Cyclic Distributed Garbage Collection with Group Merger

Helena Rodrigues* and Richard Jones

Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK
Tel: +44 1227 827943, Fax +44 1227 762811
email: {hccdr,R.E.Jones}@ukc.ac.uk
http://www.cs.ukc.ac.uk/people/staff/rej/

Abstract. This paper presents a new algorithm for distributed garbage collection and outlines its implementation within the Network Objects system. The algorithm is based on a *reference listing* scheme augmented by *partial tracing* in order to collect distributed garbage cycles. Our collector is designed to be flexible thereby allowing efficiency, expediency and fault-tolerance to be traded against completeness. Processes may be dynamically organised into groups, according to appropriate heuristics, in order to reclaim distributed garbage cycles. Unlike previous group-based algorithms, multiple concurrent distributed garbage collections that span groups are supported: when two collections meet they may either merge, overlap or retreat. 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.

Keywords: distributed systems, garbage collection, termination detection

1 Introduction

With the continued growth of distributed systems, designers are turning their attention to garbage collection [35, 30, 24, 22, 23, 7, 26, 27, 25, 14, 31, 15, 33, 20, 28, 18], prompted by the complexity of memory management and the desire for transparent object management. The goals of an ideal distributed garbage collector are that:

safety: only garbage should be reclaimed.
completeness: all garbage, including distributed cycles, at the start of a collection cycle should be reclaimed by its end.
concurrency: neither mutator nor local collector processes should be suspended; distinct distributed collection processes should run concurrently.
promptness: garbage should be reclaimed promptly.
efficiency: time and space costs should be minimised.

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

locality: inter-process communication should be minimised.

expediency: garbage should be reclaimed despite the unavailability of parts of the system.

scalability: it should scale to networks of many processes.

fault tolerance: it 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. For this reason we add a further goal:

flexibility: the collector should be configurable, guided by heuristics or hints from either the programmer or compiler.

Distributed garbage collection algorithms generally follow one of two strategies: tracing or reference counting. Tracing algorithms visit all 'live' objects [17, 13]; 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 [6, 39, 16, 29, 36, 7, 26].

On the other hand, reference counting algorithms cannot collect cycles of garbage, although cyclic connections between objects in distributed systems are fairly common. For example, objects in client-server systems may hold references to each other, and often this communication is bi-directional [40]. 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 [27]. Although inter-process cycles of garbage can be broken by explicitly deleting references, this leads to exactly the error-prone scenario that garbage collection replaces.

Systems using distributed reference counting as their primary distributed memory management policy must reclaim cycles by using a complementary tracing scheme [22, 24, 21, 25, 33, 28, 18], 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 [35, 27]. However, migration is communication-expensive and existing complementary tracing solutions require global synchronisation and the cooperation of all processes in the system [22], place additional overhead on the local collector and application [25], rely on cooperation from the local collector to propagate necessary information [24], or are not fault-tolerant [24, 25].

This paper presents an algorithm and outlines its implementation for the Network Objects system [8]. A fuller description and a proof of its correctness is to be found in [34]. Our algorithm is based on a *reference listing* [7], augmented by *partial tracing* in order to collect distributed garbage cycles [21, 33]. Our algorithm preserves our primary goals of efficient reclamation of local and distributed acyclic garbage, low synchronisation overheads, and avoidance of global synchronisation. In brief, our aim is to match rates of collection against rates of allocation of data structures. Objects only reachable from local processes have very high allocation rates, and 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.

To these ends, we permit some degree of completeness and efficiency in collecting distributed cycles to be traded, although eventually all these cycles will be reclaimed. We use heuristics to form groups of processes *dynamically* that co-operate to perform partial traces of subgraphs suspected of being garbage. Our earlier work offered only limited support for multiple, independently-initiated distributed garbage collections, as we imposed the restriction that no two distributed garbage collections could overlap; that is, no object could be simultaneously a member of more than one group and hence subject to more than one garbage collection [33]. This restriction prevented the collection of garbage cycles that spanned groups. In this paper, we lift this restriction and furthermore offer considerable flexibility to the programmer/compiler over how groups interact.

The paper is organised as follows. Section 2 introduces the computational model: the distributed system, mutator processes, visibility of objects across the network, reference passing and liveness. Section 3 introduces our partial tracing algorithm before Sect. 4 describes multiple, independently initiated, distributed garbage collections and deals with the problem of cycles that span groups. Section 5 introduces the problems of concurrency between mutators and collectors, and explains how the collectors are synchronised and termination achieved. Section 6 outlines a proof of correctness, and Sect. 7 maps our abstract description of our collector onto a concrete implementation using Modula-3's Network Objects system. Section 8 identifies the parameters that determine the cost of our algorithm and discusses how heuristics may be used to tune the collector. Finally we discuss related work in Sect. 9, and conclude in Sect. 10.

2 Computational Model

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, $\ldots$, and objects by lower-case letters (subscripted by the identifier of their owner), e.g. x_A, x_B, $\ldots$

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 *garbage*, to be reclaimed by a *collector*. A collector that operates solely within a local heap is called a *local collector*.

For the moment, we abstract away from the details of the implementation by considering each process to maintain two tables. The *in-table* of a process lists all the remotely referenced *in-objects* belonging to the process. Only in-objects may be shared by processes. The process accessing an in-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. Objects cannot migrate from one process to another.

A client cannot directly access an in-object but can only invoke the methods of a corresponding *out-object*, which in turn makes remote procedure calls to the owner. Associated with each entry in an in-table is a reference list, or *client set*, of the processes holding out-objects for the in-object. The *out-table* of each process lists all its out-objects and the remote in-objects to which they refer. A process holds at most one out-object for a given in-object and all references in the process to the remote object point to the corresponding out-object.

The heap of a process is managed by garbage collection. Local collections are based on tracing from process roots — the stack, registers, global variables and also the in-table. The in-table is considered a root by the local collector in order to preserve objects reachable only from other processes. In-table entries are managed by the distributed memory manager.

