A Cyclic Distributed Garbage Collector for Network Objects

Helena Rodrigues* and Richard Jones

Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK
Tel: +44 1227 764000 x7754, Fax +44 1227 762811
email: {hccdr,R.E.Jones}@ukc.ac.uk

Abstract. This paper presents an algorithm for distributed garbage collection and outlines its implementation within the Network Objects system. The algorithm is based on a *reference listing* scheme, which is augmented by *partial tracing* in order to collect distributed garbage cycles. Processes may be dynamically organised into groups, according to appropriate heuristics, to reclaim distributed garbage cycles. The algorithm places no overhead on local collectors and suspends local mutators only briefly. Partial tracing of the distributed graph involves only objects thought to be part of a garbage cycle: no collaboration with other processes is required. The algorithm offers considerable flexibility, allowing expediency and fault-tolerance to be traded against completeness.

Keywords: distributed systems, garbage collection, algorithms, termination detection, fault tolerance

1 Introduction

With the continued growth of interest in distributed systems, designers of languages for distributed systems are turning their attention to garbage collection [24, 21, 16, 14, 15, 3, 18, 19, 17, 9, 22], motivated by the complexity of memory management and the desire for transparent object management. The goals of an ideal distributed garbage collector are

safety: only garbage should be reclaimed;
completeness: all objects that are garbage at the start of a garbage collection cycle should be reclaimed by its end. In particular, it should be possible to reclaim distributed cycles of garbage;
concurrency: distributed garbage collection should not require the suspension of mutator or local collector processes; distinct distributed garbage collection processes should be able to run concurrently;
efficiency: garbage should be reclaimed promptly;
expediency: wherever possible, garbage should be reclaimed despite the unavailability of parts of the system;

* Work supported by JNICT grant (CIENCIA/BD/2773/93-IA) through the *PRAXIS XXI* Program (Portugal).

scalability: distributed garbage collection algorithms should scale to networks of many processes;

fault tolerance: the memory management system should be robust against message delay, loss or replication, or process failure.

Inevitably compromises must be made between these goals. For example, scalability, fault-tolerance and efficiency may only be achievable at the expense of completeness, and concurrency introduces synchronisation overheads. Unfortunately, many solutions in the literature have never been implemented so there is a lack of empirical data for the performance of distributed garbage collection algorithms to guide the choice of compromises.

Distributed garbage collection algorithms generally follow one of two strategies: tracing or reference counting. Tracing algorithms visit all 'live' objects [12, 7]; global tracing requires the cooperation of all processes before it can collect any garbage. This technique does not scale, is not efficient and requires global synchronisation. In contrast, distributed reference counting algorithms have the advantages for large-scale systems of fine interleaving with mutators, and locality of reference (and hence low communication costs). Although standard reference counting algorithms are vulnerable to out-of-order delivery of reference count manipulation messages, leading to premature reclamation of live objects, many distributed schemes have been proposed to handle or avoid such race conditions [2, 11, 20, 23, 3, 18].

On the other hand, distributed reference counting algorithms cannot collect cycles of garbage. Collecting interprocess garbage cycles is an important issue of our work: we claim that it is fairly common for objects in distributed systems to have cyclic connections. For example, in client-server systems, objects that communicate with each other remotely are likely to hold references to each other, and often this communication is bidirectional [27]. Many distributed systems are typically long running (e.g. distributed databases), so floating garbage is particularly undesirable as even small amounts of uncollected garbage may accumulate over time to cause significant memory loss [19]. Although interprocess cycles of garbage could be broken by explicitly deleting references, this would lead to exactly the scenario that garbage collection is supposed to replace: error-prone manual memory management.

Systems that use distributed reference counting as their primary distributed memory management policy must reclaim cycles by using a complementary tracing scheme [14, 16, 13, 17], or by migrating objects until an entire garbage cyclic structure is eventually held within a single process where it can be collected by the local collector [24, 19]. However, migration is communication-expensive and existing complementary tracing solutions either require global synchronisation and the cooperation of all processes in the system [14], place additional overhead on the local collector and application [17], rely on cooperation from the local collector to propagate necessary information [16], or are not fault-tolerant [16, 17].

This paper presents an algorithm and outlines its implementation for the Network Objects system [4]. Our algorithm is based on a modification of distributed reference counting — *reference listing* [3] — augmented by *partial tracing* in order to collect distributed garbage cycles [13]. We use heuristics to form groups of processes dynamically that cooperate to perform partial traces of subgraphs suspected of being garbage.

Our distributed algorithm is designed not to compromise our primary goals of efficient reclamation of local and distributed acyclic garbage, low synchronisation overheads, avoidance of global synchronisation, and fault-tolerance. To these ends, we trade some degree of completeness and efficiency in collecting distributed cycles. However, eventually distributed garbage cycles will be reclaimed. In brief, our aim is to match rates of collection against rates of allocation of data structures. Objects that are only reachable from local processes have very high allocation rates, and therefore must be collected most rapidly. The rate of creation of references to remote objects that are not part of distributed cycles is much lower, and the rate of creation of distributed garbage cycles is lower still and hence should have the lowest priority for reclamation.

