116

Distributed Garbage Collection for Network Objects

Andrew Birrell, David Evers, Greg Nelson,
Susan Owicki, and Edward Wobber

December 15, 1993

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Distributed Garbage Collection for Network Objects

Andrew Birrell, David Evers, Greg Nelson,
Susan Owicki, and Edward Wobber

December 15, 1993

Affiliations

David Evers is currently at the University of Cambridge Computer Laboratory. Susan Owicki is an independent consultant. This work was completed while the authors were at the Systems Research Center.

©Digital Equipment Corporation 1993

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

Authors' Abstract

In this report we present a fault-tolerant and efficient algorithm for distributed garbage collection and prove its correctness. The algorithm is a generalization of reference counting; it maintains a set of identifiers for processes with references to an object. The set is maintained with pair-wise communication between processes, so no global synchronization is required. The primary cost for maintaining the set is one remote procedure call when an object reference is transferred to a new process for the first time. The distributed collector collaborates with the local collector in detecting garbage; any local collector may be used, so long as it can be extended to provide notification when an object is collected. In fact, the distributed collector could be used without a local collector; in that case, the programmer would insert explicit *dispose* commands to release an object. The algorithm was designed and implemented as part of the Modula-3 network objects system, but it should be suitable for a wide range of applications. It tolerates communication and process failure, and can reclaim the space for objects held by a crashed process. The algorithm balances functionality, performance, and fault-tolerance in a way that makes it highly practical to use in implementing distributed systems.

Contents

1 Introduction **1**

2 The algorithm **3**
2.1 Transmitting a network object 5
2.2 Deleting a surrogate 7
2.3 Communication failures 7
2.4 Process termination 8
2.5 Forgetting sequence numbers 9

3 Correctness of the algorithm **9**
3.1 Safety 10
3.2 Liveness 12

4 Related work **12**

5 Summary **14**

Acknowledgements **15**

References **17**

1 Introduction

Garbage collection is a valuable tool for programming distributed systems, for all the reasons that apply to programs that run in a single address space. In addition, network servers often issue shared resources, such as file locks, to their clients; garbage collection can trigger recovery of these resources when the associated storage is collected. Unfortunately, designing a distributed collector that operates well is not a simple problem, and it is small wonder that many algorithms have been proposed. This proliferation of algorithms is in part due to conflicts between various aspects of an ideal collector; for example, minimizing communication costs conflicts with fault-tolerance.

This report describes a distributed garbage collector that is fault-tolerant, handling both process crashes and communication failures, and yet modest in overhead costs. In particular, it collects objects reliably even if processes holding references to them crash. This is essential for the support of long-running servers, which could otherwise suffer from disabling leakage of storage or other resources. The collector was designed to support a distributed programming paradigm called *network objects* [BNOW93], but it should be suitable for a variety of distributed systems.

Network objects provide a means to incorporate remote procedure call in an object-oriented programming style. An *object* consists of a data record and a set of methods, or operations, that can be invoked on the object. A *network object* is an object that can be shared by processes in a distributed system. The process that allocated the network object is called its *owner*, and the instance of the object at the owner is called the *concrete* object. Other processes, known as *clients* may have references to the object. The client and owner can run on different machines or in different address spaces on the same machine. The roles of client and owner are specific to a particular object: the owner of one object may well be a client of another.

A client cannot directly read or write the data fields of a network object to which it holds a reference, it can only invoke its methods. A reference in the client program actually points to a *surrogate* object, whose methods perform remote procedure calls to the owner, where the corresponding method of the concrete object is invoked. There is at most one surrogate for an object in a process, and all references in the process point to that surrogate.

References for a network object may be *marshaled* from one process to another during method invocation as arguments or results. A network object is marshaled by transmitting its *wireRep*, which consists of a unique identifier for the owner process, plus the index of the object at the owner. Since object indices are not reused in a process, the wireRep uniquely identifies the object for all time. Note that the concrete object does not migrate. Network objects may be passed from one client to another as well as from the owner to a client.

