Collecting Cyclic Distributed Garbage by Controlled Migration

Umesh Maheshwari Barbara Liskov

M.I.T. Laboratory for Computer Science
Cambridge, MA 02139

Abstract

Distributed reference counting provides timely and fault-tolerant garbage collection in large distributed systems, but it fails to collect cyclic garbage distributed across nodes. A common proposal is to migrate all objects on a garbage cycle to a single node, where they can be collected by the local collector. However, existing schemes have practical problems due to unnecessary migration of objects. We present solutions to these problems: our scheme avoids migration of live objects, batches objects to avoid a cascade of migration messages, and short-cuts the migration path to avoid multiple migrations. We use simple estimates to detect objects that are highly likely to be cyclic garbage and to select a node to which such objects are migrated. The scheme has low overhead, and it preserves the decentralized and fault-tolerant nature of distributed reference counting and migration.

1 Introduction

Systems that store objects on multiple nodes need distributed garbage collection to reclaim storage of inaccessible objects. These systems can use either global marking [HK82] or distributed reference counting [Bis77]. Global marking requires the cooperation of all nodes before it can collect any garbage. Distributed reference counting is preferred for systems with large numbers of nodes because it is more fault-tolerant and scalable, and quicker at collecting distributed garbage. Many variants of distributed reference counting schemes have been proposed to enhance fault-tolerance and reduce overheads [Ali84, Bev87, Ves87, Piq91, LL92, SDP92, BENOW93, ML94].

Distributed reference counting algorithms cannot collect multi-node cycles of garbage objects. This is particularly undesirable in long-lived systems such as persistent object stores, where even small amounts of uncollected garbage can accumulate over time to cause a significant storage loss. The problem can be solved either by using a complementary marking scheme to collect cyclic garbage [Ali84, JJ92, LQP92], or by migrating objects so that cyclic garbage ends up in a single node and is collected by the local collector [Bis77, SGP90, GF93]. The advantage of migration is that, like distributed reference counting, it is decentralized and fault-tolerant. The collection of a cycle requires the cooperation of only those nodes that contain it, and progress can be made even if other nodes or other parts of the network fail. Therefore, migration is an attractive solution in large-scale systems that allow objects to migrate between nodes.

However, existing migration schemes have some practical problems. First, they tend to migrate live objects along with garbage. Most schemes migrate locally unreachable objects, either immediately [Bis77, Bag91] or if the objects are not used for some time period [GF93]. In a persistent store, however, live objects may not be accessed for long periods (say, weeks or months), so even systems that wait will migrate live objects. Migration of live objects is undesirable because it wastes processor and network bandwidth. Also, it interferes with load balancing. For example, a database might be partitioned between two nodes to position objects close to frequent users, yet it may have a single logical root, say, on the first node. In that case, all objects on the second node would be locally unreachable, but should not be moved. (Of course, users could prevent this migration by explicitly rooting the second partition locally, but this leads to exactly what garbage collection is supposed to replace: error-prone manual memory management.) The problem with load-balancing is exacerbated because migration must occur in one direction to avoid thrashing, as explained in Section 4.2.

This paper presents a simple way to avoid unnecessary migration. We estimate the length of the shortest path from any root to each object. Estimates for cyclic distributed garbage objects keep increasing without bound; those of other objects do not. We migrate only objects with very large estimates.

We also do the actual migration of objects efficiently. Consolidating a distributed garbage cycle may involve migrating objects multiple times before they converge on the same node. Some schemes avoid this problem by migrating objects to a fixed dump node [GF93], but having a single dump node in a large system can be a performance or fault-tolerance bottleneck. We have a simple way of selecting one of the nodes containing a garbage cycle as the destination, and we migrate all objects on the cycle directly to that node. Further, migrating an object often leaves behind objects that have to be migrated later; our scheme determines all objects at a node that should be moved together, and batches them in a single message.

Our scheme is based on a fault-tolerant variant of reference counting described in Section 2, and adds very little space and time overhead to the base scheme. Unlike some other schemes [BE86, KA93], ours does not require the local collector to trace the object graph multiple times. It also preserves the naturally decen-

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 N00014-91-J-4136.

PODC 95 Ottawa Ontario CA © 1995 ACM 0-89791-710-3/95/08..$3.50

tralized and fault-tolerant nature of distributed reference counting and migration.