The paper is organised as follows. Section 2 briefly describes the overall design of the Network Objects system and introduces terminology. Section 3 describes how partial tracing works in the absence of failures. Section 4 describes termination and how the collectors are synchronised. Section 5 describes how process failures are handled. We discuss related work in Section 6, and conclude and present some points for future work in Section 7.

2 Terminology and Network Objects Overview

Our algorithm is designed for use with the Network Objects system, a distributed object-based programming system for Modula-3 [4]. A distributed system is considered to consist of a collection of *processes*, organised into a network, that communicate by exchange of *messages*. Each process can be identified unambiguously, and we identify processes by upper-case letters, e.g. A, B, ..., and objects by lower-case letters suffixed by the identifier of the process to which they belong, e.g. xA, xB, ...

From the garbage collector's point of view, *mutator* processes perform computations independently of other mutators in the system (although they may periodically exchange messages) and allocate objects in local heaps. The state of the distributed computation is represented by a *distributed graph* of objects. Objects may contain references to objects in the same or another process. Each process also contains a set of *local roots* that are always accessible to the local mutator. Objects that are reachable by following from a root a path of references held in other objects are said to be *live*. Other objects are said to be *unreachable* or *dead*, and they constitute the *garbage* to be reclaimed by a *collector*. A reference to an object held on the same process is said to be *local*; a reference to an object held on a remote process is said to be an *external* or *remote*. A collector that operates solely within a local heap is called a *local collector*.

Only *network objects* may be shared by processes. The process accessing a network object for which it holds a reference is called the *client*, and the process containing the network object is called its *owner*. Clients and owners may run on different processes within the distributed system. Network objects cannot migrate from one process to another.

A client cannot directly access the data fields of a network object for which it holds a reference, but can only invoke the object's methods. A reference in the client process points to a *surrogate* object (see the dashed circle in figure 1), whose methods perform remote procedure calls to the owner, where the corresponding method of the *concrete* object is invoked. A process may hold at most one surrogate for a given concrete object, in which case all references in the process to that object point to the surrogate.

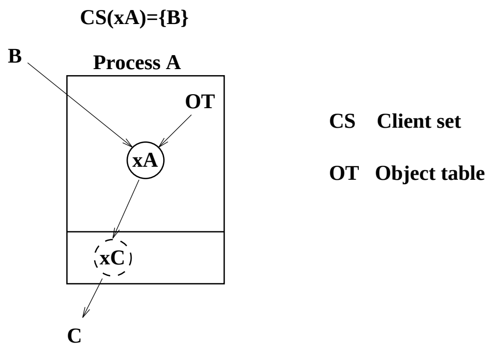

Fig. 1. Surrogates, object table and client sets

Each process maintains an *object table* (see figure 1). The object table of the owner of a concrete object xA contains a pointer to xA as long as any other process holds a surrogate for it. A process's object table also holds entries for any surrogates held (not represented in the figure).

The heap of a Modula-3 process is managed by garbage collection. Local collections are based on tracing from local roots — the stack, registers, global variables and also the object table. We shall refer to those public network objects that are referenced from other processes through the table as *OT roots*. The object table is considered a root by the local collector in order to preserve objects reachable only from other processes.

Object table entries are managed by the distributed memory manager. The Network Objects system uses a variation of reference counting called *reference listing* to detect distributed acyclic garbage [3]. Rather than maintain a count in each concrete object of the number surrogates for it, each object maintains a *client set*[2] of the names of all those processes that hold a surrogate for it.

For the purpose of the algorithm there are two operations on references that are important in the system: transmission of a reference to another process and deletion of a remote reference.

[2] In Network Objects terminology this set is called the *dirty set*.

References to a network object may be *marshalled* from one process to another either as arguments or results of methods. A network object is marshalled by transmitting its *wireRep* — a unique identifier for the owner process plus the index of the object at the owner. If the process receiving the reference is not the owner of the object, then the process must create a local surrogate. In order for a process to marshall a wireRep to another process, the sender process needs either to be the owner of the object or to have a surrogate for that object. This operation must preserve a key invariant: whenever there is a surrogate for object xA at client B, then $B \in xA.clientSet$.

Suppose process A marshalls a network object xA to process B, as an argument or result of a remote method invocation. A may be the owner of xA, or it may be a client that has a surrogate for xA. In either case:

1. A sends to B the wireRep of xA and waits for an acknowledgement.
2. Before B creates a surrogate for xA, it sends a *dirty call* to xA's owner.
3. Assuming no communication failure, the owner receives the call, adds B to xA's client set and then sends an acknowledgement back to B.
4. When the acknowledgement is received, B creates the surrogate and then returns an acknowledgement back to A. If B already has a surrogate for xA, the surrogate creation step is skipped.

Surrogates unreachable from their local root set are reclaimed by local collectors. Whenever a surrogate is reclaimed, the client sends a *clean call* to the owner of the object to inform it that the client should be removed from its client set. When an object's client set becomes empty, the reference to the object is removed from the object table so that the object can be reclaimed subsequently by its owner's local collector.