Remote references may be deleted or copied from one process to another either as arguments or results of methods. If the process receiving a reference is not the owner of the in-object, then the process may need to create a local out-object. In order to marshal a reference to another process, the sender process needs either to be the owner of the object or to have a out-object for that object. This operation must preserve a key invariant: whenever there is a out-object for an in-object x_P belonging to owner P at client C, then $C \in x_P.clientSet$.

Out-objects unreachable from their local root set are reclaimed by local collectors, in which case the corresponding owner is informed that the reference should be removed from its client set. When an in-object's client set becomes empty, the object is removed from the in-table so that it can be reclaimed subsequently by its owner's local collector. The invariants necessary to avoid race conditions and prevent premature reclamation of in objects are maintained in the standard way [7].

3 The Basic Algorithm

Our algorithm is based on the premise that distributed garbage cycles exist but are less common than acyclic structures. Thus reclamation of distributed cyclic garbage may be performed more slowly than that of local or distributed acyclic data. One consequence is that it is important that collectors — whether local or distributed — should not unduly disrupt mutator activity. We rely on local data being reclaimed by a tracing collector [20], whilst distributed acyclic structures are managed by reference listing [7]. 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. 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. To simplify exposition, we start by describing the basic mechanisms, restricting our discussion to the collection of garbage within a single group of cooperating processes. We add multiple, independent but co-operating, distributed collectors in Sect. 4 and discuss concurrency and termination in Sect. 5.

Our algorithm operates in three phases [11, 33]. The first, *mark-red*, phase identifies a distributed subgraph that may be garbage, to which subsequent efforts are confined. The mark-red phase also identifies *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. 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.

The second, *scan*, phase determines whether members of this subgraph are actually garbage. This phase must also detect that any other collections upon which this collection depends have also terminated. Finally the *sweep* phase makes any garbage objects available for reclamation by local collectors.

The distributed collector requires that each item in processes' in- and out-tables has a *colour* — red, green or none — and that initially all objects are uncoloured (i.e. colour 'none'). In-objects also have a *red set* of process names, akin to their client set.

Partial tracing is initiated at *suspect* objects: out-objects suspected of belonging to a distributed garbage cycle (any distributed cycle must contain some out-object). 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, the process may not have contributed to a distributed collection for a long time, or the process may simply choose to start a new distributed collection whenever it discovers a suspect object. Suspects should be chosen with care both to maximise the amount of garbage reclaimed and to minimise redundant computation or communication. A naïve view is to consider an out-object to be suspect if it is not referenced locally, other than through the in-table. This information is provided by the local collector — any out-object that has not been marked is suspect. This heuristic is very simplistic and may lead to undesirable wasted and repeated work. For example, it may repeatedly identify an out-object as a suspect even though it is reachable from a remote root. Rather, our algorithm should be seen as a framework: any better heuristic could be used [26]. In Section 8 we show how more sophisticated heuristics improve the algorithm's discrimination and hence its efficiency.

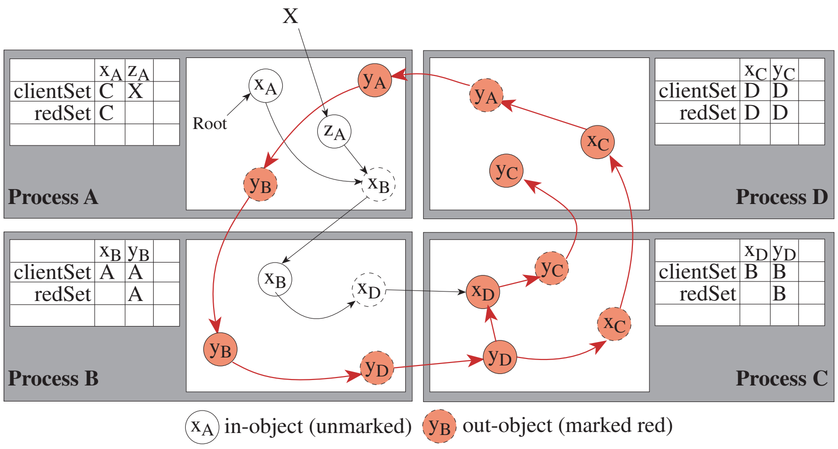


Fig. 1. mark-red identifies a subgraph suspect of being garbage

The **mark-red phase** paints the transitive referential closure of suspect out-objects red. It proceeds by a series of alternating local and remote steps. A *local step* forwards a colour from an object i in a process' in-table to all objects in its out-table reachable from i. A *remote step* sends a request from an out-table object to its corresponding in-table object, in this case reddening the in-object and inserting the name of the sending process into the red-set to indicate that this client is a member of the suspect subgraph[1]. Thus red-sets can be thought of as the 'dual' of client-sets: client-sets list all references to an in-object but red-sets list only those references believed to be dead.

The example in Fig. 1 illustrates a mark-red process. The figure contains a garbage cycle: $y_A y_B y_D x_C y_A$. Process A has initiated a partial trace; y_B is a suspect because it is not reachable from a local root (other than through the

[1] 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 [21]). Red sets avoid this need for cooperation as well as allowing the algorithm to identify which processes have sent mark-red requests.

in-table). The mark-red process paints the suspect's transitive closure red, and constructs the red sets. Note that objects x_D and y_C are not garbage although they have been painted red: their liveness will be detected by the scan phase.

At the end of the mark-red phase, a group of processes has been formed that 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. It is executed concurrently on each process in the group. The first step is to compare the client- and red-sets of each red in-table object. If does not have a red-set (e.g. x_D in Fig. 1), or the difference between its client- and 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 may be live. Again, the scan phase proceeds by a series of alternating local and remote steps. All red in- and out-table objects reachable from local roots or from green in-table objects are now repainted green by a local step. A remote step sends a scan-request from each out-table object repainted green to its corresponding in-table object. If this object was red, it is also repainted green. The scan phase terminates when the group contains no green objects holding references to red children within the group.

