trans, gc-time: time, ts: timestamp) **returns** (timestamp)
modifies σ
effects Returns a timestamp $\geq$ ts of a state s that records the acc, paths, trans and gc-time for the node. Adds s and its associated timestamp to σ if they are not already there. May create several such states s; their timestamps are generated so as to satisfy the invariant. Information about other nodes is taken from some other state $s' \in \sigma$.

query = **proc** (q: qlist, ts: timestamp) **returns** (list)
effects Computes the accessibility of objects in q based on a state whose timestamp is $\geq$ ts. Returns the list of objects in q found to be inaccessible.

end

Figure 3: Specification of reference service.

The reference service could provide additional operations. Since very often a call of *info* is followed by a call of *query*, a combined operation would be convenient. An operation that would inform the service just about *trans* would permit the node to discard this information more frequently. In what follows, however, we ignore such additional operations.

3.3. Implementation of the Reference Service

This section describes the implementation of the reference service using the replication technique of section 2. Our discussion ignores the processing of cycles; cycle detection is discussed in section 3.4.

The reference service should be highly available so that inaccessible public objects can be collected in a timely fashion. Therefore, its implementation must consists of several replicas. Since each node communicates with the reference service only in background mode, fast response time is not a strong requirement, and a voting scheme could be used to interpret the replicas consistently. However, the nodes can easily satisfy the constraints on clients imposed by the replication scheme described in the preceding section. Each *query* operation executed by a node must have access to information provided by all previous *info* operations of the node; this requirement defines the notion of "recent enough" mentioned in section 2.5. Also, the property needed by the nodes is stable: once an object becomes inaccessible, it will continue to be inaccessible forever. Since our replication scheme performs better than voting, in this section we will describe how to use it to implement the reference service.

Each replica maintains a multipart timestamp and the state that corresponds to that timestamp. The state consists of a complete description of all the inter-node references, including references that are in transit. The state maps each node k to a tuple

<gc-time, acc, paths, to-list>

where *gc-time* is the time of k's garbage collection, *acc* and *paths* are its *acc* and *paths* computed at that time, and the *to-list* is a list of references that were sent to k and may still be in transit. The *to list* is the list of pairs <x, t>, where t is the time at which a reference to x was sent to k. Initially the *gc-time* is zero and the other parts of *state*(k) are empty.

Each replica also maintains a second multipart timestamp, *max ts*, which reflects the latest timestamp produced as a result of processing an *info* message. *Max ts* is needed to detect that information at the replica is incomplete.

The execution at the replica is presented by describing the processing of the *info* and *query* messages it receives and the gossip messages exchanged. As was the case earlier, the calls of service operations cause corresponding messages to be sent to one or more replicas. The arguments in these messages are those of the associated operations plus one additional one, the identity of the node that made the call. To clarify the presentation, we will prefix the names of the arguments with "msg", e.g., *msg.acc*.

The processing of an *info*(*acc*, *paths*, *trans*, *gc time*, *ts*, *i*) proceeds as follows:

1. Discard an old message:
   {If *msg.gc-time* < *state*(*i*).*gc-time* then go to 5.

2. Replace *state*(*i*).*acc*, *state*(*i*).*paths* and *state*(*i*).*gc-time* by *msg.acc*, *msg.paths* and *msg.gc-time*, respectively.

3. Find all the objects in the node's *trans* that should have arrived at the node and delete the entries for which time is up.

```
for all <x, t> ∈ state(i).to-list
   if $t + \epsilon + \delta <$ msg.gc-time
      then delete <x, t> from state(i).to-list
      else do nothing
```

4. Update the replica's database to reflect all the references that are in transit from i. For each object in transit, keep the latest sending time.

```
for all <x, j, t> ∈ msg.trans
    if $t + \epsilon + \delta > $ state(j.gc-time
        then if ∃ <x, t'> ∈ state(j).to-list
            then if t' < t
                then replace <x, t'> by <x, t>
                else do nothing
            else add <x, t> to state(j).to-list
        else do nothing
```

5. end

Note in steps 3 and 4 that we make use of the bound on message delay. Since a node is guaranteed to discard messages that come in too late, it is safe to discard information about in-transit references if the node's *gc time* is later than the bound + ϵ.

When processing is complete, a new multipart timestamp is generated for the replica (unless the message was rejected in step 1). Then the replica's timestamp is merged with the caller's timestamp; the result of the merge is sent back to the caller in the reply message and also is merged into *max ts*.