Race conditions between dirty and clean calls must be avoided. If a clean call were to arrive before a dirty call, removing the last entry from the client set, an object may be reclaimed prematurely. To prevent this scenario arising, an object's client set is kept non-empty while its wireRep is being transmitted. This scheme also handles messages in transit. Each clean or dirty call is also labelled with a sequence number; these increase with each new call sent from the client. This scheme makes the transmission and deletion of references tolerant to delayed, lost or duplicated messages.

Processes that terminate, whether normally or abnormally, cannot be expected to notify the owners of all network objects for which they have surrogates. The Network Objects collector detects termination by having each process periodically ping the clients that have surrogates for its objects. If the ping is not acknowledged in time, the client is assumed to have terminated, and is removed from all client sets at that owner. A more detailed description of these operations, with a proof of their correctness, is described in [3].

3 Three-Phase Partial Tracing

Our algorithm is based on the premise that distributed garbage cycles exist but are less common than acyclic distributed structures. Consequently, distributed cyclic garbage must be reclaimed but its reclamation may be performed more slowly than that of acyclic or local data. It is important that collectors — whether local or distributed — should not unduly disrupt mutator activity. Local data is reclaimed by Modula-3's *Mostly Copying collector* (slightly modified) [1], and distributed acyclic structures are managed by the Network Objects reference listing collector [3]. We augment these mechanisms with an incremental three-phase partial trace to reclaim distributed garbage cycles. Our implementation does not halt local collectors at all, and suspends mutators only briefly. The local collectors reclaim garbage independently and expediently in each process. The partial trace merely identifies garbage cycles without reclaiming them. Consequently, both local and partial tracing collector can operate independently and concurrently.

Our algorithm operates in three phases. The first, *mark-red*, phase identifies a distributed subgraph that may be garbage: subsequent efforts of the partial trace are confined to this subgraph alone. This phase is also used to form a group of processors that will collaborate to collect cycles. The second, *scan*, phase determines whether members of this subgraph are actually garbage, before the final, *sweep*, phase makes those garbage objects found available for reclamation by local collectors. A new partial trace may be initiated by any process not currently part of a trace. There are several reasons for choosing to initiate such an activity: the process may be idle, a local collection may have reclaimed insufficient space, or the process may not have contributed to a distributed collection for a long time.

The distributed collector requires that each object has a *colour* — red or green — and that initially all objects are green. Network objects also have a *red set* of process names, akin to their client set.

3.1 Mark-Red Phase

Partial tracing is initiated at *suspect* objects: surrogates suspected of belonging to a distributed garbage cycle. We observe that any distributed garbage cycle must contain some surrogate. Suspects should be chosen with care both to maximise the amount of garbage reclaimed and to minimise redundant computation or communication. At present, we consider a surrogate to be suspect if it is not referenced locally, other than through the object table. This information is provided by the local collector — any surrogate that has not been marked is suspect.

This heuristic is actually very simplistic and may lead to undesirable wasted and repeated work. For example, it may repeatedly identify a surrogate as suspect even though it is reachable from a remote root. Only measurements of real implementations will show if this is indeed a serious problem. However, our algorithm should be seen as a framework: any better heuristic could be used. In Section 6 we show how more sophisticated heuristics improve the algorithms discrimination and hence its efficiency.

The mark-red phase paints the transitive referential closure of suspect surrogates red. Any network object receiving a mark-red request also inserts the name of the sending process into their red set to indicate that this client is a member of the suspect subgraph (*cf.* client sets)[3].

The second purpose of the mark-red phase is to identify dynamically groups of processes that will collaborate to reclaim distributed cyclic garbage. A group is simply the set of processes visited by mark-red. Group collection is desirable for fault-tolerance, decentralisation, flexibility and efficiency. Fault-tolerance and efficiency are achieved by requiring the cooperation of only those processes forming the group: progress can be made even if other processes in the system fail. Groups lead to decentralisation and flexibility as well. Decentralisation is achieved by partitioning the network into groups, with multiple groups simultaneously but independently active for garbage collection. Communication is only necessary between members of the group. Flexibility is achieved by the choice of processes forming each group. This can be done statically by prior negotiation or dynamically by mark-red. In the second case, heuristics based on geography, process identity, distance from the suspect originating the collection, or time constraints can be used.

An interesting feature of this design is that it does not need to visit the complete transitive referential closure of suspect surrogates. The purpose of this phase is simply to determine the scope of subsequent phases and to construct red sets. Early termination of the mark-red phase trades *conservatism* (tolerance of floating garbage) for a number of benefits: expediency, bounds on the size of the graph traced (and hence on the cost of the trace), execution of the mark-red process concurrently with mutators without need for synchronisation, and cheap termination of the phase. Section 4 explains how termination of mark-red is detected.

We allow multiple partial traces, initiated by different processes, to operate concurrently, but for now we do not permit groups (hence partial tracings) to overlap. Although inter-group references are permissable, mark-red is not propagated to processes that are members of other groups. One reason for this restriction is to prevent a mark-red phase from interfering with a scan phase, or vice-versa. A second reason is to allow simpler control over the size of any group: merging groups would add considerable complexity.