The network object garbage collector is based on a generalization of reference counting. The owner of a shared object O maintains a set *O.dirtySet* which contains identifiers for all the processes that have a surrogate for O. The set is maintained by communication between processes. When a client first receives a reference to a particular object, it makes a *dirty* call to the owner and then creates a surrogate. When the surrogate is no longer reachable, as determined by the client's local garbage collector, the client makes a *clean* call and deletes the surrogate. When *O.dirtySet* is empty, the owner can reclaim the memory for O, unless it is being used locally. The collector thus preserves a key invariant: If there is a surrogate for object O at client A, then $A \in O.dirtySet$. (We shall see later that this invariant must be modified slightly to deal with long-lasting communication failures.)

Note that the owner keeps not just a count of the references to an object, but the identities of all processes with surrogates. This is helpful for achieving fault-tolerance, because it allows the clean and dirty operations to be idempotent. Moreover, it makes collection possible when a client process terminates without making a clean call. The network object runtime at the owner of an object detects termination of any client process; it can then remove the client from any dirty sets in which it appears.

This distributed collector, like collectors that use simple reference counts, is unable to collect cycles. Therefore, the programmer should either take care not to form cycles or break them explicitly to allow for collection. The cost of full cycle collection is quite high, as will be discussed in Section 4. We are considering extending the algorithm to collect cycles that span a small number of machines.

A number of properties are desirable in a distributed collector:

- It should collect all objects that are unreachable, and no others.
- It should tolerate process and communication failure, and deal gracefully with intermittent communication outages. In particular, process failure should not cause objects to become uncollectable.
- It should be able to take advantage of an existing local collector running within a process. Ideally, it would be independent of the algorithm used by the local collector.

- The presence of garbage collection should be transparent to the programmer. In particular, it should be possible to transmit object references from client to client, and not just from the owner to a client.
- The overhead of garbage collection should not be too high. This means avoiding excess inter-process communication and synchronization, and allowing the collector to run in parallel with the computation.

Our collection algorithm comes close to meeting all of these goals. We believe that the tradeoffs between conflicting goals have been made in ways that make the collector very attractive for practical distributed programs. In rare circumstances, communication failure may be interpreted as process failure, and an object may be collected prematurely. If communication is restored and an attempt is made to use the surrogate, the error will be detected and reported. In this case, the goal of reclaiming space when a process crashes conflicts with the goal of reclaiming only unreachable objects.

The algorithm described in this report has been implemented as part of the network object system.

The remainder of this report is organized as follows. Section 2 describes the algorithm in some detail, and Section 3 provides a proof of its correctness. Section 4 compares our approach to others in the literature. Finally, section 5 summarizes our results.

2 The algorithm

Here we discuss our collection algorithm in detail. We start with a description of the data structures it requires.

Object table. Each process maintains an *object table* (see Figure 1), which maps a wireRep $w(O)$ to the local instance of the corresponding network object O, if there is one. For the owner of an object (process P in Figure 1) the table contains a pointer to the concrete object. A concrete object must be in the table whenever another process has a surrogate for it. To ensure this, a concrete object is entered into its owner's table when it is first marshaled; it remains there until the distributed collector detects the deletion of its last surrogate.

The object table also contains entries for all surrogates that exist in the process. It maps the wireRep for a remote object to the unique local surrogate for that object, if one exists (see Process Q in Figure 1). If the