The scheme achieves its desirable performance properties by delaying the collection of cyclic garbage: it waits to migrate objects until they have a large distance estimate, and until the destination node has been selected, thus avoiding the cost of unnecessary migrations that occur in other schemes. We believe that slowness in collecting distributed cyclic garbage is not a practical problem because we expect cyclic garbage to be a small fraction of the total garbage. Thus, our scheme makes an appropriate tradeoff: cyclic garbage is always collected, but in a way that does not degrade overall system performance.

The rest of the paper is organized as follows. Section 2 describes the environment in which our technique is to be used and how reference counting works in that environment. Section 3 describes the distance heuristic used to recognize cyclic garbage. Section 4 discusses how we migrate objects efficiently. Section 5 surveys related work in collection of distributed cyclic garbage, and Section 6 contains our conclusions.

2 The Problem Context

Our algorithm is designed for use in the Thor object-oriented database system [LDS92], although it is applicable to a wide range of similar distributed systems. Thor stores persistent objects at geographically distributed nodes. At any time, an object resides at one node, although it can be migrated to another node. Objects contain references to other objects, which may reside at any node. For efficient access, a reference to an object contains the identity of the node where the object resides [DLMM93]. Objects are clustered within nodes so that internode (remote) object references are rarer than intra-node (local) references.

Persistence of objects is determined by reachability from the persistent root objects, which may be on any node. An object y is *reachable* from x if y is x or if x contains a reference to z and y is reachable from z. We also say that an object y is *locally reachable* from x if y and x are on the same node and y is reachable from x through only local references.

An object that is not reachable from any persistent root is garbage. (In this paper, we ignore transient roots such as stack variables.) If two garbage objects on different nodes are reachable from each other, they are on a multi-node garbage cycle and are said to be *cyclic distributed garbage*. In general, a number of garbage cycles may be reachable from each other, thus forming a compound cycle, and further, there may be non-cyclic chains of references incident on or outgoing from a cycle. In our scheme, garbage objects on chains outgoing from a garbage cycle are treated like objects on garbage cycles. Therefore, we shall often not distinguish between the two.

In distributed reference counting schemes, each node does a local collection independent of other nodes. The local collection is based on tracing from a root set that includes the local persistent root objects as well as local objects that are referenced from other nodes, called the *secondary roots*. Different variants of distributed reference counting employ different methods and information to track the secondary roots, ranging from one-bit counts [Ali84, JJ92] to weighted reference counts [Bev87] to reference lists, in which each node tracks the identities of the nodes that refer to its objects [Bis77, SDP92, BENOW93].

We use reference listing because it handles catastrophic node failures and provides better fault-tolerance for messages. (For a full description see [ML94].) Our scheme works as follows:

1. A node N_1 keeps, for every other node N_2, a list of objects in N_1 that N_2 may hold references to. We call the list the *inlist* for N_2 at N_1.
2. When a new internode reference is created from node N_2 to an object x at node N_1, N_1 is told to enter a reference to x in its inlist for N_2.
3. The local collector uses inlist entries as secondary roots. As it traces, it records all references to remote objects that it encounters. At the end of collection at node N_2, the list of reachable references to objects in another node N_1 is sent to N_1. We call this the *outlist* for N_1 at N_2.
4. When N_1 receives an outlist from N_2, it uses it to replace its inlist for N_2. This serves to remove unnecessary entries from the inlists.

3 Distance Heuristic

This section describes how we recognize distributed cyclic garbage. Our approach is based on estimating the "distances" of objects:

> The *distance* of an object is the minimum number of internode references in any path from a persistent root to that object. The distance of an object unreachable from the persistent roots is infinity.

Figure 1 illustrates the notion of distance. Object r is a persistent root and therefore has zero distance; so does z since it is locally reachable from a persistent root. Object s is reachable from r through two paths: one with two internode references and another with one; its distance is therefore one. Objects x and y are garbage and have infinite distance. Note that even the distance of a live object is theoretically unbounded: it can be more than the number of nodes in the system, since references can go back and forth between nodes.

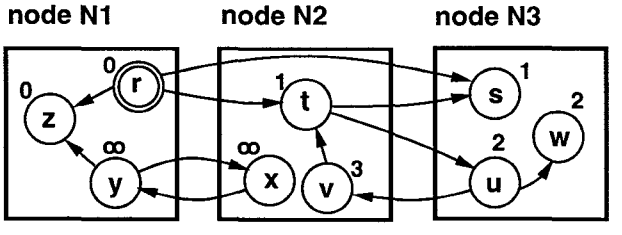

Figure 1: Actual distances of objects.

3.1 Estimating Distances

We maintain estimates of object distances as follows. A distance is associated with each reference in the root set. There may be multiple references in the root set to the same object, each with a different distance. The estimated distance of an object is the minimum distance of any reference in the root set it is reachable from.