The example in figure 2 illustrates a mark-red process. The figure contains a garbage cycle ($yA \rightarrow yB \rightarrow yD \rightarrow xC \rightarrow yA$). Process A has initiated a partial trace; yB is a suspect because it is not reachable from a local root (other than through the object table). The mark-red process paints the suspect's transitive closure red, and constructs the red sets. In the figure, the red set of an object xX is denoted by $RS(xX)$; clear circles represent green objects and shaded ones red objects. Note that objects xD and yC are not garbage although they have been painted red: their liveness will be detected by the scan phase.

[3] Notice that cooperation from the acyclic collector and the mutator would be required if, instead, mark-red removed references from client sets or copies of client sets (see [13]). Red sets avoid this need for cooperation as well as allowing the algorithm to identify which processes have sent mark-red requests.

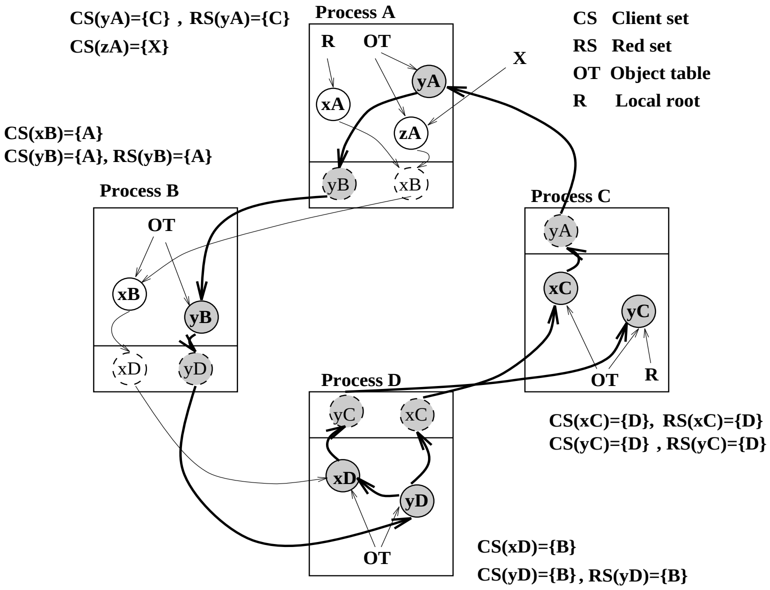

Fig. 2. Mark-Red Phase

3.2 Scan Phase

At the end of the mark-red phase, a group of processes has been formed. Members of this group will cooperate for the scan-phase. The aim of this phase is to determine whether any member of the red subgraph is reachable from outside that subgraph. The phase is executed concurrently on each process in the group. The first step is to compare the client and red sets of each red concrete object. If a red object does not have a red set (e.g. xD in figure 2), or if the difference between its client and its red sets is non-empty, the object must have a client outside the suspect red graph. In this case the object is painted green to indicate that it is live. All red objects reachable from local roots or from green concrete objects are now repainted green by a *local-scan* process. If a red surrogate is repainted green, a *scan-request* is sent to the corresponding concrete object. If this object was red, it is repainted green, along with its descendents. The scan phase terminates when the group contains no green objects holding references to red children in the group.

Continuing our example, each process calculates the difference between client and red sets for each red concrete object it holds. For instance, xD in process D has no red set so xD is painted green and becomes a root for the local-scan. Figure 3 shows the result of the scan phase: the live objects xD and yC have been repainted green.

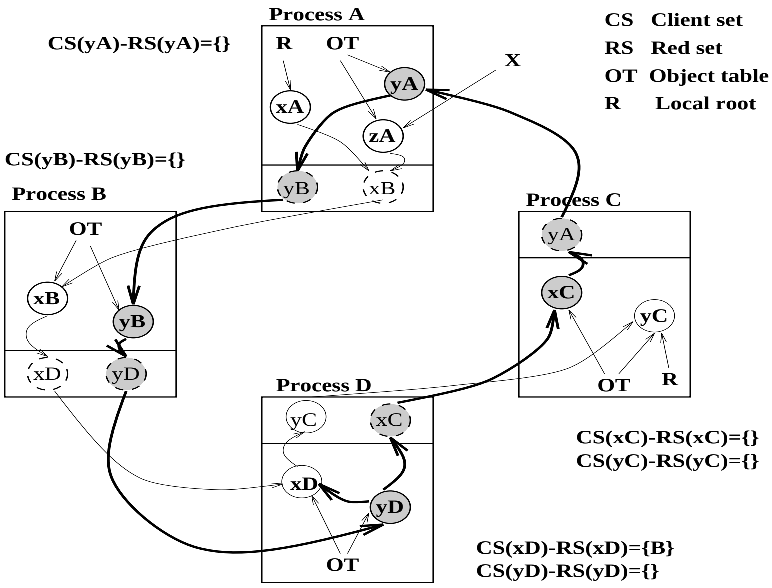

Fig. 3. Scan Phase

3.3 Sweep Phase

At the end of the scan phase, all live objects are green[4]. Any remaining red objects must be part of inaccessible cycles, and can thus be safely reclaimed. The sweep phase is executed in each process independently. Each object table is swept for references to red objects. If any are found, any local references held in the object, including references to surrogates, are deleted, thus breaking the cycle. The red descendents of these objects are now available for reclamation by the local collector. Reclamation of a surrogate causes the Networks Objects reference listing system to send a delete message to the owner of the corresponding concrete object: when its client set becomes empty, that object will also be reclaimed.