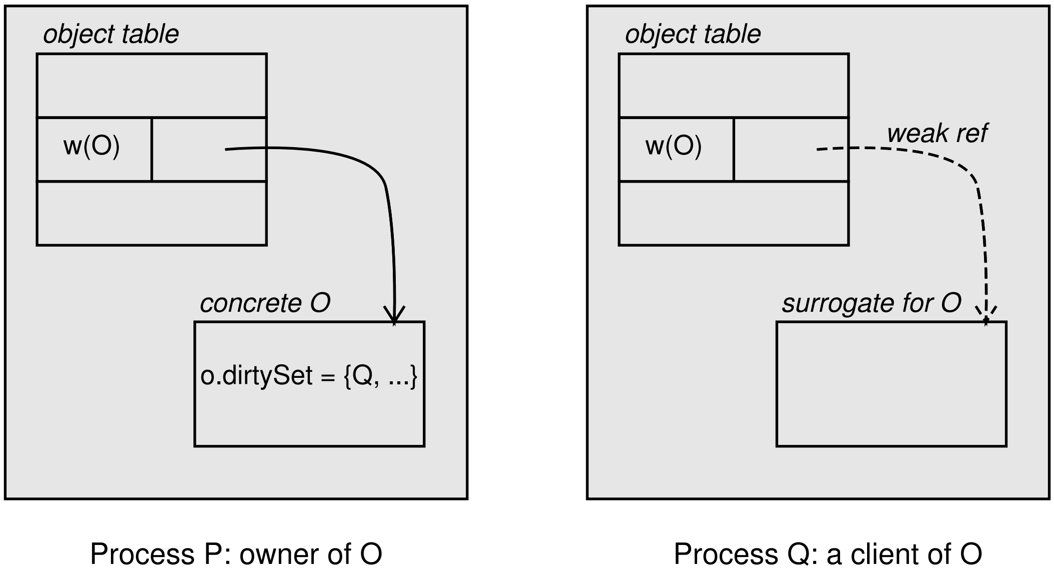

Figure 1: Object tables at owner and client processes

wireRep has been received but a surrogate has not yet been created, the mapping yields NIL. Once the surrogate has been created (after a dirty call to the owner) it is placed in the table.

The object table plays two key roles in garbage collection. First, it is used to find the object referred to by an incoming wireRep; this is required for method invocation, clean/dirty calls, and unmarshaling a transmitted object. Second, the table's references to concrete and surrogate objects are central to the interaction with the local garbage collector. For a concrete object, the reference keeps the object reachable, so that it will not be reclaimed by the local collector. Since the object remains in the table until the distributed collector detects that there are no remote references to it, this guarantees that the object will not be collected prematurely.

For surrogates, however, the reference in the table is a *weak ref*, which has quite a different effect. A weak ref does not keep its referent from being collected by the local collector. However, when it is collected, a cleanup routine associated with the weak ref is scheduled for execution. Thus a surrogate becomes unreachable when there is no path to it except through the object table. At this point the local collector replaces the weak ref in the object table with a distinguished *null* weak ref value and schedules the cleanup routine. As we shall see below, the cleanup routine for surrogates causes the required clean call to the object's owner. Weak refs provide the interface between the local collector and the distributed collector. Any local collection strategy is acceptable, so long as the collector can support this interface or an equivalent one. Hayes [Hay92] discusses strategies for implementing collector-based object cleanup.

Client Information. As already mentioned, a *dirty set* is maintained for each object by its owner. The dirty set contains identifiers for all processes that have surrogates for the object. When the dirty set becomes empty, the object can be removed from its owner's object table. The dirty set may be maintained conservatively: it may sometimes contain processes that do not have surrogates, so long as the collector guarantees to remove them eventually. This conservative management is necessary for handling communication failures. It also conveniently allows us to delay clean calls, which can then be batched for better performance.

Dealing with communication failures requires us to keep further information on client processes. Even with a reliable transport, failures and multiple threads may cause calls to be delivered out of order, as described in section 2.3. To deal with this, a *sequence number* is attached to each clean or dirty call. The sequence number must increase with each new operation from the client. (Some authors use the term "timestamp" to refer to this sort of sequence number.) Let *seqno(O, P)* be the largest sequence number seen at O's owner on a clean or dirty call for object O from process P. An incoming operation will be performed only if its sequence number exceeds this value; otherwise it has no effect.

It might appear that this use of sequence numbers forces us to retain a sequence number for every process that has ever had a surrogate for an object. In fact, in most cases we need keep sequence numbers only for clients in the dirty set. The exact circumstances under which sequence numbers must be retained are described in Section 2.5.

2.1 Transmitting a network object

In this section we discuss the steps involved in transmitting a network object under normal operation; the treatment of communications failures is delayed until Section 2.3.

Suppose process P marshals a network object O to process Q, as an argument or result of remote method invocation. P may be the owner of O, or it may be a client that has a surrogate for O. In either case, P sends Q the wireRep *w(O)*. When *w(O)* arrives, Q looks it up in its object table to see if there is a corresponding local object.