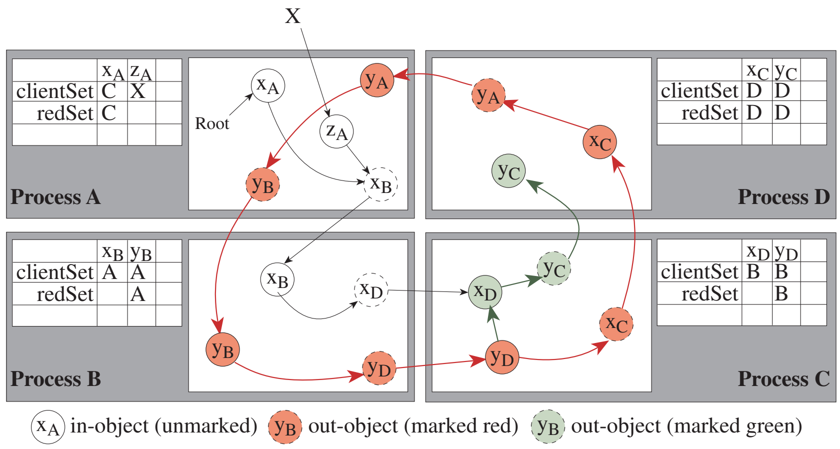


Fig. 2. the scan phase 'rescues' any red objects that may be live

Figure 2 shows the result of the scan phase. x_D in process D has no red-set so is painted green and becomes a root for the local step. y_C is reachable from x_D and so is also repainted green[2]. Note that the only inter-process edge traversed in the scan phase is that between the y_C.

[2] Notice that other group-based partial tracing schemes do not consider public objects internal to the group to be roots [24]. In our example that would require extra messages to be sent from A to B and from B to D in order to preserve x_D.

At the end of the scan phase, all live objects are green[3]. 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: at the next local collection, red in-table objects are not considered to be roots, and thus their (garbage) descendents will be reclaimed. The reclamation of an out-table item causes the reference listing mechanism to send a delete message to the owner of the corresponding in-table object: when its client-set becomes empty, that object will also be reclaimed.

4 Multiple Group Collection

Very few studies have measured the performance of distributed garbage collection algorithms and behaviour of the programs they support. In particular, comparatively little is known about the topology or demographics of distributed object systems —for example, how common are distributed cycles, how large are they, how long lived are they? A deficiency of many proposals for group-based distributed collectors, including our earlier work [33], is the treatment of inter-group garbage cycles.

Our new algorithm allows different collecting groups to cooperate for garbage collection. Scalability demands that distributed garbage collections may be initiated independently, but this raises the possibility that two independently initiated groups may meet in one or more processes. There are two ways in which distributed structures, hence groups identified by mark-red, may overlap. First, a *process* may be a member of more than one group despite there being no reference from any object in one group to any object in any other group. Consequently no object will visited by more than one group. Alternatively, an *object* may be referenced by objects in more than one group. It is this more interesting and challenging alternative that we address now; the simpler problem is also solved by our algorithm.

There are three possible strategies for resolving this matter. First, all interaction between two independent distributed garbage collections could be prohibited whilst nevertheless permitting inter-group references [33]. This has the advantage of simplicity as it eliminates all interaction between distributed collectors, and obviates any need for synchronisation either to assure correctness and termination, or to bound the size of a collection. However, it fails to collect garbage cycles that span groups.

The second strategy is to allow both collections to proceed, but to ignore one another. In effect, the groups retain their own identity but *overlap*. This requires that the collectors do not share any state (the colour and red-set information held in the in- and out-tables). This could be achieved by maintaining one copy of this state information for each collection group, and having all garbage collection messages signed with the identity of their group (i.e. the identity of their initiating process). The obvious drawback is that, while this is scalable and complete, it is neither time- nor space-efficient as it leads to repeated work.

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

The third strategy is to *merge* the two collecting groups into a single group, thereby giving completeness and efficiency albeit at the cost of greater complexity. To collect garbage data structures that span two groups, some form of synchronisation must exist between the groups. One group maybe be *dependent* on the other, and unable to determine that the structure it is holding is garbage until the other has also determined that its portion of the structure is garbage. In the example in Fig. 3, the group containing processes A and B cannot detect that its structure is garbage until the CD group has completed its scan phase.

4.1 Partial Tracing Objects

Our algorithm records this dependency information explicitly. Each in- and out-object holds (in addition to the colour) a *marks* list of groups that have visited the object; the head of this list is called the *mark*.

The Network Objects library handles all communication between network objects through a special object in each process [8]. We adopt the same approach to support our partial tracing mechanisms by constructing a new partial tracing object (pto) PT_{id} when a collection for group *id* visits a participant process for the first time in this collection cycle:

$$PT_{id} = (\textit{id, participants, ins, outs, guardians, dependents})$$

id is a unique identifier. A distributed garbage collection can be identified by its initiating process and the set of suspect objects from which it starts. For simplicity we shall usually assume *id* and the initiator to be synonymous.
participants the members of the group collaborating to collect garbage.
ins, outs in- and out-objects in this process visited by this group.
dependents pto's that are dependent on this object.
guardians pto's that are guardians of this object.

For convenience, we denote the colour of an object with mark A by red_A, $green_A$, etc. Most communication between groups is handled through these local pto's (ambassadors, maybe?) without exchanging messages across the network.

4.2 Merging Mark-red

The mark-red phase is initiated from a suspect out-object (e.g. v_C in Fig. 3) by creating a pto identified by this process, say D. This *initiating pto* is said to be *active-disquiet*. The suspect is reddened and its *mark* set to PT_D. The pto then executes alternate local and remote steps, colouring objects that it reaches. It performs a local mark-red step from each in-object i newly marked red_D to colour each out-object o reachable from i as follows:

(ML.1) If o has not been coloured, then it is reddened and its mark set to PT_D: it is red_D.
(ML.2) If o is already red_D, then no further action is necessary.

(ML.3) If o is red_A and $A \neq D$, then two groups have met in the same phase. We merge the groups and say that A is *dependent* on D and D is a *guardian* for A. PT_D is appended to $o.marks$, PT_D is added to the *guardians* set of the pto PT_A, and PT_A to the *dependents* set of pto PT_D. Both these interactions take place between the pto's in this process — no messages are sent.

(ML.4) If o is green, it must have been marked by another group operating in a later phase so the red wave-front retreats from this object.