4 Synchronisation and Termination

4.1 Mark-Red Phase

By accepting that the red subgraph may include only a subset of the set of garbage objects — i.e. a conservative approximation — we gain a number of benefits including the removal of any need for synchronisation with mutators and cheaper termination.

[4] Note that the converse, *i.e.* that all green objects are live, is not necessarily true.

The solution we have currently adopted is based on that presented by Derbyshire [7]. A mark-red process is launched by the process initiating the collection. Subsequent mark-red requests export further mark-red processes. Terminating processes return an acknowledgement, identifying their process and those visited by any mark-red processes that they have exported. A process terminates when it has finished colouring its local subgraph and it has received acknowledgements for all the mark-red requests it has sent. As soon as the initiating process has received acknowledgements from all the mark-red processes that it has exported, the mark-red phase is complete and the membership of the group is known. Members of the group are then instructed by the initiating process to start the scan phase and informed of their co-members.

4.2 Scan Phase

The scan phase terminates when all members of the group have completed their scan processes and no messages are in transit. In contrast to the mark-red phase, the scan phase must be complete with respect to the red subgraph, since it must ensure that all live red objects are repainted green. As for other concurrent marking schemes (e.g. Dijkstra *et al.* [8]), this requires synchronisation between mutator and collector. Termination detection is also expensive as local-scans must be able to detect any change to the connectivity of the graph made by a mutator.

A local mutator may only change this connectivity by overwriting references to objects. Such writes can be detected by a *write barrier* [26]. We have adapted the Mostly Parallel garbage collection algorithm for the scan phase [5]. This technique uses operating system support to detect those objects modified by mutators (actually pages that have been updated within a given interval). When the local-scan process has visited all objects reachable from its starting points, the mutator is halted while the local-scan retraces the graph from any modified objects, as well as from the roots. Because most of the scanning work has already been done, it is expected that this retrace will terminate promptly (the underlying assumption is that the rate of allocation of network objects, and of objects reachable from those network objects, is low).

On termination of a local-scan, any red concrete object o and its descendents must be isolated from the green subgraph held in that process. Thus a red object cannot become reachable through actions of the local mutator. However its reachability can still be changed if:

1. a remote method is invoked on o;
2. a new surrogate in some other process is created for o;
3. another object in the same process receives a reference to o.

Notice that this scenario could only occur if the red surrogate were still alive. Although such mutator activity could be handled by the Mostly Parallel scan, this would be implementationally expensive. Instead, mutator messages are trapped by a *dirty-barrier*, a 'snapshot-at-the-beginning' barrier [26]. If a client invokes a remote method on a red surrogate, or copies the wireRep held by a red surrogate to another process, before the client's local-scan has terminated (including the receipt of acknowledgements for all the scan-requests that it has made), a scan-request is sent to the corresponding concrete object to arrive before the mutator message[5]. How this scan-request is handled depends on the state of the owner process. If the owner's local-scan has not terminated, the concrete object is repainted green and becomes an additional root for the local-scan; the scan-request is acknowledged immediately. If, on the other hand, the local-scan has terminated, the local mutator is halted and the descendents of this concrete object are repainted green as well by an *atomic-scan* process. We claim that this does not cause excessive delay as it is likely that many of its descendents will already have been repainted green. In this case, the scan-request is not acknowledged until all descendents (in the group) of the red surrogate have been painted green, if necessary by further scan-requests to other processes.

Global scan phase termination is again detected by a distributed termination detection algorithm based on Derbyshire's algorithm. A local-scan process notifies all other members of its group as soon as it has finished colouring green all objects reachable from its local roots and local green concrete objects and has received the acknowledgements for all the scan-requests it has generated (*cf.* mark-red phase). However, this account does not take the mutator actions described above into consideration. Correct termination of Derbyshire's algorithm requires that each scan-request (and subsequent scan) has a local-scan responsible for it — scan-requests from atomic-scan processes1 generated by mutator activity breach this invariant. However, trapping mutator messages with the snapshot-at-the-beginning barrier preserves the invariant. If the owner's local-scan has not terminated, it takes over the responsibility for scanning the descendents of the object. If the local-scan has terminated, the scan-request is not acknowledged until all the descendents have been scanned; the local-scan in the client process cannot terminate until it has received this acknowledgement. Notice that the mutator operation cannot have been made from a red surrogate in a process which has completed its local-scan. If the local-scan had been completed, the red surrogate would have been unreachable from the client's local roots: the action must therefore have been caused by a prior external mutator action. But in this case the surrogate would have been repainted green by an atomic-scan process. Thus the barrier suffices to ensure that any scan-request has a local-scan process ultimately responsible for it.