The distance of a persistent root is implicitly zero, and each reference in an inlist has an associated distance field. When an inlist entry is created because of a new internode reference, its distance is initialized to one for want of better information.

The local collector propagates distances from roots to outlists, setting the distance of an outlist entry to one plus the minimum distance of any root it is reachable from. It accomplishes this by tracing from the roots in the increasing order of their distances and tracing completely from one root before going on to the next. This does not necessitate depth-first traversal; a copying collector can be used as long as one root is traced completely before the next untraced root is copied. As with other practical schemes, once an object has been traced, it is not traced again. When an outlist entry is created, its distance is set to one plus the distance of the root being traced. This technique is similar to timestamp propagation in [Hug85], which is described in Section 5.

As before, outlists are sent to other nodes after a collection. When a node receives an outlist, it uses it to replace its inlist for the sender, including the associated distance fields.

This scheme has little space and time overhead. Distance fields are only associated with inlist and outlist entries, not with all objects, and require only a few bits (a one-byte field can account for chains with 255 internode references). No extra messages are required over those already sent for collecting non-cyclic distributed garbage, although the outlist messages are a little larger. The inlists can be traced in distance order efficiently as follows. Each inlist is kept sorted by distance and the local collector simply merges them on the fly. Note that the outlists are generated in distance order as well. Since inlists are updated using outlists, the inlists don't need to be sorted explicitly.

The scheme does depend on the use of inlists rather than reference counts. Inlists allow us to maintain different distances for the same object — one for each node that references it. If the minimum-distance entry is removed, the next smallest becomes effective automatically. For instance, in Figure 1, N_3 contains two inlist entries for s: in the inlist for N_1 with distance one and in the inlist for N_2 with distance two. If the reference from r to s is deleted, N_3 will eventually receive an updated outlist from N_1, remove the corresponding inlist entry, with the result that the estimated distance of s will change to two. With reference counts, it would not be possible for N_3 to update the distance without querying other nodes.

3.2 Distance of Cyclic Garbage

Propagation through local collections and outlist messages causes the distances of cyclic garbage to increase without bound. Intuitively, this happens because each local collection increments the estimates as it propagates them from inlists to outlists, and there is no persistent root to hold down this increase. This section quantifies the rate of increase of distance estimates for cyclic garbage.

Nodes do local collections at different times and different rates. For simplicity of analysis, assume that nodes do at least one local collection in a certain period of time, called a *round*. In each round, nodes update inlists using the outlists received in the previous round, do a local collection, and send new outlists to other nodes. (Section 3.4 discusses the implications of slow nodes.)

First consider a simple example. Figure 2 shows a cycle of C internode references that is "rooted" at some object s, which is reachable from a persistent root r through a chain of D internode references. Thus, the distance of s is D and that of each successive object in the cycle is one higher, such that the distance of the last object t in the cycle is $D + C - 1$.

The cycle turns into garbage when a reference in the chain from r to s is deleted. Then, the chain of references leading to s will be removed in at most D rounds through regular distributed collection. When this happens, the estimated distance of s jumps from D to the next best alternative, $D + C$, due to the reference from t. The increase in the distance of s starts a wave of increased distances down the cycle and, C rounds later, the wave reaches s again, increasing its distance to $D + 2C$. Similarly, after every C rounds, the distances of objects on the cycle increase by C. Thus, the distances of all objects on the cycle will cross any given value, T, in about T rounds.

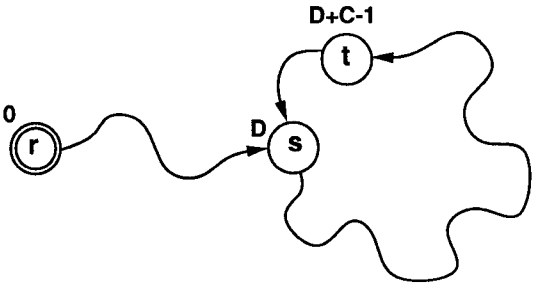

Figure 2: Distances in a cycle of references.

In fact, the following theorem holds for arbitrary graphs of garbage objects, including compound cycles with incident and outgoing chains.

Theorem 1

J rounds after an object became garbage, the estimated distance of the object will be at least J if the object is not collected by then.

Proof (by induction)

Consider an object x that became garbage at some point. At that time, the set of all objects that x is still reachable from must be garbage as well. Further, since the mutator does not create new references to garbage objects, this set cannot grow.

The theorem holds trivially when J is zero.