If Q finds either a surrogate or concrete object, that object is used as the required argument or result. Note that if a client transmits a remote object back to its owner, this use of the object table causes the owner to access the concrete object; no surrogate is created.

If Q does not find an object in the table, there are two possibilities to consider. First, *w(O)* may be in the table but mapped to a NIL reference. In this case surrogate creation is under way, and the thread doing the unmarshaling suspends itself until the surrogate is created or the attempt fails. Alternatively, *w(O)* may not be in the table, or it may be there with a null weak ref indicating that a surrogate existed but had been collected. In this case, the recipient must create a new surrogate. It first enters *w(O)* in the table with a mapping to NIL, releases the lock on the table, and then makes a dirty call to the owner of the object. Assuming no communication failures, the owner receives the call and adds Q to O's dirty set. When the dirty call returns, Q creates a surrogate for O and enters it in the object table.

There is one more wrinkle to be considered in transmitting a network object. This is the potential race condition between the dirty call from client Q and a clean call from a client whose surrogate has been deleted. If the clean call arrived first, and if it left *O.dirtySet* empty, then O might be removed from its owner's object table and its space reclaimed by the local collector. When the dirty call arrives, the object will no longer be available.

To prevent this scenario, we make sure that *O.dirtySet* remains non-empty while O is being transmitted. When the sending process P is O's owner, this is accomplished by putting P into *O.dirtySet* until an acknowledgment from Q indicates that the reference has been received. Since Q sends the acknowledgment after completing the dirty call, this guarantees that O's dirty set remains non-empty, and its memory is not collected. When P is not the owner of O, it must have a surrogate for O. This surrogate is kept reachable until Q's acknowledgment is received. Since a reference to the surrogate is on the stack during transmission, we simply ensure that the transmitting procedure not return until acknowledgment from Q is received. So long as this surrogate is reachable, the basic collector invariant guarantees that P is in *O.dirtySet* and O's space will not be collected.

It appears that we may now require an extra acknowledgment during method invocation. Just how much overhead is added? Recall that network objects are transmitted as arguments or results of remote method invocations. When P marshals O as an argument, the method's return serves as the acknowledgment that transmission is complete; no additional message is required. When P marshals O as a result, an explicit acknowledgment must be sent when unmarshaling is finished. In this case the need for acknowledgement results in an extra message. However, the thread that waits for the acknowledgment is not on any critical path, so performance is not seriously affected.

2.2 Deleting a surrogate

We now consider how surrogate deletion is treated in the normal case, and once again delay the discussion of fault-handling until Section 2.3.

Collection of surrogates is the responsibility of the client's local garbage collector. When the client's collector determines that a surrogate is unreachable, the object's owner must be informed so that the client can be removed from its dirty set. We have already mentioned how weak refs allow the distributed garbage collector to be informed of surrogate collection so it can take this action. To recap, when the client's collector determines that the surrogate is not reachable (except from the weak ref in the client's object table), it prepares to reclaim the surrogate's memory. However, it first schedules a cleanup routine that was registered when the weak ref was created and replaces the weak ref with a special null value.

When the cleanup routine begins execution, it checks the object table to see if the entry for this object's wireRep still has the special null weak ref. If not, a new surrogate for the object will have been created (or will be in the process of being created) and no cleanup action is required. But if the null weak ref is still present, cleanup action is necessary. The object table entry is removed, and the wireRep is put on a queue of objects to be processed later by a cleaning demon. This demon is responsible for sending clean calls to the owner. Delaying the clean calls allows them to be batched, to reduce communication cost and improve performance. However, the clean operation logically occurs when the cleanup routine puts the wireRep on the demon's queue, so its sequence number is generated at this time.

2.3 Communication failures

Although the garbage collector makes use of a reliable transport, it is still possible for remote calls to fail because of temporary communication problems. We assume that a remote call returns with an indication of success or failure. A successful return means that the remote operation was performed. When failure is reported, however, it is impossible to tell whether or not the remote operation was carried out. It is even possible that a message was delayed in the communication system, and the remote operation will be performed at some unpredictable future time. The distributed collection algorithm must deal with failures of clean and dirty calls.