Remote steps executed by PT_D propagate colours from out-objects o in a process P to in-objects i in a remote process Q. A new PT_D pto is constructed in Q to represent this group (unless one already exists for this group as a result of an earlier mark-red request in this collection cycle).

(MR.1) If i is uncoloured or red_D, P is added to its red set and i is marked red_D.

(MR.2) If i is red_A and $A \neq D$, P is still added to i's red set. Once again two groups have met and, as in the local step, PT_D is appended to $i.marks$ and to $PT_A.guardians$, PT_A to $PT_D.dependents$ in process Q; no messages are exchanged.

(MR.3) If i is green, no further action is taken.

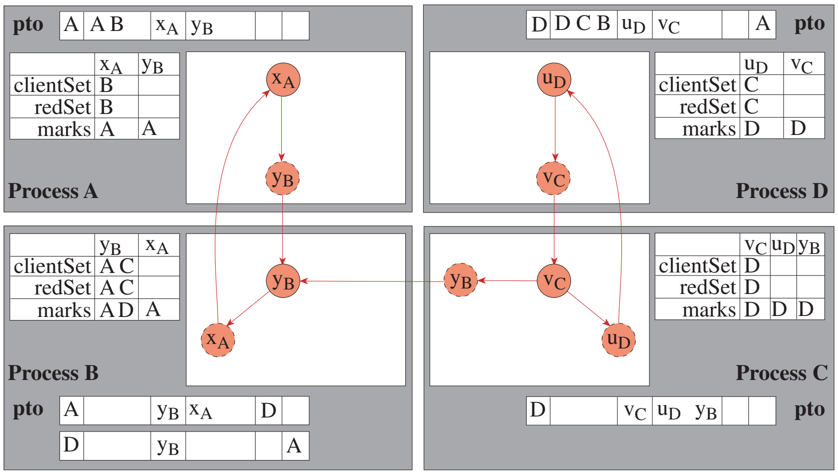


Fig. 3. end of the mark-red phase

When a pto has no more local steps to perform and has received acknowledgements for all the tracing requests that it has sent, the pto returns an acknowledgement to the pto whose remote step caused it to be created. The acknowledgement contains a list of the participating processes that it and its children have visited. It is now said to be *passive-quiet* but may be re-awakened by further mark-red requests (in which case it becomes *passive-disquiet*). No further synchronisation is needed between the mark-red phases of each group[4]. At the end of the mark-red phase, each initiating pto will know the participants in its group, and objects in these participant processes reachable from suspects will have been painted red (with their *marks* list identifying all the groups of which they are members). Figure 3 shows an example in which two processes, A and D, have initiated independent distributed collections which have met at y_B. Note that process B contains pto's for both groups A and D.

4.3 Scanning Merged Groups

The aim of the merging collector is now to identify those objects that are not reachable from a root or from outside the merged super-group. A component group cannot make such a decision in isolation. Thus each initiating pto must determine that (a) it and all the participants in its group, and (b) all the groups upon which it depends, have completed their scan phase.

On termination of its mark-red phase, the initiator instructs all participants in its group to start the scan phase. In this phase, each pto will 'rescue' any live objects that it had inadvertently marked red. Again, after an initial step to colour green any in- or out- object reachable from the local roots, the scan phase proceeds by an alternating series of local and remote steps. The roots of the scan phase for a pto PT_A are:

- the process' roots (stack, registers, static area...),
- any in-object that is not red,
- any red in-object marked either by PT_A or any group B responsible for it — i.e. $B \in PT_A.guardians$ — whose client and red sets differ (i.e. there is a path to this object that mark-red has not traversed), and
- any other red in-objects marked by other groups.

For example, in Fig. 1 x_D is a scan root, but in the example in Fig. 3 there are no scan roots. The initial scan step of each pto PT_A greens any objects directly reachable from the root set that it had previously visited: these will be the starting points for the 'rescue' trace.

(SI.1) Mark green any red in- or out-object x for which $x.mark = PT_A$ (x had been visited by a mark-red request from PT_A) that is in, or reachable from, the root set.

The local scan phase step for PT_A propagates the green colour from a $green_A$ in-object i to those out-objects o in the same process reachable from i that PT_A had previously visited in the mark-red phase:

(SL.1) Green o if it is *red* and $A \in o.marks$.

The remote step from a $green_A$ out-object o propagates the green colour to the corresponding in-object i:

[4] The termination of each phase is discussed in more detail in Sect. 5.

(SR.1) If i is red and $PT_A \in i.marks$, mark i green.
(SR.2) If i is red but $PT_A \notin i.marks$, retreat.
(SR.3) If i is not red, retreat.

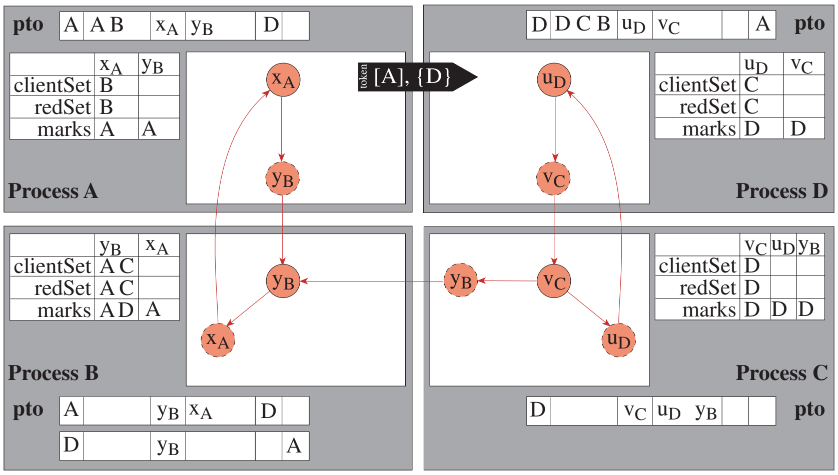


Fig. 4. end of the scan phase

Remote steps do not invoke local steps directly. Rather, the pto that 'owns' the in-object (identified by its *mark*) will execute a local step once it has started its scan phase. Note that an in-object may be part of more than one group (the length of its *marks* list is greater than one). If a pto receives a scan-request before it receives the instruction to start the scan phase, it simply marks the in-object green but does not yet take a local step.