Suppose the theorem holds when J is K. Then, after K rounds, the distance estimates for x and all objects it is reachable from must be at least K. Thus, all outlist entries generated by tracing through any of these objects must have a distance of at least $K + 1$.

In round $K + 1$, the outlists from round K are used to update the corresponding inlists. If x is not reachable from a local inlist entry, it will be collected by the local collector, since x is also not reachable from any persistent root. Otherwise, the estimated distance of x will be the minimum distance of any inlist entry it is reachable from. But the distances of all such entries must be at least $K + 1$. Thus the theorem holds when J is $K + 1$.

□

3.3 The Threshold

While Theorem 1 holds for all garbage objects, any non-cyclic garbage is duly collected by distributed reference counting. Thus, the distance estimates for cyclic garbage objects increase indefinitely, while those of other objects do not. Therefore it is possible to select a *threshold* distance, T, such that all objects with a greater distance are highly likely to be cyclic garbage. Only those objects are migrated.

The choice of the threshold depends on the expected distances

of live objects. However, estimated distances of live objects may deviate temporarily from their actual distances. Figure 3 shows a somewhat contrived scenario where the deviation may be significant. Object r is a persistent root; s is reachable from r via 10 internode references; t is reachable from s along two paths: one with one internode reference, and another with 10 internode references. The distance of t is thus 11. Now suppose the mutator creates a reference from r to s and then removes the direct reference from s to t, so that the actual distance of t remains 11. When N_3 learns that the reference from s to t has been deleted, the estimated distance to t jumps up to 20, which would have been the old distance of t if the reference from s to t were absent. Only after the information propagates on the path from s to t, is the estimated distance of t reduced to 11.

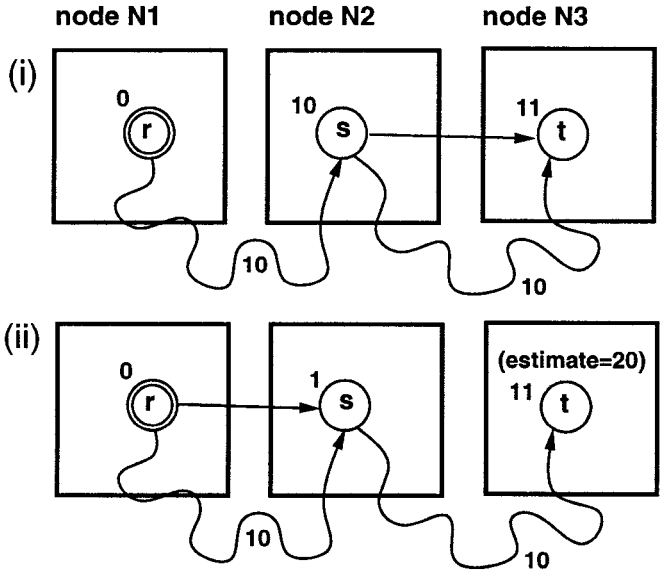

Figure 3: Deviation in the estimates.

Although the deviation of estimates from actual distances cannot be bounded, it is reasonable to expect that a single chain is unlikely to have many changes of the sort illustrated in Figure 3 occurring at the same time. Thus the threshold can be chosen to be only a small multiple of the expected maximum distance. For example, if the expected maximum distance is 10, it is reasonable to set the threshold to 30.

Setting the threshold involves a tradeoff: The threshold should be high enough that non-cyclic garbage is unlikely to be migrated, but a low threshold will collect cyclic garbage faster. Fortunately, the penalty on misjudging the threshold is not severe. If the threshold is too low and live objects with larger distances are migrated, safety is not compromised since the objects will be deleted only if they are actually unreachable from the roots. If the threshold is too high, Theorem 1 still guarantees that all cyclic garbage will be detected eventually.

3.4 Fault Tolerance

Distance propagation has the locality and fault-tolerance of distributed reference counting: the detection of distributed garbage needs the cooperation of only the nodes that the garbage is reachable from. Stated differently, if a node is crashed, partitioned from others, or otherwise slow in doing local collection, it will hinder the collection of only the garbage that is reachable from its objects. The scheme does not require any global mechanism: it makes progress through decentralized, pair-wise communication between nodes.

When a node N_1 is uncooperative, the inlists for N_1 at other nodes are not updated. This is safe because the estimated distances of the objects reachable from such an inlist entry will not increase above the distance of that entry. An uncooperative node might delay the recognition of garbage, but that appears to be unavoidable: if some garbage is reachable from N_1, then N_1 must play its role in the detection of that garbage.