When a dirty call fails, no surrogate is created. It would not be safe to create one, because the object's owner may not have received the dirty call. However, it is also possible that the owner *did* receive the dirty call, so the object and a sequence number for the clean call are added to the cleanup demon's queue. Note that this may cause an unnecessary clean call, but that does no harm. The effect of a clean call is to remove the client from the object's dirty set; if it is not in the set, the clean call is a no-op.

When a clean call fails, the cleanup demon merely leaves the request on its queue, keeping the same sequence number. The clean call will be repeated until it succeeds, or until the owner's termination is detected.

2.4 Process termination

Processes that terminate, whether normally or abnormally, cannot be expected to notify the owners of all network objects for which they have surrogates. Therefore, the distributed collector must detect process termination and update dirty sets accordingly. Various mechanisms may be used for detecting termination; for example, when owner and client are running on the same machine, the operating system can provide the information. Our collector detects termination by having each process periodically ping the clients that have surrogates for its objects. If the ping is not acknowledged after sufficient time, the client is assumed to have terminated, and is removed from all dirty sets at that owner. Thus a process that dies holding a surrogate does not prevent an object from being collected.

Note that this method of detecting termination carries with it the risk of mistaking a communication failure for process termination. If this occurs, the collector may incorrectly reclaim the space for some object O even though a surrogate still exists. Later, if communication is restored, the client might try to invoke one of O's methods. If this happens, the owner will be able to detect the error, because there will be no entry for the wireRep $w(O)$ in its object table. (This is why wireReps are not reused.) Thus the impact on the client will be that the operation fails, as it would have failed had it been attempted during the communication outage.

2.5 Forgetting sequence numbers

Earlier, we described how sequence numbers are used to detect and ignore out-of-order operations. The straightforward implementation retains, for each object O, the highest sequence number received from each process P that has ever called in clean or dirty for O. Fortunately, a more space-efficient implementation is possible. In most cases, P's sequence number can be dropped when P is removed from O's dirty set. It is only when some dirty call from P for O has failed that sequence number information must be retained.

The reason is not hard to see. The algorithm must protect against two potential errors: removing P from O's dirty set because of a late clean call, and adding P to O's dirty set because of a late dirty call. Note that a late clean call can do no harm when P is not in O's dirty set.

Clean calls are generated asynchronously by P's cleanup demon, and may arrive at any time. Thus a sequence number must be retained when P is in O's dirty set, for protection from late clean calls. However, dirty calls are synchronous with surrogate creation. No clean or dirty call with a later sequence number will be generated until after the dirty call has returned. Only if the dirty call reports failure is there a possibility that it will be performed out of order at the owner. Thus if no dirty call for O from P has ever failed, there is no need for protection from late dirty calls, and the sequence number can be dropped when P is not in O's dirty set. Recall that when a dirty call fails, a cleanup request is added to the cleanup demon's queue. Clean calls scheduled as a result of failed dirty calls are flagged as *strong clean* calls, while those scheduled as a result of surrogate deletion are not. Once the owner receives a strong clean call, it retains *seqno(O, P)* until P or the owner itself terminates.

3 Correctness of the algorithm

Ideally, a garbage collector should collect all storage that becomes unreachable (*liveness*), and nothing else (*safety*). In the network object system, implementation considerations constrain us to accept a less than perfect collector. A communication failure between an object's owner and a client may allow the object to be collected prematurely. And some objects that are no longer reachable may not be reclaimed, either because they are part of a cycle that spans multiple address spaces, or because a conservative local collector fails to detect that a surrogate is unreachable. In this section we state precisely what our distributed collector is intended to do and prove that it meets this specification.

First, let us define some notation. Let *nofail(P, Q)* mean that process P has never concluded that process Q terminated. We will say that a surrogate exists at a client from the time of its creation (after a successful dirty call) until the local collector determines that it is collectable; at this point the object table entry receives a null weak ref, and the cleanup routine is scheduled.

3.1 Safety

To demonstrate the safety of the algorithm, we show that if P has a surrogate for O, and *nofail(owner(O), P)*, then O is in its owner's object table. The proof relies on two invariants:

INVARIANT 1. If P has a surrogate for O, and *nofail(owner(O), P)*, then $P \in O.dirtySet$.