As before, a pto PT_A becomes passive-quiet in the scan phase when it has no more local steps to perform and has received acknowledgements for all the remote steps that it has executed. It returns to its parent tracing object a list of all the groups upon which it is dependent, $PT_A.guardians$[5].

$$PT_A.guardians = \bigcup_{i.mark=A} i.marks$$

Figure 4 shows the example at the end of the scan phase. PT_A in process B has reported to the process A that initiated the collection that the collection is dependent on D.

4.4 Sweep Phase

The scan phase of a group cannot terminate as long as it is possible for a member of that group to receive further scan requests. We describe our termination

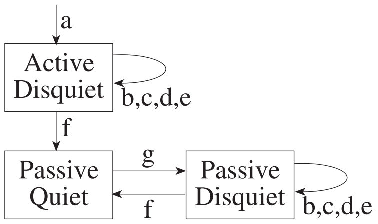
[5] A group A may be a guardian for group B and vice-versa.

mechanism in Sect. 5.4 below. At the end of the scan phase, any red_A objects are unreachable from process roots either within or without this group. First we give the single step taken by the sweep phase for PT_A in each process concurrently:

(SW.1) Remove all red_A from the in-table; repaint as uncoloured all $green_A$, setting their *marks* lists to empty.

5 Concurrency and Termination

The algorithm requires that each initiator of a distributed collection detect the termination of its mark-red and scan phases, and that it detects the termination of any collections upon which it is dependent. We approach these two problems separately.



Events	
a	start phase
b	send *mark-request(out-object)*
c	receive *mark-request(out-object)*
d	receive *ack(out-object)*
e	perform *local step*
f	all acks received
g	receive *mark-request(out-object)*

Fig. 5. state transitions for termination detection. An additional initial state required by pto's that receive a scan-request before the instruction to start the scan phase is omitted for simplicity.

5.1 Group Termination

For termination of each group, we use an acknowledgement-based termination detection protocol for both the mark-red and the scan phases that does not require processes to be known at the start of distributed garbage collection [38]. Following Augusteijn [2], we introduce three possible states for a process (see Diagram and Table 5): *active-disquiet*, *passive-disquiet* and *passive-quiet*.

- A process initiating a phase is active-disquiet.
- The receipt of a tracing request causes a passive-quiet process to become passive-disquiet.
- Requests have no effect on the *state* of an active-disquiet process — the request is simply added to the process' work-list.
- When a process has no more work to do — no local steps to perform and it has received acknowledgements for all its tracing requests — the process becomes passive-quiet.

- On becoming passive-quiet, processes return an acknowledgement, identifying themselves and those identified by any tracing requests that they have in turn exported. Requests to active-disquiet processes on the other hand are acknowledged immediately[6].

From this it can be seen that there is always an active-disquiet process responsible for any passive-disquiet process. It therefore suffices to detect the termination of the active processes alone. In the mark-red phase, this is particularly simple as there is just one active process: the initiator; in the scan phase however there may be many active disquiet processes.

5.2 Mark-red Phase

Within a single group, the mark-red phase is initiated by a single active-disquiet process. As soon as this process has received acknowledgements from all the mark-red processes that it has exported, it becomes passive-quiet: the mark-red phase is complete and the membership of the group is known.

Our collector does not need to visit the complete transitive referential closure of suspect out-objects. The purpose of the mark-red phase is simply to determine the scope of subsequent phases and to construct red-sets. Early termination of this phase can be used to trade *conservatism* (tolerance of floating garbage) for expediency, and bounds on the size of the graph traced (and hence on the cost of the trace). We believe that our approach also shows promise for other NUMA problems that use partitioned address spaces, such as distributed object-oriented databases and persistent storage systems [34]. Importantly, this conservativism allows the phase to be executed concurrently with mutators without need for synchronisation and so permits cheap termination of this phase.

5.3 Scan Phase

The scan phase is initiated concurrently on each participant process holding part of the red, suspect subgraph. The pto's representing this group collection in each participant process are set to active-disquiet on receipt of the instruction from the initiating pto to start this phase. Group scan phase termination is detected by the protocol described above. Each of the active-disquiet pto's for this group informs the initiating pto as soon as it becomes passive-quiet.

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 with other concurrent marking schemes, this requires synchronisation between mutator and collector. We consider the effect of mutator actions on first scan phase local steps and then scan phase remote steps.

Termination detection requires that scan phase local steps 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.

[6] We piggy-back collector acknowledgements on the back of RPC acknowledgements.

Such writes can be detected by a *write barrier* [41] — our implementation is described in Sect. 7. Once a process has no more local scan steps to perform, any red in-object o and its red descendents are isolated from the green subgraph held in that process — they cannot become reachable through actions of the local mutator. However their reachability can still be changed if:

(a) a remote method is invoked on o; or
(b) a new out-object corresponding to o is created in some other process;
(c) another object in the same process receives a reference to o from a remote process.

None of these events can occur unless the red out-object is still alive. Correct termination detection requires that each scan-request (and subsequent scan) has an active-disquiet process ultimately responsible for it. Trapping mutator messages with a 'snapshot-at-the-beginning' barrier preserves this invariant. If a client invokes a remote method from a red out-object, or copies the reference held by a red out-object to another process, a scan-request is sent to the corresponding in-object, along with the mutator message[7]. The scan-request paints the in-object, and any local out-objects reachable from it, green in an atomic local-step operation before the mutator message is handled.

The mutator operation that sprung the trap cannot have been made from a passive-quiet process. If it were, the red out-object would have been unreachable from the client process' roots unless a prior external mutator action had caused the object to become locally reachable. But in this case the out-object would have been repainted green by the write barrier. Thus, an active-disquiet process is always responsible for the scan-request generated by the barrier. If the owner of the in-object is also active-disquiet, an acknowledgement is returned immediately and this process takes over responsibility for any consequent scan-requests. If it is passive, the scan-request is not acknowledged until all the descendents have been scanned; the client process cannot become quiet until it has received this acknowledgement.