As before, each process informs other members of the group as soon as it has received acknowledgements from all these scan-requests. Termination of the scan phase is complete once a process has received notification of termination from each member of the group.

[5] In our implementation, we send both messages in the same remote procedure call.

5 Fault-Tolerance

5.1 Network Objects

Our algorithm is built on top of the reference listing mechanism provided by the Network Objects distributed memory manager, albeit slightly modified. The Network Objects collector is resilient to communication failures or delays, and to process failures.

Network Objects uses reference lists (actually reference *sets*) rather than reference counts. Furthermore, any given client process holds at most one reference — the surrogate – to any given concrete object. Communication failures are detected by a system of acknowledgements. However, a process that sends a message but does not receive an acknowledgement cannot know whether that message was received or not: it does not know whether the message or its acknowledgement was lost. Its only course of action is to resend the message. Unlike reference counting, reference listing is resilient to duplication of messages; the Network Objects dirty and clean operations are idempotent.

As we showed in Section 2, the dirty call mechanism also prevents out-of-order delivery of reference count messages from causing the premature reclamation of objects.

An owner of a network object can also detect the termination of any client process. Any client that has terminated is removed from the dirty set of the corresponding concrete object. Reference listing therefore allows objects to be reclaimed even if the client terminates without making a clean call. However, communication delay may be misinterreted as process failure, in which case an object may be prematurely reclaimed. Such an error will be detected when and if an attempt is made to use the the surrogate after the restoration of communication.

Proof of the safety and liveness of the Network Objects reference listing system is beyond the scope of this paper, but may be found in [3]. In the rest of this section, we shall informally discuss two aspects of the correctness of our partial tracing algorithm: safety and liveness.

5.2 Safety

First we shall establish the necessary conditions for our algorithm to be unsafe and show that these cannot arise. The safety requirement for our algorithm is that no live objects are reclaimed.

First we note that the system of acknowledgements ensures that marking requests are guaranteed to be delivered to their destination unless either client or owner process fail before the message is safely delivered and acknowledged. Although it is possible that messages might be duplicated, marking is an idempotent operation (*cf.* reference listing, above).

For the safety requirement to be breached, that is, for an object o to be incorrectly reclaimed in the sweep phase, the conditions below must hold at the end of the scan phase.

1. o must be red.
2. All members of o's client set must be members of the group.
   If this condition does not hold, then o would have been repainted green at the start of the scan phase, when o's dirty set was compared with the group membership.
3. All members of o's red set must be members of the group.
   Likewise, if this condition does not hold, then o would have been repainted green at the start of the scan phase, when o's red set was compared with the group membership.
4. All processes on at least one path of references from a root, possibly in another process, to o are still active.
   If this condition does not hold, then o is no longer reachable. Further more, the Network Objects reference listing mechanism would detect the failed process and would have deleted it from all client sets. Eventually this mechanism would reclaim o too.

From this we can conclude that o would only be prematurely reclaimed if no scan-request has traversed a path of references from a root to o, even though all processes on that path are still active. The system of acknowledgements implies that there is at least one scan-request that has not been acknowledged, and hence that there is at least one responsible local-scan process that has not terminated. But this means that the scan phase has not yet terminated.

5.3 Termination

We have argued informally above that our algorithm cannot incorrectly reclaim live objects through communication or process failures. We now show informally that each phase of the partial trace must terminate in each (non-terminating) process, and how this is achieved. First, we note that a mark-red or local-scan process will not terminate until it has received acknowledgements for all the mark-red (scan) requests that it has made. We note further that communication failures are handled by acknowledgements and the idempotency of marking requests and acknowledgements.

However, if a process with a group fails, the current phase may not terminate unless that failure is detected. If a process fails, its clients know that they need not wait for an acknowledgement from the failed process of any outstanding requests. Fortunately, the Network Objects system provides that owners detect failures of clients, if necessary reclaiming the corresponding concrete objects. We therefore provide the same system of timeouts to client processes that the Network Objects provide for owners. Thus a mark-red or local-scan process may terminate as soon as it has received acknowledgements of requests from all processes that are still active.

This leaves the scenario that a client of an object in a process p may fail, leaving p partially detached from the rest of the group. This process and its descendents may reach two undesirable states. First, any phase of the partial trace may not terminate in p before the instruction to move to the next phase arrives (through another concrete object owned by p). In this case, p must abandon this partial trace by deleting all its red sets and restoring all objects to green.

The second, and at first sight more intransigent problem, is that p may be fully detached from the rest of the group, and thus never receive the instruction to move to the next phase of the trace. Worse, not only it cannot proceed to the next phase, but the rule that a process may only collaborate in one group at a time means that it cannot participate in any group in the future! We resolve this problem by requiring that all garbage collection messages are stamped with the identity of the initiating process, *Init*. If a process has terminated a phase but the client to which it is to send the acknowledgement has failed, then responsibility for the request is inherited by *Init*, i.e. the acknowledgement is sent directly to *Init*.