INVARIANT 2. If $O.dirtySet$ is not empty, then O is in its owner's object table.

It is easy to see that the two invariants together imply the desired safety property. Initially, both invariants hold, since no surrogates exist and all dirty sets are empty. To show that the collector's actions preserve the invariants, we make use of two lemmas.

LEMMA 1. Suppose P makes a successful dirty call for O. Then as long as the surrogate created as a result of that call exists, no dirty or clean call for O will be initiated at P.

PROOF. From the time the dirty call is initiated until the surrogate becomes collectable, P's object table entry for O contains either the NIL reference or a weak ref to a surrogate. Before a dirty call is initiated during object unmarshaling, or a clean call is initiated by the cleanup routine, the state of the object table is checked. If it contains NIL or a non-null weak ref, the clean or dirty call is not made.

LEMMA 2. Suppose P creates a surrogate after a successful dirty call with sequence number sn. While the surrogate exists, $seqno(O, P) = sn$.

PROOF. The successful dirty call sets $seqno(O, P) := sn$. By Lemma 1, no clean or dirty call with a sequence number later than sn is initiated as long as the surrogate exists. Clean or dirty calls with earlier sequence numbers may reach *owner(O)*, but they will have no effect on *seqno(O, P)*.

Now let us turn to the proof of the two invariants. To show that Invariant 1 is preserved, we must show:

- When a surrogate for O is created at P, $P \in$ *O.dirtySet*.

- When P is removed from *O.dirtySet*, there is no surrogate for O at P.

For the first point, note that surrogate creation follows a successful dirty call, which made $P \in$ *O.dirtySet*. By Lemma 1, no clean call with a greater sequence number can occur between the dirty call and surrogate creation. Thus $P \in$ *O.dirtySet* when the surrogate is created. For the second point, consider a clean call with sequence number *sn*. At the time the clean call was initiated, a test of P's object table found that there was no surrogate for O, and no outstanding dirty call. Suppose a surrogate exists when the clean call is executed at O's owner. The surrogate must have been created after a successful dirty call with sequence number $sn' > sn$. By Lemma 2, $seqno(O,\ P) = sn' > sn$, and the clean call will have no effect on *O.dirtySet*.

To show that Invariant 2 is preserved, we need to show:

- When P is added to *O.dirtySet*, O is in its owner's object table.

- When O is removed from its owner's object table, *O.dirtySet* is empty.

The second is trivially true, since it is detection of an empty dirty set that causes O to be removed from its owner's object table. To see the first, note that there are two ways that P may be added to *O.dirtySet*. The first occurs when O is marshaled from its owner, and the owner is added to *O.dirtySet*; in this case O was added to the object table (if not there already) as part of the same marshaling operation. The other occurs as part of executing a dirty call for O from P. For the dirty call to execute successfully, O must be in its owner's object table.

The combined effect of Invariants 1 and 2 is to guarantee that, so long as communication between a client and the object's owner has not failed, the client will be able to invoke methods of O, because the owner will have an object table entry for O, and that will protect O's storage. In addition, in the absence of communication failures, a dirty call will find the object in its owner's object table. This follows easily from a third invariant.

INVARIANT 3. While P is marshaling O, P is in *O.dirtySet*.

PROOF. If P is the owner of O, P puts itself into *O.dirtySet* before it marshals O and keeps itself there until an acknowledgment indicates that marshaling is complete. If P is a client, it must have a surrogate for O, and it keeps that surrogate reachable until receiving the acknowledgment. By Invariant 1, this implies $P \in$ *O.dirtySet*.

3.2 Liveness

In order to demonstrate that unreachable objects are eventually collected, we show that if after some time there is no surrogate for O at P, and no process marshals O to P, then eventually it remains true that $P \notin O.dirtySet$.

PROOF OF LIVENESS. Consider the last dirty call initiated at P for O; let its sequence number be sn. This dirty call either succeeded or failed. If it succeeded, a surrogate was created. Since that surrogate no longer exists, it was detected to be collectable, and a clean call with sequence number $sn' > sn$ was enqueued at P. If it failed, a clean call was enqueued immediately, also with sequence number $sn' > sn$. Eventually P's cleanup demon will make a clean call to O's owner, repeating it if it fails. If it is ever received, the clean call will remove P from $O.dirtySet$. If not, there is a long-lasting communication failure between P and O's owner. Eventually this will cause O's owner to conclude that P has terminated and remove P from $O.dirtySet$.