5.4 Super-group Termination

The scan phase of an individual group cannot terminate as long as it is possible for a member of the group to receive further scan requests. Our modification of Augusteijn's algorithm resolves this for members of a single group, but a group may also receive scan-requests from members of other groups in its super-group. We note however that there will be an active-disquiet process responsible for these requests, and say that a process is *stable* if it is not active-disquiet. Once a process becomes stable it can never become active-disquiet again: although it may perform scan steps these will be on behalf of other active-disquiet processes. We say that a group is *partially terminated* if all its participants are stable. Our termination property for a single group is that all groups (initiating pto's) on which it depends are partially terminated. We define a relation *Dependent*:

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

Definition 1. *$\forall$ pto's PT_A, PT_D in a process, Dependent$(PT_A, PT_D) \equiv PT_D \in PT_A.guardians$*

and we calculate its reflexive transitive closure, *Dependent**. We adopt the simple protocol of passing tokens around a ring formed by initiator members of the super group [32], so that when a token has returned to the initiator that created it, the scan phase of that group is complete. As soon as an initiator A partially terminates, it constructs a token. The token has two parts:

terminated a list of the groups in ring that are known to have partially terminated; initially this holds A alone.
next a set of initiators not yet visited; initially this holds the groups responsible for A, $A.guardians$.

Propagation of the token around the ring is simple. An initiator process A retains the token until all members of its group are stable, i.e. the group is partially terminated. If the head of the token's *terminated* list is A then the scan phase has fully terminated. Otherwise A (i) removes itself from the *next* set to the end of to the *terminated* list, (ii) inserts any of its guardian groups that are not members of the *terminated* list in the *next* set, and (iii) passes the token to any member of the *next* set. If this set is empty, all the *Dependent*(A)* groups have terminated and the token is returned to its owner, the head of the *terminated* list. Figure 4 shows the token sent by A to its guardian, D; D will return the token with an empty *next*-set to the head of the *terminated*-list, A. As D has an empty *guardians* set, it does not need to wait for any other group to terminate.

6 Safety

The safety requirement for our algorithm is that live objects are never reclaimed. First we note that the system of acknowledgements ensures that marking requests are guaranteed to be delivered to their destination unless either the client or owner process fails 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).

To demonstrate that the merging algorithm is correct, we briefly outline how it can be shown that, if a pto PT_B is in its sweep phase, then no red_B objects in the same process can receive a scan request, and hence that no red_B object can be live in B's sweep phase. First, we conservatively define an object x to be *live* if

$$(\exists P \in supergroup \wedge \exists r \in Roots(P) \wedge path(r,x)) \quad \vee$$
$$(\exists Q \notin supergroup \wedge \exists o \in out\text{-}table(Q) \wedge path(o,x))$$

Suppose that x is live but erroneously reclaimed by pto PT_B in process P. By (SW.1), $red_B(x) \wedge live(x)$. Thus

$$(\exists i \in in\text{-}table(P) \wedge path(i,x) \wedge live(i)) \vee (\exists r \in Roots(P) \wedge path(r,x))$$

There cannot have been such a path from a local root before PT_B took its initial scan step — since (SI.1) would have greened x — so only a subsequent remote method invoked from an out-object o' on an in-object i' from which x is reachable could have created this path. If o' was red_B, x would have been repainted green by o''s barrier. If it was $green_B$, a scan request has been sent to repaint i and hence x green (if the scan-request acknowledgement has not been received then the request will be sent again, with the mutator message). If o' had not been visited by B, then i would have been a scan root for PT_B.

So x must have been reachable from i when PT_B took its initial scan step, and this i cannot have been a local scan root (SI.1). Hence

$$red(i) \wedge (i.redSet = i.clientSet) \wedge (i.mark = PT_B \vee i.mark \in PT_B.guardians)$$

All out-objects o (i.e. $o \in i.clientSet$) from which i is reachable must be red (MR.1 or MR.2), and by hypothesis, at least one such red_A, for some group A, must be live. We need to show that group A has completed its scan phase and hence that o can never become green.

If $A = B$ then the pto's responsible for both x and o are members of the same group. Hence o's pto has completed its scan phase and so cannot generated further scan requests. Alternatively, $A \neq B$ in which case $A \in PT_B.guardians$ (MR.1 or MR.2) and hence a member of the *guardians* set of the initiator of group B (the final action of a pto in the scan phase is to return a list of guardian groups to its initiator). The scan phase termination for group B must send a token to, *inter alia*, the initiator of group A (since $A \in PT_B.guardians$). Group B does not enter the sweep phase until A (and other guardian groups) have returned the token, but group A will not do so until all its members are passive-quiet. No scan request can be generated from within group A.

Neither can a scan request originate from a group in *Dependent*(A)* as all pto's within *Dependent*(A)* are partially terminated by the time that A has received its token back. Any request must be from an out-object o'' in a third group, $C \notin$ *Dependent*(A)*. Since the in-object was red, its red set contained o'' and hence its *marks* contained C, i.e. C is a guardian for group B. But this means that $C \in$ *Dependent*(A)*. Thus no such scan request can occur. Hence group A has completed its scan phase and the red objects cannot be live.

7 Mapping the Algorithm onto Network Objects

Our algorithm is built on top of the reference listing mechanism provided by the Network Objects distributed memory manager, albeit slightly modified [8]. The Network Objects collector is resilient to communication failures or delays, and to process failures. Object migration is not supported. In this section we describe how our algorithm is mapped onto the Network Objects system. In particular we are bound to account for the collection of local and acyclic distributed garbage, and synchronisation between mutators and collectors

Network Objects is a distributed object library for Modula-3, a garbage-collected language [10]. Our local collector is a slightly modified version of the

SRC Modula-3 incremental, mostly copying collector [4]. Synchronisation between the mutator and the local collector is provided by a page-wise read-only barrier supported by the operating system [1].

Network Objects uses reference lists rather than counts: any client process holds at most one *surrogate* for any given network, or *concrete*, object. Our in-table is represented by that part of (a modified version of) the Network Objects' *object table*, that contains references, or *wireReps*, to concrete network objects, and our out-table is that part that contains references to surrogate objects.