The *Init* process treats such 'unexpected' acknowledgements according to when they arrive. If the acknowledgement of a request arrives before the end of the phase for which the request was issued, it is treated like any other acknowledgement: the sending process will continue to collaborate as a member of the group. If such an acknowledgement arrives at *Init* out of phase, then the sending process is effectively suspended from group membership: no further instruction to proceed to the next phase will be issued to it. Instead, at the end of the sweep phase, it will be instructed by *Init* to abandon the work that it has done, before passing this instruction on to the owners of any surrogates it holds.

If the initiating process itself fails, then the objects originally suspected of being garbage certainly are. The reference listing mechanism will propagate knowledge of *Init*'s death to all other members of the group.

6 Related Work

Distributed reference counting can be augmented in various ways to collect distributed garbage cycles. Some systems, such as Juul and Jul [14], periodically invoke global marking to collect distributed garbage cycles. With this technique, the whole graph must be traced before any cyclic garbage can be collected. Even though some degree of concurrency is allowed, this technique cannot make progress if a single process has crashed, even if that process does not own any part of the distributed garbage cycle. This algorithm is complete, but it needs global cooperation and synchronisation, and thus does not scale.

Maeda *et al.* [17] present a solution also based on earlier work by Jones and Lins using partial tracing with weighted reference counting [13]. Weighted reference counting is resilient to race conditions, but cannot recover from process failure or message loss. As suggested by Jones and Lins, they use secondary reference counts as auxiliary structures. Thus they need a weight-barrier to maintain consistency, incurring further synchronisation costs.

Maheshwari and Liskov [19] describe a simple and efficient way of using object migration to allow collection of distributed garbage cycles, that limits the volume of the migration necessary. The *Distance Heuristic*[6] estimates the length

[6] This idea was also discussed by Fuchs [10].

of the shortest path from any root to each object. The estimate of the distance of a cyclic distributed garbage object keeps increasing without bound; that of a live object does not. This heuristic allows the identification of objects belonging to a garbage cycle, with a high probability of being correct. These objects are migrated directly to a selected destination process to avoid multiple migrations. However, this solution requires support for object migration (not present in Network Objects). Moreover, migrating an object is a communication-intensive operation, not only because of its inherent overhead but also because of the time necessary to prepare an object for migration and to install it in the target process [25]. Thus, this algorithm would be inefficient in the presence of large objects.

The total overhead performed by our algorithm depends on how frequently it is run. A very simplistic heuristic may led to wasted and repeated work. However, even with a simplistic heuristic, a probability of being garbage can be assigned to each suspect object that has survived a partial tracing. For example, we could take a round-robin approach by tracing only from the suspect that was least recently traced. Better still, the Distance Heuristic should increase the chance of our algorithm tracing only garbage subgraphs. The more accurate this heuristic is, the more likely that its use would:

1. decrease the number of times a partial trace is run;
2. limit the mark-red trace to just garbage objects;
3. reduce the number of messages for the scan phase to the best case (in which no atomic-scans would be generated);
4. avoid suspension of mutators to handle scan-requests.

Lang *et al.* [16] also presented an algorithm for marking within groups of processes. Their algorithm uses standard reference counting, and so inherits its inability to tolerate message failures. It relies on the cooperation from the local collector to propagate necessary information. Firstly, they negotiate to form a group. A initial marking marks all concrete objects within the group depending on whether they are referenced from inside or from outside the group. Marks of the concrete objects are propagated locally towards surrogates. Finally marks of surrogates are propagated towards the concrete objects within the group to which they refer. When there is no more information to propagate, any dead cycles can be removed.

This algorithm is difficult to evaluate because of the lack of detail presented. However, the main differences between this and our algorithm is that we trace only those subgraphs suspected of being garbage and that we use heuristics to form groups opportunistically. In contrast, Lang's method is based on Christopher's algorithm [6]. Consequently it repeatedly scans the heap until it is sure that it has terminated. This is much more inefficient than simply marking nodes red. For example, concrete objects referenced from outside the suspect subgraph are considered as roots by the scan phase, even if they are only referenced inside the group. In the example of figures 2 and 3 our algorithm would need a total of 6 messages (5 for mark-red phase and 1 for scan phase), against a total of 10 messages (7 for the initial marking and 3 for the global propagation) for Lang's algorithm. Objects may also have to repeat traces on behalf of other objects (i.e. a trace from a 'soft' concrete object may have to be repeated if the object is hardened). Their 'stabilisation loop' may also require repeated traces. Finally, failures cause the groups to be completely reorganised, and a new group garbage collection restarted almost from scratch.

7 Conclusions and Future Work

This paper has presented a solution for collecting distributed garbage cycles. Although designed for the Network Objects system, it is applicable to other systems. Our algorithm is based on a *reference listing* scheme [3], augmented by partial tracing in order to collect distributed garbage cycles [13]. Groups of processes are formed dynamically to collect cyclic garbage. Processes within a group cooperate to perform a partial trace of only those subgraphs suspected of being garbage.