It remains to be shown that $P \notin O.dirtySet$ remains true. The only way P could be re-inserted in $O.dirtySet$ is as the result of a dirty call. If a dirty call arrives after the clean call with sequence number sn' it must be one that was reported as a failure, since successful calls are delivered synchronously. P must have made a strong clean call in response to that failure, so *seqno(O, P)* will be retained even when $P \notin O.dirtySet$. Since any dirty call from P for O has a sequence number less that sn', its arrival will not cause P to be added to $O.dirtySet$.

4 Related work

A variety of distributed collection algorithms have been presented in the literature, some based on reference counting and others on tracing. We consider each type in turn. It should be noted that some of these algorithms can deal with objects that persist over process crashes and objects that can migrate from one process to another. We have not attempted to extend our approach to handle those situations, since network objects are not persistent and do not migrate.

Among the reference-counting schemes are several with lower overhead than the approach proposed here [Bev87, Gol89, Piq91, WW87]. All can transfer references with a single message—they don't require a dirty call to the object's owner—but none can collect an object after a client terminates while holding a reference to it. Each of the approaches has its own way of avoiding the need for a dirty call. Weighted references [Bev87, WW87] are the simplest to describe. In this approach, each reference has an associated integer weight. When the reference is transmitted to another process, part of the weight goes with the newly-created reference, and part remains with the original. When a reference is deleted, the owner is informed of its weight; when the sum of deleted weights equals the starting weight, the object may be collected. Note, however, that if a process crashes, the owner has no way of knowing how much weight it held, and so cannot recover from the failure.

Shapiro et al. [SDP92] maintain much the same information as we do, although their data structures are organized differently. For example, the owner of an object has a data structure called a *scion* for each client holding a reference to the object; the set of scions for an object plays much the same role as our dirty set in making reference-count operations idempotent. Shapiro et al. use sequence numbers to eliminate out-of-order calls, and thus achieve better resiliency to communication errors than the weighted reference scheme. However, as with weighted references, they do not notify the owner when an object is transferred. This makes it difficult to recover from process failure. They sketch several possible extensions to handle process failure, but do not work out the details.

In contrast, the scheme proposed by Mancini and Shrivastava [MS91] deals explicitly with process failure. However, their model of object transfer is somewhat less general than ours. In their approach, the owner is notified when an object is transferred; however the notification is done by the sender, before transmitting the reference. When the object is transferred from its owner, this notification involves only a local procedure call rather than remote communication. The tricky point in this approach is that the sender might crash after notifying the owner but before sending the reference, leaving the owner's reference count too high. This is handled by maintaining an *unused* bit at the owner for each client of an object; the bit is initially true, and is set to false when the client invokes a method of the object. If an object remains unused for a long time, the client is queried as to whether it received the reference; if not, the reference count can be decremented. There is a potential race condition between the query and the arrival of the reference. It is dealt with by having the recipient reply "yes" to the query if it has asked for a reference but not yet received it. Thus it is necessary for the recipient to know the identity of the object it is about to receive. This works in Mancini and Shrivastava's programming model, because an object is transferred only after a process has explicitly requested it. We do not see how to apply this scheme in an environment like ours, where the recipient typically cannot know in advance what object it will receive in a remote invocation.

Tracing collectors, whether based on mark-and-sweep [Aug87, Der90, HK82, Hug85, LL92, LL86, Sch89] or copying [Rud86], can collect cycles, a strong point in their favor. In the algorithms cited, garbage collection is performed in parallel at all processes. During the tracing phase, processes exchange information about the reachability of external references. The end of the tracing phase is reached when no process has any untraced references. Either a distributed termination detection algorithm or a centralized server is typically used to determine when this point is reached.