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. Unlike reference counting, reference listing operations are idempotent and so resilient to duplication of messages. Network Objects' dirty call mechanism also prevents out-of-order delivery of 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 client set of the corresponding concrete object, allowing objects to be reclaimed even if the client terminates abnormally. Unfortunately, communication delay may be misinterpreted as process failure, in which case an object may be prematurely reclaimed. Proof of the safety and liveness of the Network Objects system may be found in [7].

Unlike the mark-red phase, scan phase tracing must be accurate with respect to the red subgraph in order to ensure that it reaches all red objects that are live. Mostly Parallel garbage collection [9] 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 phase process has visited all objects reachable from its starting points, the mutator is halted while the graph is retraced from roots any modified objects. 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). In any event, this retrace may be interrupted and restarted later.

Scan-requests caused by mutator action are asynchronous and these may require the out-object descendents of the receiving in-object to be repainted green atomically. The simplest method of propagating marks from in- to out-objects is to 'stop the world' in that process and perform a standard recursive trace from the in-object. We claim that this does not cause excessive delay as this event is unlikely to occur if our heuristic for finding suspects is good, and moreover it is likely that objects reachable from a live in-object are already known to be live.

8 Costs and Heuristics

The costs associated with our algorithm can be divided into two categories: those associated with the RPC calls exchanged between processors and those common to any incremental collector caused by running scan phase local steps concurrently with mutators. We analyse the former here.

Call the number of inter-process edges in the subgraph visited by mark-red e, and the number of participants in this group n. Note that $e \leq$ the number of edges in the transitive referential closure of the suspect objects.

- The mark-red phase for each group issues e mark-request RPC calls, by definition.
- The number of mark-red acknowledgement calls depends on whether the request is sent to a quiet or a disquiet pto, and this in turn depends on the topology (degree of sharing) of the subgraph. An acknowledgement from a disquiet pto can be piggy-backed onto the RPC acknowledgement; that from a quiet process requires a separate call. Thus, between $n-1$ (one acknowledgement for each pto-creating request) and e (one per edge) calls are required.
- Each acknowledgement message has a length $\leq n$, the maximum number of processes to which the request message can have been forwarded.

Thus the number of RPC calls $\mathcal{C}_{MR}$ caused by mark-red is:

$$e + n - 1 \leq \mathcal{C}_{MR} \leq 2e$$

- Scan phase initiation requires $n-1$ messages to participating pto's.
- The number of scan requests sent depends on the accuracy with which suspects are identified. In the best case, no requests will be sent but each pto must report termination to the initiator; in the worst case, the number of RPC calls is the same as that for mark-red[8]. Let p be the probability that our suspect identification heuristic is accurate.
- Super group termination requires d_A calls for each group A where $d_A = |\mathit{Dependent^*(A)}|$.

Thus the expected number of RPC calls $\mathcal{C}_{SC}$ caused by the scan phase is:

$$(1-p)e + 2(n-1) + d \leq \mathcal{C}_{SC} \leq 2(1-p)e + (1+p)(n-1) + d$$

- The sweep phase requires $n-1$ messages.

The total number of RPC calls $\mathcal{C}$ required is:

$$(2-p)e + 4(n-1) + d \leq \mathcal{C} \leq 2(2-p)e + (2+p)(n-1) + d$$

The cost of our algorithm is determined by the parameters n, e, d and p. p depends on our choice of suspect; n, e and d are partly determined by the topology of the subgraph and the dynamics of distributed collections but can also be controlled by policy decisions on the extent of mark-red's coverage of the graph. Because little is known of the demographics of distributed objects,

[8] The intermediate case occurs when a subset of the red sub-graph is found to be live.

flexibility is a key goal of our collector. Our collector can be seen as a framework within which policy decisions can be implemented. Policy guides the choice of suspects, the choice of processes forming each group and the merger of groups.

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, the process may not have contributed to a distributed collection for a long time, or the process may simply choose to start a new distributed collection whenever it discovers a suspect object. A very simple heuristic would be to use the local collector alone to identify those surrogates only reachable from the object table but the better the heuristic the greater the chance p that our algorithm traces only garbage subgraphs thereby decreasing the number of times a partial trace is run, limiting the mark-red trace to garbage and reducing the number of scan phase messages to the best case, and decreasing the chance of wasted and repeated work.

A more sophisticated heuristic is to estimate an object's minimum distance from a root, measured by inter-process references — the *distance heuristic* [26]. The distance heuristic requires each in-object to periodically propagate an estimate of its distance from a root to its children, who use this estimate to adjust their own distance estimate. The insight is that the estimate for objects in a garbage cycle will increase without bound; once a threshold value is reached for an object's distance, we have some confidence, but no guarantee, that the object is garbage. A drawback of the distance heuristic is that several objects in a garbage cycle may attain the threshold together, leading to multiple collections in the same cycle (increasing both d and the number of pto's in processes where the collections meet). By only propagating distances over a certain threshold with mark-red requests we can reduce the risk of multiple distributed collections in the same garbage cycle and therefore reduce the overheads of our algorithm. 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.

p and n can be controlled by bounding the amount of work done by mark-red. Recall that this phase needs only make a conservative estimate of the transitive referential closure of suspect objects — it need not visit the whole closure. This policy decision can be taken statically by prior negotiation or dynamically by mark-red. It may be determined by the collector itself or by the user program, globally or on a per-process or even per-object basis. Heuristics based on geography, process identity, distance from the suspect originating the collection, minimum distance from any object known to be live, or time constraints may be used to restrict the extent of mark-red or the decision whether to merge with, overlap or retreat from other distributed collections. In the absence of knowledge of the problem being computed, it is unclear what action should be taken when two groups meet. A merger may not always be desirable. Instead it may be preferable to run multiple overlapping groups. For example the best compromise may be to combine simultaneously occasional long-running but complete collections over very large groups with more frequent faster completing collections over small groups. Our algorithm offers the implementer the choice between completeness and promptness at the level of groups, processes and individual objects. Groups can decide whether or not to merge, processes can decide whether to allow groups to merge, to overlap or to retreat from one another, and objects can decide on merger or retreat.