To handle a *query(qlist, timestamp, i)* message the replica checks each object in the *qlist* to see if the object is accessible from any node. An object x is accessible if a reference to it exists in *acc*, *paths* or *to-list* for some node j.

To process a query, a replica need not have the most recent *info* information for each node but it must have access to a complete sequence of *info* operations for each node. This requirement prevents in-transit objects being treated as inaccessible by ensuring that the replica has information about all objects in transit from a node as of the gc-time stored for it. We fulfil this requirement by waiting until the replica's timestamp is equal to its *max ts*; the replica gossips with other replicas to obtain more information if necessary.

In addition, a *query* cannot be processed until the replica's timestamp is greater than or equal to the timestamp in the message. This condition guarantees that the replica has recent enough information for the node doing the query. Once the condition is satisfied, processing proceeds as follows:

```
create empty alist
  for each x in the msg.qlist
    for each node j
      if x ∈ state(j).acc
           ∨ ∃ <x, t> ∈ state(j).to-list
           ∨ ∃ <y, x> ∈ state(j).paths
         then % x is accessible
           add object x to the alist
           exit inner loop
    end
  end
```

When processing is complete, the list of inaccessible objects, *qlist* – *alist*, is computed and sent back to the caller in a reply message.

In addition to *info* and *query* messages, the replicas must also send and process gossip messages. The sending replica's timestamp and its *max-ts* are included in the gossip message; they are merged with the receiver's corresponding timestamps as part of processing the message. Gossip messages could either contain the entire state of the replica or a sequence of *info* messages. In the latter case, which we assume in the paper, the sequence must include all *info* messages known to the sender that may not be known to the receiver. To bound the sequence, we can associate with each *info* message the timestamp generated when it was first processed. A timestamp table like that in section 2.3 can then be used to decide when an *info* message is known at another replica. Recall that this table stores for each replica the largest timestamp received in a gossip message from it, and that at any time, the timestamp of a replica is larger than that stored for it in the table. Thus a replica must know about an *info* message when its timestamp in the table is larger than the *info* message's timestamp. An *info* message can be discarded when all replicas know about it.

Processing a gossip message consists of processing the sequence of *info* messages. Some of these messages may be old, i.e., *msg.gc-time* < *state(i).gc-time*, where *i* is the sender of the message. The only interesting part of an old *info* message is *trans*; all other information in the message has already been superceded by more recent *info* messages. Therefore, only *trans* is processed for old *info* messages, i.e., only step 4 of the *info* processing algorithm is executed for them.

3.4. Multiple Node Cycles

If the inaccessible inter-node references form a tree, then when the node containing the root of the tree is garbage collected, the rest of the tree will eventually be deleted. But if the inaccessible inter-node references form a cycle, inaccessible objects will never be deleted in our scheme. For example, suppose object x at node A has a reference to object y at B and y has a reference to object x. The inter-node references for x and y form a cycle that spans node boundaries. Even if x and y are both locally inaccessible, they appear to be globally accessible and therefore are not reclaimed by the local garbage collectors at their nodes. They are also not recognized as inaccessible by the algorithm presented in the preceding section.

One way of deleting cyclic objects is to follow the idea suggested in [3] of moving an object that is inaccessible locally to one of the nodes that has a reference to it. Moving the object converts a inter-node cycle to a intra-node cycle whose storage is reclaimed as soon as the node is garbage collected. However, this approach may not be suitable for a distributed system. Moving an object can be expensive. In addition the code that manipulates the object (e.g., the code implementing the operations of its data type) may not be present at the new node. and we may not wish to place the code there. either because the nodes are not homogeneous, or because having all code at all nodes consumes too much space.

Therefore we will use another method in which the central service detects inaccessible cyclic objects using its global view of the external references. Each replica runs a cycle-detection algorithm periodically. The replica does this somewhat independently of other replicas. When cycle detection is finished, the replica conveys the information about inaccessible objects to other replicas by gossip messages.

As was the case with queries, to run cycle detection a replica need not have the most recent *info* information for each node, but it must have access to a complete sequence of *info* operations for each node. Again we fulfil this requirement by waiting until the replica's timestamp is equal to its *max ts*; the replica gossips with other replicas to obtain more information if necessary.