4 Migration

This section discusses practical issues that arise in consolidating a garbage cycle on a single node. We assume the system already possesses a mechanism for migrating objects and updating the references to them in other objects [SGP90, DLMM93]. We discuss how to batch objects for migration, and how to determine where to send the migrating objects. The emphasis is on reducing the number of object migrations because of their processing cost: sending messages and updating references.

4.1 Batching Objects

Migrating a remotely referenced object that contains references to local objects is likely to create more remotely referenced objects that may have to be migrated later. However, not all objects reachable from the migrating object should be migrated with it. For example, when object x in Figure 4 is migrated, we ought to also migrate y but not z.

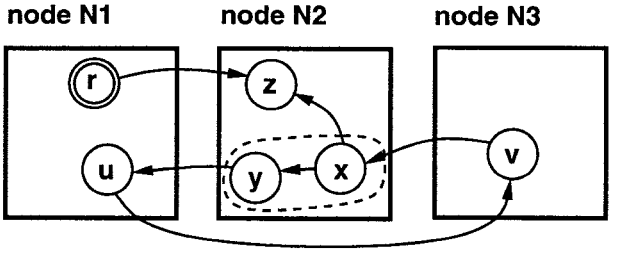

Figure 4: Batching objects to be migrated.

Tracing inlist entries in the increasing order of distances makes it simple to select objects that should be migrated together. Once the distance of the root being traced is above the threshold, any object traced thereafter is likely to be garbage. Then, all objects traced from the same root reference are migrated together. In Figure 4, z will be traced from the root r, and x and y will be batched together.

4.2 Where to Migrate

The goal is to migrate all objects on a garbage cycle to a single node. Some schemes migrate objects to a fixed dump node [GF93], but this can be a performance or fault-tolerance bottleneck in a large system. The dump node might be far away from the nodes containing the garbage cycle, or it might be unavailable when it is time to migrate the cycle.

Other schemes migrate objects to nodes that refer to them. To ensure that all objects in a cycle converge on the same node instead of following each other in circles, nodes are totally ordered and migration is allowed only in one direction [SGP90]. However, objects on a multi-node cycle may require multiple migrations before converging on the same destination node. For a simple cycle that spans C nodes, $O(C^2)$ object migrations may be performed: the object closest to the final destination node is migrated once, while the object farthest from it may be migrated up to $C-1$ times. For example, in Figure 4, assuming that migration is allowed in the direction from N_1 to N_2 to N_3, x and y are migrated just once, but u may be migrated twice (unless u is migrated after x and y have been migrated).

We too use an ordering on the nodes: migration is only allowed in the direction of increasing node ids. However, we estimate the destination node (the node with the maximum id) and migrate all objects on the cycle directly to it. To this end, we propagate estimates of the destination node along with distances that are above the threshold, and wait before migrating objects until this information is likely to have propagated around the cycle.

Inlist and outlist entries with distances greater than the threshold have an associated destination field. When the collector creates such an outlist entry, the destination is set to the higher of the following:

1. The id of the node the outlist is for.
2. The destination of the inlist entry it is traced from, or, if the inlist entry does not have a destination field, the local node id.

As before, the outlists received from other nodes are used to update inlists.

To propagate the maximum node id, all inlist entries above the distance threshold are traced in decreasing order of their destination fields. (It is acceptable to not trace them in distance order because they are already likely to be cyclic garbage.) If such entries were traced in distance order, the maximum node id might not propagate around the cycle. For instance, if there are multiple inlist entries to an object, the one with the largest destination may be blocked by another with a lower distance. This situation is illustrated in Figure 5, which shows a compound garbage cycle; all objects in the figure are on different nodes. Here, s has two inlist entries: from N_1 with distance D_1+1 and from N_2 with distance D_2+1. If $N_1 < N_2$ and $D_1 < D_2$, N_2 would not propagate beyond s if we propagate in distance order. This would result in multiple migrations: initially, some objects would migrate to N_1 and others to N_2, and then the objects migrated to N_1 would migrate to N_2. Tracing in destination order ensures that the maximum id will eventually propagate to other nodes, even in compound cycles, so that objects do not have to be migrated multiple times.

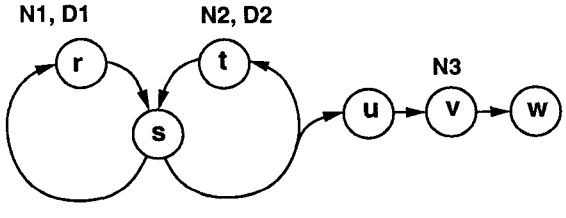

Figure 5: Destination propagation in a compound cycle.