The cost of distributed collection is comparatively robust against the rate of modification of references held in objects. The cost to the local scan step depends, as with any incremental uniprocessor collector, on how it is synchronised with the mutator [20]. Modifications caused by exchange of mutator messages trigger the barrier, causing the collector to do work, only on the first occasion in a collection cycle that a message is sent from a red out-object. Indeed, the frequency of messages sent to an object that would otherwise be considered suspect is a useful measure of whether that object should be considered as a root for a distributed collection.

9 Related Work

Distributed reference counting can be augmented in various ways to collect distributed garbage cycles. Juul and Jul [22], periodically invoke global marking to collect distributed garbage cycles, tracing the whole graph 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.* [25] present a solution also based on earlier work by Jones and Lins using partial tracing with weighted reference counting [21]. 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 [27] 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* estimates the length of the shortest path from any root to each object. 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 [37]. In a recent paper Maheshwari and Liskov use the same distance heuristic to identify suspect objects from which they start a back trace in an attempt to discover a root [28]. They employ similar reference listing and barriers schemes to those presented here. Unlike [15], their algorithm provides an efficient method of calculating back-references and takes account of concurrency.

Lang *et al.* [24] 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. 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 [11]. Consequently it repeatedly scans the heap until it is sure that it has terminated. This is less efficient 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 Fig. 1 and 2 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.

Hudson *et al.* have adapted their Mature Object Space 'train' algorithm for distributed garbage collection [19, 18]. While their new algorithm collects all garbage, including distributed garbage, it requires an *object substitution protocol* to ensure that all old references to an object are updated to refer to the new copy. Detecting that a train has no external references is also more complex in a distributed environment than in a uniprocessor one: they use a similar token ring technique to that we use for detecting super group termination.

10 Conclusions and Future Work

This paper has outlined a solution for collecting distributed garbage cycles, designed for the Network Objects system but applicable to other systems — a complete treatment will be found in [34]. Our algorithm is based on a *reference listing* scheme [7], augmented by *partial tracing* in order to collect distributed garbage cycles [21]. *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. If necessary, groups can cooperate to collect garbage cycles that span them.

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 to complete a local scan.

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*. The use of the acyclic collector and groups also permits *scalability* whereas the ability to merge groups ensures *completeness*.

Our collector provides a *flexible* framework for the implementation of policy decisions directing the collection of garbage. Heuristics may be applied to govern the choice of suspects, the extent of the subgraph to be traced and whether to allow independent collections that meet to merge, overlap or retreat. Our algorithm therefore offers the implementer the choice between completeness and promptness at the level of groups, processes and individual objects.

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.

Early versions of our algorithm have been implemented. 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. Finally, we would like to thank the anonymous referees for their helpful comments on the earlier draft of this paper.

References

[1] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[2] Lex Augusteijn. Garbage collection in a distributed environment. In de Bakker et al. [12], pages 75–93.

[3] Henry Baker, editor. *International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[4] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Lisp Pointers 1, 6 (April–June 1988), pp. 2–12.

[5] Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 September 1992. Springer-Verlag.

[6] David I. Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer-Verlag, June 1987.

[7] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for network objects. Technical Report 116, DEC Systems Research Center, December 1993.

[8] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. Technical Report 115, DEC Systems Research Center, February 1994.

[9] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[10] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research Report PRC–131, DEC Systems Research Center and Olivetti Research Center, 1988.

[11] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[12] Jacobus W. de Bakker, L. Nijman, and Philip C. Treleaven, editors. *PARLE'87 Parallel Architectures and Languages Europe*, volume 258/259 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

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

[14] Paulo Ferreira and Marc Shapiro. Asynchronous distributed garbage collection in the Larchant cached shared store. Available from Marc Shapiro, May 1996.

[15] Matthew Fuchs. Garbage collection on an open network. In Baker [3].

[16] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Conference on Programming Languages Design and Implementation*, volume 24(7) of *ACM SIGPLAN Notices*, pages 313–320, Portland, June 1989.

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

[18] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Garbage collecting the world: One car at a time. In *Conference on Object-Oriented Systems, Languages and Applications — Twelth Annual Conference*, volume 32(10) of *ACM SIGPLAN Notices*, pages 162–175. ACM Press, October 1997.

[19] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Bekkers and Cohen [5].

[20] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[21] Richard E. Jones and Rafael D. Lins. Cyclic weighted reference counting without delay. *Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1993.

[22] Neils-Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Bekkers and Cohen [5].

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

[24] Bernard Lang, Christian Quenniac, and José Piquer. Garbage collecting the world. In *Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 39–50. ACM Press, January 1992.

[25] 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 Baker [3].

[26] Umesh Maheshwari. Fault-tolerant distributed garbage collection in a client-server object-oriented database. In *Conference on Parallel and Distributed Information Systems, Austin*, September 1994.

[27] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *Principles of Distributed Computing*, 1995.

[28] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by back tracing. In *Principles of Distributed Computing*, 1997.

[29] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1991.

[30] David Plainfossé and Marc Shapiro. Experience with fault-tolerant garbage collection in a distributed Lisp system. In Bekkers and Cohen [5].

[31] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In Baker [3].

[32] S. P. Rana. A distributed solution to the distributed termination problem. *Information Processing Letters*, 17:43–46, July 1983.

[33] Helena C. C. D. Rodrigues and Richard E. Jones. A cyclic distributed garbage collector for Network Objects. In *International Workshop on Distributed Algorithms WDAG'96*, Bologna, October 1996.

[34] Helena C.C.D. Rodrigues. *Cyclic Distributed Garbage Collection.* PhD thesis, Computing Laboratory, The University of Kent at Canterbury, 1998. In preparation.

[35] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage collection protocol. In *Symposium on Reliable Distributed Systems*, Pisa, September 1991.

[36] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, November 1992.

[37] N.G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, December 1992.

[38] Gerard Tel and Friedmann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1), January 1993.

[39] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In de Bakker et al. [12], pages 432–443.

[40] Paul Wilson. Distr. gc general discussion for faq. gclist mailing list (gclist@iecc.com), March 1996.

[41] Paul Wilson. Garbage collection and memory hierarchy. In Bekkers and Cohen [5].