The algorithm consists of a *mark* phase in which all objects that are definitely accessible are marked, and a *sweep* phase in which inaccessible objects are identified. The *mark* phase consists of the following:

1. Mark all the objects that occur in *acc* or *to list* of some node.

2. Mark x if there exists an object y such that y is marked and <y, x> ∈ *paths* of some node.

The *sweep* phase then identifies the inaccessible objects as follows:

```
for each node i
  for each <x, y> ∈ state(i).paths
    if x is unmarked then flag the pair
  end
end
```

In processing a *query*, we ignore tagged pairs. The reason we flag the pair in *paths* instead of deleting it is to insure that the results of the cycle reclamation algorithm persist. We retain information about inaccessible objects gained in this fashion until the owners of the inaccessible objects are known to know about them. Until that time, the owner of an object may send an *info* message that contains information about the object in *paths*, and we must avoid reintroducing that information into the replica's state. As soon as an *info* message arrives that does not contained the flagged pair, we know the owner knows about the inaccessibility. At this point we remove the pair from *paths*.

4. Discussion

The distributed garbage collection algorithm discussed above allows garbage to be collected in a timely and efficient manner. It has several performance advantages. First, it propagates information quickly and with few messages. A node merely informs a single replica; the replica then informs all the others. Assuming that *info* messages are propagated as gossip as soon as they arrive, this implies 2 + n messages are sent, where n is the number of replicas. Once these messages are sent, all nodes' queries can be processed using the new information. Thus the information from one node is available to any other node in 4 + n messages. Such propagation is better than what could be done in a system where nodes communicate directly, assuming the number of nodes is larger than the number of replicas.

The result of fast propagation of information is timely deletion of inaccessible objects. In systems where nodes communicate directly (e.g., [1], [8], [9], [15]), either all nodes must communicate to decide about inaccessibility, or a node must communicate with those other nodes that have references to its objects. In either case, if any of these nodes in down or inaccessible, garbage collection cannot proceed. In our scheme, progress can be made as long as the replicas are available. At worst, this means a smaller number of sites need be up. More likely, however, progress can be made even when some replicas are down because information is replicated.

Another advantage is that the method off-loads the work of garbage collection from the nodes. Even though most of the work of garbage collection is done in background, the amount of work is substantial. By having this work done at the replicas, we free the nodes to do their real job, the processing of user computations.

Finally, the method has the merit that it usually does not cause delays in user computation. The one exception is that nodes must log information about an in-transit reference on stable storage before the message containing the reference can be sent. Writing to stable storage is not really necessary, but it greatly speeds up global garbage collection after a crash. Not doing such logging results in the loss of the *inlist* and *trans* after a crash. Loss of the *inlist* means that all the node's objects must be considered to be public. Loss of *trans* means that we must assume the node has sent messages containing references to all objects it knows about to all other nodes. The service must make worst case assumptions about the objects the node might know: It is assumed to know about all public objects it knew about at its last garbage collection plus any objects sent to it that might have arrived after that. In addition it knows about all (local) objects it ever created.

Ultimately, the truth of all these assumptions will be known, but we must wait until every other node has communicated with the central server with a *gc-time* $> t + \delta + \epsilon$, where t is the time of the crashed node's local clock just after it recovers from the crash. (This condition assumes the crash is instantaneous and that the node sent a reference in a message to some other node just before it crashed.) This wait can be long. A special problem is crashes of other nodes, since that prevents them from communicating with the reference service. If there are a large number of nodes, the probability of a crash of one of them during this period can be non-negligible.

There are two ways of avoiding this wait without incurring the real-time delay of logging entries in *trans*. The first is to log the identities of other nodes that a node communicates with. This information limits the number of nodes that must be heard from after a crash before an accurate assessment of the crashed node's *trans* and *inlist* can be made. The approach is helpful only when communication patterns are fairly static and when nodes communicate with just a small number of other nodes. It is not clear how many applications will exhibit such a pattern.

The second approach is more generally promising and will work in any system in which computations are carried out as atomic transactions. In such a case, logging of *trans* need not be completed until the transaction that caused the messages to be sent prepares in the first phase of two-phase commit. Therefore, *trans* can be logged in background mode between the time the message is sent and the prepare; at worst, it can be written to stable storage as part of the prepare record.

The reason transactions allow this optimization is the following. We assume that the crash of any node visited by a transaction causes the transaction to abort. When a transaction aborts, it is as if it never ran, and therefore we need not be concerned about messages it sent. Therefore, we need not log these messages.

In a distributed system with resilient objects, transactions are needed to make sense of computations in the presence of concurrency and failures. It is likely that the distributed garbage collector will be running in a system with transactions.

5. Summary

This paper has described two techniques for use in a distributed system. The first is a way to implement a highly-available service. The technique guarantees that clients see consistent views of the service's database, but requires that they communicate with only one of the replicas implementing the service. It is appropriate for use in applications in which clients can proceed correctly on the basis of information that is out-of-date. Such clients are typically interested in stable properties, properties that once true are always true.