Migration is not necessary for objects that are reachable from a distributed garbage cycle but are not part of the cycle: if these objects did not migrate, they could still be collected through non-cyclic collection after the cycle has been collected. Our scheme does not prevent the migration of such objects, but it does avoid migrating them multiple times. It may migrate them to different nodes, however. For instance, consider Figure 5, where a garbage chain passes through a node N_3 that is higher than the highest node N_2 on the cycle proper. The cycle and the front of the chain (object u) will migrate to N_2, while the trailing part of the chain (object w) would migrate to N_3. The objects will not migrate further, however, so multiple migrations are still avoided.

Unlike some leader election algorithms [LeL77], ours does not incorporate termination detection, so nodes must guess when destination propagation has completed; we discuss how to make this guess in the next section. The advantage of our scheme is that it is simple and effective even in compound cycles.

4.3 When to Migrate: the Second Threshold

To avoid migrating objects before receiving the final destination information, nodes wait until the distances of inlist entries are above a second threshold, T_2, which is higher than the threshold T used to detect cyclic garbage. Setting the second threshold, T_2, involves a tradeoff similar to that for setting T. It should be high enough that by the time the distance of an entry increases to T_2, its destination field is likely to be set to the highest node in the cycle. But it should be low so that cyclic garbage is migrated quickly. In this section we provide an estimate for how high T_2 should be.

First, we quantify the number of rounds destination propagation takes to complete. From Theorem 1, T rounds after a cycle became garbage, the distances of all objects on the cycle will be at least T. Thereafter, all associated inlist entries will be traced in destination order. In M more rounds, the maximum node id on the cycle will propagate to the inlist entries on other nodes, where M is the the maximum length of the shortest path between any two objects — counted in internode references. Thus, destination propagation in a cycle completes in $T+M$ rounds after it became garbage.

However, nodes do not have any knowledge of the number of rounds that have passed; they can only guess it from the distances of inlist entries. The distances of the various entries on the cycle may cross the threshold T at different times. As in the example illustrated in Figure 2, the distances may actually increase in jumps of C, the size of the cycle. Thus, when the entry on the highest node crosses the threshold T, its distance might actually be as high as $T+C-1$. After that, in the M rounds it takes for its id to reach another node, the distances might increase up to $T+C+M$. Thus, an appropriate setting for T_2 is $T+C+M$, where C is the expected maximum cycle size, and M is the expected maximum internode distance between two objects as discussed above.

Using a threshold to guess termination is only a heuristic. If the threshold is reached before destination propagation is complete, objects on a cycle may initially be migrated to different nodes, resulting in multiple migrations. This can happen even when the chosen values of T, C and M are conservative. The following contrived example shows that no fixed setting for threshold T_2 can prevent multiple migrations. Figure 6 shows two cycles, where the right cycle is reachable from the left cycle. Suppose that the left cycle becomes garbage before the right cycle. Further, when the distances of objects on the left cycle are near the second threshold T_2, those on the right cycle have just crossed T. Since the right cycle is now traced in destination order, the maximum node on the

left cycle, N_1, may propagate to some objects on the right cycle. Moreover, the distances of these objects will jump above T_2, causing them to migrate to N_1. If the maximum node on the right cycle, N_2, is higher than N_1, the remaining objects on the right cycle will migrate to N_2. Those that were migrated to N_1 will later be migrated to N_2. If the nodes on the right cycle that migrated objects to N_1 had waited longer, N_2 would have propagated around the cycle, so that all objects would have migrated to N_2 directly.

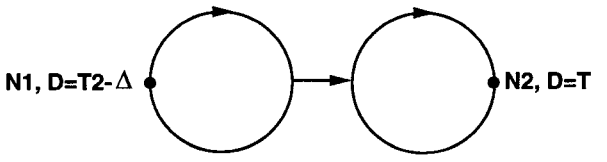

Figure 6: Destination propagation in connected cycles.

Fortunately, using a fixed threshold to guess termination does not compromise correctness or liveness. Destination propagation is used only as an optimization to reduce the number of migrations before the objects converge.

5 Related Work

Distributed reference counting can be augmented in various ways to collect cyclic distributed garbage. Some systems periodically invoke global marking to collect cyclic garbage [Ali84, JJ92]. Lang *et al.* proposed marking within *groups* of nodes such that each round can tolerate failures of nodes outside the group [LQP92]. However, the formation, management, and reconfiguration of groups is still complex and speculative. Ferreira *et al.* group partitions that are cached in memory on the same node, and thus collect any inter-partition cyclic garbage that lies within the group [FS94].