Our memory management system is highly *concurrent*: mutators, local collectors, the acyclic reference collector and distributed cycle collectors operate mostly in parallel. Local collectors are never delayed, and mutators are only halted by a distributed partial tracing either to complete a local-scan or, after that, to handle incoming messages to garbage-suspect objects; if all suspects are truly garbage, the latter event will never occur.

Our system reclaims garbage *efficiently*: local and acyclic collectors are not hindered. The efficiency of the distributed partial tracing can be increased by restricting the size of groups, thereby trading *completeness* for promptness. Appropriate choice of groups ensures completeness: eventually all cyclic garbage is reclaimable. The use of the acyclic collector and groups also permits some *scalability*, although our strategy could be defeated by pathological configurations in which a single garbage cycle spans a large number of processes.

Finally, our distributed collector is *fault-tolerant*: it is resilient to message delay, loss and duplication, and to process failure. *Expediency* is achieved by the use of groups.

Our algorithm is presently being implemented and measured. In particular, some choices for cooperation with the mutator require further study and depend mainly on experimental results and measurements. We are also interested in heuristics for suspect identification and group formation. Aspects of the concurrency and termination should be supported by formal proof and formal analysis of costs.

The management of overlapping groups is a further area for study. Currently, we prevent groups from overlapping. The distributed partial traces never deadlock, but continue their work without the cooperation of any process refusing a mark-red request. This can lead to wasted work, mainly if subgraphs span groups. We are currently exploring avenues to improve this, based on merging of groups and partitioning of work. We believe that this may also improve the algorithm's handling of failures.

References

1. Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, California, February 1988. Also in Lisp Pointers 1, 6 (April–June 1988), pp. 2–12.
2. David I. Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, pages 176–187. Springer Verlag, LNCS 259, June 1987.
3. Andrew Birrel, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for network objects. Technical report SRC 116, Digital - Systems Research Center, 1993.
4. Andrew Birrel, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. Technical report SRC 115, Digital - Systems Research Center, 1994.
5. Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.
6. T.W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.
7. Margaret H. Derbyshire. Mark scan garbage collection on a distributed architecture. *Lisp and Symbolic Computation*, 3(2):135 – 170, April 1990.
8. Edsgar W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.
9. Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS), Hong Kong*, May 27-30 1996.
10. Matthew Fuchs. Garbage collection on an open network. In *International Workshop IWMM95*, pages 251–265, Berlin, 1995. Springer Verlag, LNCS 986.
11. Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Proceedings of SIGPLAN'89 Conference on Programming Languages Design and Implementation*, pages 313–321. ACM Press, June 1989.
12. Paul R. Hudak and R.M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pa.*, pages 168–78, August 1982.
13. Richard E. Jones and Rafael D. Lins. Cyclic weighted reference counting without delay. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE'93 Parallel Architectures and Languages Europe, Munich*, Berlin, June 1993. Springer-Verlag. LNCS 694.
14. Neils-Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *Proceedings of International Workshop on Memory Management, St. Malo, France*, volume LNCS 637, DIKU, University of Copenhagen, Denmark, September 16–18 1992. Springer Verlag.
15. Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *International Conference on Distributed Computing Systems, Yokahama*, June 1992.
16. Bernard Lang, Christian Quenniac, and José Piquer. Garbage collecting the world. In *ACM Symposium on Principles of Programming, Albuquerque*, pages 39–50, January 1992.

17. Munenori Maeda, Hiroki Konaka, Yutaka Ishikawa, Takashi TomoKiyo, Atsushi Hori, and Jorg Nolte. On-the-fly global garbage collection based on partly mark-sweep. In *Proceedings of International Workshop on Memory Management, Kinross, UK*, Tsukuba Research Center, Japan, September 27–29 1995. Springer Verlag. LNCS 986.
18. U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server objected-oriented database. In *Proceedings of the third International Conference on Parallel and Distributed Information Systems*, pages 239–248, September 1994.
19. U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of the Principles of Distributed Computing*, 1995.
20. J. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In Aarts *et al.*, editor, *PARLE'91 Parallel Architectures and Languages Europe*, Berlin, 1991. Springer Verlag, LNCS 505.
21. David Plainfossé and Marc Shapiro. Experience with fault-tolerant garbage collection in a distributed Lisp system. In *Proceedings of International Workshop on Memory Management, St. Malo, France*, INRIA, France, September 16–18 1992. Springer Verlag. LNCS 637.
22. David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management, Kinross, UK*, INRIA, France, September 27–29 1995. Springer Verlag. LNCS 986.
23. M. Shapiro, P. Dickman, and D. Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proceedings of the Symposium on Principles od Distributed Computing*, 1992.
24. Marc Shapiro, O. Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapports de Recherche 1320, INRIA-Rocquencourt, November 1990. Also in ECOOP/OOPSLA'90 Workshop on Garbage Collection.
25. N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12):33–44, December 1992.
26. Paul R. Wilson. Garbage collection and memory hierarchy. In *Proceedings of International Workshop on Memory Management, St. Malo, France*, volume LNCS 637, University of Texas, USA, September 16–18 1992. Springer Verlag.
27. Paul R. Wilson. Distr. gc general discussion for faq. gclist (gclist@iecc.com), March 1996.