These algorithms require co-operation among a potentially very large set of processes: those in the transitive closure of the relation "share a network object". This is unacceptable in an environment where network objects are used to implement services—for example, all processes that use the same name server (at least) would be involved in a collection. Thus tracing algorithms do not seem a realistic alternative for network objects.

However, we are considering supplementing the current network objects collector with tracing, to collect cycles that span a fixed small number of processes. Lang et al. [LQP92] propose a collector that operates on a *process group*, collecting all cycles whose edges do not leave the group. Such an approach should be feasible in the network objects system.

5 Summary

The ideal distributed collector would be safe (collecting only garbage), live (collecting all garbage), efficient, and fault-tolerant, and it would impose no burden on the programmer. The design of the network object collector is based on a set of choices about the relative importance of these goals. For example, our design has relatively low overhead (the primary overhead is one remote procedure call on the receipt of a new object). Weighted references have even lower overhead, but do not allow the collection of objects when a process fails. By contrast, our design deals gracefully with process failure, but risks the occasional collection of a reachable object when there is a long-lasting communication failure. Our algorithm cannot recover cyclic garbage; this can be viewed as a failure of liveness or a burden on the programmer (who can ensure liveness by avoiding or breaking cycles). The alternative is to use mark-and-sweep collection, and we know of no way to do this without seriously reducing performance.

We believe that the set of choices embodied in our collector represents a good balance for a wide range of programs, including client/server systems and distributed computations. In general, programming with network objects is quite easy; in particular, the collector places no restrictions on the transmission of objects between processes. The network objects system has been in operation for over a year. Early users, who are primarily programming client/server systems, find that it suits their needs well. Further experience will allow us to evaluate its use in a wider range of applications.

Acknowlegements

Several of the ideas presented here originated in discussions with Andrew Black, Rivka Ladin, and Butler Lampson. We would also like to thank Cynthia Hibbard, Jim Horning, Sharon Perl, and Mike Schroeder, who contributed to the presentation of this report.

References

[Aug87] Lex Augusteijn. Garbage collection in a distributed environment. In *PARLE '87 – Parallel Architectures and Languages Europe*, pages 75–93. Springer-Verlag, June 1987.

[Bev87] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE '87 – Parallel Architectures and Languages Europe*, pages 176–187. Springer-Verlag, June 1987.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *14th ACM Symposium on Operating Systems Principles*, pages 217–230, December 1993.

[Der90] Margaret H. Derbyshire. Mark scan garbage collection on a distributed architecture. *Lisp and Symbolic Computation*, 3(2):135–170, April 1990.

[Gol89] B. Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 313–321, June 1989.

[Hay92] Barry Hayes. Finalization in the collector interface. In *Memory Management International Workshop IWMM 92*, pages 277–297. Springer-Verlag, September 1992.

[HK82] Paul Hudak and Robert Keller. Garbage collection and task deletion in distributed applicative processing systems. In *ACM Symposium on Lisp and Functional Programming*, pages 168–178, August 1982.

[Hug85] John Hughes. A distributed garbage collection algorithm. In *Functional Programming and Computer Architecture*, pages 256–272. Springer-Verlag, September 1985.

[LL86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *5th ACM Symposium on the Principles of Distributed Computing*, pages 29–39, August 1986.

[LL92] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *International Conference on Distributed Computing Systems*, 1992.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *10th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–49, January 1992.

[MS91] L. Mancini and S. K. Shrivastava. Fault-tolerant reference counting for garbage collection in distributed systems. *The Computer Journal*, 34(6):503–513, December 1991.

[Piq91] José Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE '91 – Parallel Architectures and Languages Europe*, pages 150–165. Springer-Verlag, June 1991.

[Rud86] Martin Rudalics. Distributed copying garbage collection. In *ACM Symposium on Lisp and Functional Programming*, pages 364–372, August 1986.

[Sch89] Marcel Schelvis. Incremental distribution of timestamp packets: A new approach to distributed garbage collection. In *OOPSLA'89 Conference Proceedings*, pages 37–48, October 1989.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed, references and acyclic garbage collection. In *11th ACM Symposium on Principles of Distributed Computing*, pages 135–146, August 1992.

[WW87] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE '87 – Parallel Architectures and Languages Europe*, pages 432–443. Springer-Verlag, June 1987.