The second technique is a fault-tolerant, distributed garbage collection algorithm. The algorithm is both efficient and fault-tolerant. It allows the nodes containing the distributed heap to do local garbage collection independently, each using an algorithm of its choice. Information about inter-node references is maintained at a highly-available central service. We described an implementation of this central service based on the technique developed in the first part of the paper.

In the preceding section we argued that the garbage collection method had a number of desirable properties: it propagates information quickly, it detects inaccessible objects quickly, it off-loads work from the nodes, and it avoids delaying user computations. These properties are really due to our use of a central service as opposed to having nodes communicate directly. The central service could have been implemented using a different technique such as voting. Our implementation technique, however, has better performance than voting for applications that can use it; as was argued in section 2.4, both response time and availability are improved through its use. Garbage collection is an example of an application that can benefit from the technique.

Acknowledgment

The authors gratefully acknowledge the many helpful comments and suggestions provided by Toby Bloom and Bill Weihl. Numerous other people, including Brian Coan and Elliot Kolodner, also provided critical comments that greatly improved the content and presentation of the paper.

References

[1] Ali, K. A. M. *Object Oriented Storage Management and Garbage Collection in Distributed Processing Systems.* PhD thesis, Royal Institute of Technology, Stockolm, December, 1984.

[2] Baker, H. G. List Processing in Real Time on a Serial Computer. *Comm. of the ACM* 2(4):280-294, April, 1978.

[3] Bishop, P. B. *Computer Systems with a Very Large Address Space and Garbage Collection.* Technical Report MIT/LCS/TR-178, MIT Laboratory for Computer Science, Cambridge, Ma., May, 1977.

[4] Chandy, K. M., and Lamport, L. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems* 3(1):63-75, February, 1985.

[5] Fischer, M. J., and Michael, A. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proc. of the Symposium on Principles of Database Systems.* ACM, March, 1982.

[6] Gifford, D.K. Weighted Voting for Replicated Data. In *Proc. of the Seventh Symposium on Operating Systems Principles*, pages 150-162. ACM, December, 1979.

[7] Birrell, A. D., et al. Grapevine: An Exercise in Distributed Computing. *Comm. of the ACM* 25(4):260-274, April, 1982.

[8] Hudak, P., Keller, R.M. Garbage Collection and Task Deletion in Distributed Applicative Processing Systems. In *Proceeding of the ACM Symposium on Lisp and Functional languages*, pages 168-178. August, 1982.

[9] Hughes, J. A Distributed Garbage Collection Algorthm. In *Functional Languages and Computer Architectures Confrence at Nancy, France*, pages 256-271. 1985.

[10] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM* 21(7):558-565, July, 1978.

[11] Lampson, B. W., and Sturgis, H. E. *Crash Recovery in a Distributed Data Storage System.* Technical Report, Xerox Research Center, Palo Alto, Ca., 1979.

[12] Liskov, B., and Weihl, W. Specifications of Distributed Programs. *Journal of Distributed Computing* , to appear.

[13] Lundelius, J. *Synchronizing Clocks in a Distributed System.* Technical Report MIT/LCS/TR 335, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1984.

[14] Marzullo, K. *Loosely-Coupled Distributed Services: A Distributed Time Service.* PhD thesis, Stanford University, Stanford, Ca., 1983.

[15] Nori, A. K. A Storage Reclamation Scheme for Applicative Multiprocessor Systems. Master's thesis, University of Utah, December, 1979.

[16] Liskov, B. *Overview of the Argus Language and System.* Programming Methodology Group Memo 40, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1984.

[17] Liskov, B., et al. *Preliminary Argus Reference Manual.* Programming Methodology Group Memo 39, M.I.T. Laboratory for Computer Science, Cambridge, Ma., October, 1983.

[18] Schlichting, R. D., and Schneider, F. B. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Trans. on Computing Systems* 1(3):222-238, 1983.

[19] Skeen, D., and Wright, D. D. *Increasing Availability in Partitioned Database Systems.* Technical Report 83-581, Dept. of Computer Science, Cornell University, Ithaca, N. Y., 1984.

[20] Walker, E. W. *Orphan Detection in the Argus System.* Technical Report MIT/LCS/TR 326, M.I.T. Laboratory for Computer Science, Cambridge, Ma., June, 1984.

[21] Weihl, W. *Distributed Version Management for Read-only Actions.* Programming Methodology Group Memo 47, M.I.T Laboratory for Computer Science, Cambridge, Ma., 1986. Submitted for publication.