Hughes's algorithm propagates *timestamps* from the roots to the outlists [Hug85]. The timestamps of the persistent roots are advanced to the current clock time before each local collection. The persistent and the secondary roots are then traced in the order of decreasing timestamps. As outlists are exchanged, each node records the minimum timestamp that it has yet to propagate. A global algorithm is used to detect the minimum such timestamp recorded by any node, which is used as a global threshold. Any object whose timestamp is below this threshold is garbage. The timestamping algorithm is the dual of distance propagation: it continually increases the timestamps of live objects while the timestamps of the garbage objects stagnate. If timestamps are not updated, live objects may have old timestamps, but the low threshold guards against their collection. The main weakness of timestamp propagation is that if any node is uncooperative, the global threshold will stop advancing, which will stop garbage collection in the entire system.

Ladin *et al.* proposed the use of a logically centralized service that tracks all inter-node references and uses Hughes's algorithm to collect cyclic garbage [LL92]. The centralized service avoids the need for a distributed algorithm to compute the global threshold. However, collection of cyclic garbage still depends on timely correspondence between the service and all nodes in the system. Further, the centralized service, albeit replicated, can become a bottleneck in a large system.

Beckerle *et al.* proposed that each node send information regarding which outlist entries are reachable from each secondary root to a fixed node in the system [BE86]. The fixed node uses the information to detect unreachable cycles between nodes. This scheme has two problems. First, the fixed node is a bottleneck. Second, to obtain the full set of outlist entries reachable from any secondary root, the local collector must trace from each secondary root independently. Thus, if an object is reachable from multiple secondary roots, it will have to be traced multiple times. If there are m secondary roots, n objects, and e references contained in them, then the performance of local collection with multiple tracing would be $O(m(n+e))$ instead of the usual $O(n+e)$. Note that both distance and timestamp propagation exploit an ordering on the root set to avoid multiple tracings.

Vestal proposed *trial deletion* of objects suspected to be garbage [Ves87]. A separate set of reference counts is used to propagate the effect of trial deletions. If the trial count of a trial-deleted object drops to zero, it confirms that the object was cyclic garbage. The scheme is designed for systems where local collections are based on reference counting and does not extend well to tracing. In particular, it does not work well if different nodes trial-delete objects on the same chain. Even if separate sets of trial counts are maintained for trial deletions started by different nodes, multiple tracings of the object graph from different roots would be required to propagate the information. The scheme suggested in [KA93] uses *probes* to confirm the liveness of suspected objects. It, too, requires multiple tracing.

Bishop first proposed migration of objects to collect cycles between separately traced partitions [Bis77]. In his scheme, locally unreachable objects are migrated immediately to the partition they are referenced from. Shapiro *et al.* proposed restricting the direction of migration according to a total order among nodes to ensure that all objects on a cycle converge on the same node [SGP90]. Shapiro also considered *virtual* migration to collect cyclic garbage. Here, a locally unreachable object is not physically moved between nodes; instead, the object merely changes the logical space it belongs to. Thus, a logical space may span a number of nodes. Each logical space is collected by marking, so that a local collection may require internode marking messages.

Gupta *et al.* proposed migrating objects to a fixed *dump* node in the system [GF93]. This scheme works with reference counts because it does not require the knowledge of the referencing node. Also, it does not suffer from the problem of multiple migrations. However, moving objects to a fixed node is not scalable or fault-tolerant. To avoid migrating live objects, the scheme *ages* the locally unreachable objects for a certain number of local collections before migrating them. If such an object is accessed (by the mutator) from another node while it is aging, the mutator is expected to migrate it to another node. Such a scheme does not prevent live objects from migrating to the dump node if they are not accessed during the aging period.

6 Conclusions

This paper has presented a simple and efficient way of using object migration to allow collection of distributed cyclic garbage. Our approach is to limit migration to the bare minimum. With high probability, we migrate only cyclic garbage objects since these are usually the ones with distances over the threshold. In addition, we migrate objects directly to a selected destination node to avoid multiple migrations.

Our scheme would add little overhead to distributed reference

listing for collecting non-cyclic garbage. The techniques for estimating distances and destination nodes are low cost and do not require extra messages over what must be sent anyway for distributed garbage collection. Further the scheme retains the fault-tolerance properties of distributed reference counting: as long as the nodes containing the garbage cycle cooperate, progress can be made in collecting the garbage.

The price we pay to achieve these benefits is delay in collecting cyclic garbage. We wait for estimated distances to rise above the distance threshold, and then again for the destination node to be known. We believe that slowness in collecting distributed cyclic garbage is not a serious practical problem because cyclic garbage is only a small fraction of the total garbage. Therefore, our scheme makes the appropriate tradeoff: cyclic garbage is collected eventually, without degrading overall system performance and fault tolerance.

Acknowledgments

The authors are grateful to Miguel Castro, Raymie Stata, James O'Toole, and the referees for their comments.

References

[Ali84] K. A. M. Ali. Garbage Collection Schemes for Distributed Storage Systems. *Proceedings of Workshop on Implementation of Functional Languages*, pages 422–428, Aspenas, Sweden, February 1985.

[Bag91] N. Bagherzadeh. A Parallel Asynchronous Garbage Collection Algorithm for Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 1, March 1991.

[BE86] M. J. Beckerle and K. Eknadham. *Distributed Garbage Collection With No Global Synchronization*, Research Report RC 11667, IBM T. J. Watson Research Center, Yorktown Heights, New York.

[BENOW93] A. Birrell, D. Evers, G. Nelson, S. Owicki, and E. Wobber. *Distributed Garbage Collection for Network Objects.* Systems Research Center Technical Report 116, Digital, December 1993.

[Bev87] D. I. Bevan. Distributed Garbage Collection Using Reference Counting. *Lecture Notes in Computer Science 259*, pages 176–187, Springer-Verlag, June 1987.

[Bis77] P. B. Bishop. 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.

[DLMM93] M. Day, B. Liskov, U. Maheshwari, and A. Myers. References to Remote Mobile Objects in Thor. *ACM Letters on Programming Languages and Systems*, 1994.

[GF93] A. Gupta and W. K. Fuchs. Garbage Collection in a Distributed Object-Oriented System. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 2, April 1993.

[FS94] P. Ferreira and M. Shapiro. Garbage Collection and DSM Consistency. *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 229-241, September 1994.

[HK82] P. Hudak, and R. Keller. Garbage Collection and Task Deletion in Distributed Applicative Processing Systems. *ACM Symposium on Lisp and Functional Programming*, pages 168–178, August 1982.

[Hug85] J. Hughes. A Distributed Garbage Collection Algorithm. *Functional Programming and Computer Architecture (Lecture Notes in Computer Science 201)*, pages 256–272, Springer-Verlag, September 1985.

[JJ92] N. C. Juul, E. Jul. Comprehensive and Robust Garbage Collection in a Distributed System. *1992 International Workshop on Memory Management, (Lecture Notes in Computer Science 637)*, Springer-Verlag, 1992.

[KA93] R. Kordale and M. Ahamad. A Scalable Cyclic Garbage Detection Algorithm for Distributed Systems. *OOPSLA'93 Workshop on Memory Management and Garbage Collection*, September 1993. Contact: kram@cc.gatech.edu.

[LDS92] B. Liskov, M. Day, and L. Shrira. Distributed Object Management in Thor. *Distributed Object Management*, ed. M. T. Ozsu, U. Dayal, and P. Valduriez, Morgan Kaufmann, 1992.

[LeL77] G. LeLann. Distributed Systems, towards a formal approach. *IFIP Congress*, pages 155–160, Torornto, 1977.

[LL92] R. Ladin, and B. Liskov. Garbage Collection of a Distributed Heap. *Int. Conference on Distributed Computing Systems*, pages 708–715, Yokohoma, Japan, June 1992.

[LQP92] B. Lang, C. Queinnec, and J. Piquer. Garbage Collecting the World. *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 39–50, Albuquerque, Jan 1992.

[ML94] U. Maheshwari, and B. Liskov. Fault-Tolerant Distributed Garbage Collection in a Client-Server Object-Oriented Database. *Proceedings of the third Internation Conference on Parallel and Distributed Information Systems*, pages 239-248, September 1994.

[Piq91] J. M. Piquer. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. *PARLE '91 — Parallel Architecture and Languages (Lecture Notes in Computer Science 505)*, pages 150–165, Springer-Verlag, June 1991.

[SDP92] M. Shapiro, P. Dickman, and D. Plainfosse. Robust, Distributed References and Acyclic garbage Collection. *Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992.

[SGP90] M. Shapiro, O. Gruber, and D. Plainfosse. A Garbage Detection Protocol for a Realistic Distributed Object-Support System. *Research Report 1320*, INRIA–Rocquencourt, November 1990.

[Ves87] S. C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, University of Washington, January 1987.