Garbage Collecting the World

Bernard Lang*
INRIA–Rocquencourt

Christian Queinnec†
École Polytechnique
& INRIA–Rocquencourt

José Piquer‡
Universidad de Chile

Abstract

Distributed symbolic computations involve the existence of *remote references* allowing an object, local to a processor, to designate another object located on another processor. To reclaim inaccessible objects is the non trivial task of a distributed Garbage Collector (GC). We present in this paper a new distributed GC algorithm which *(i)* is fault-tolerant, *(ii)* is largely independent of how a processor garbage collects its own data space, *(iii)* does not need centralized control nor global stop-the-world synchronization, *(iv)* allows for multiple concurrent active GCs, *(v)* does not require to migrate objects from processor to processor and *(vi)* eventually reclaims all inaccessible objects including distributed cycles.

These results are mainly obtained through the concept of a *group* of processors (or processes). Processors of a same group cooperate together to a GC inside this group; this GC is conservative with respect to the outside of the group. A processor contributes to the global GC of all groups to which it belongs. Garbage collection on small groups reclaims quickly locally distributed garbage clusters, while garbage collection on large groups ultimately reclaims widely distributed garbage clusters, albeit more slowly. Groups can be reorganized dynamically, in particular to tolerate failures of some member processors. These properties make the algorithm usable on very large and evolving networks of processors. Other than distributed symbolic computations, possible applications include for example distributed file or database systems.

*INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France — `lang@margaux.inria.fr`.
This work has been partially supported by the Eureka Software Factory project.
†LIX (URA 1439), École Polytechnique, 91128 Palaiseau Cedex, France — `queinnec@polytechnique.fr`.
This work has been partially funded by Greco de Programmation.
‡Departamento de la Computación, Universidad de Chile, Casilla 2777, Santiago, Chile — `jpiquer@dcc.dcc.uchile.cl`.

To appear in: Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92), Albuquerque, New Mexico, January 19-22, 1992.

1 Introduction

Computations performed by collections of processors are more and more common today. Shared memory systems only allow for a limited number of processors. Some problems instead are clearly parallel and would benefit from greater and greater numbers of cooperating processors. This paper presents a new distributed Garbage Collection (GC) algorithm well suited for very large nets of possibly heterogeneous processors, even for a world-wide net. Our proposal can be roughly sketched as follows.

Any processor manages its own data space with a local GC. Remotely referenced objects have a reference counter so that a large part of these objects can be easily deallocated when becoming inaccessible. Processors are organized into groups. The processors of a group cooperate to partial GCs global to the group: the aim of a group is therefore to discover and reclaim all unreachable distributed cycles of objects in the group by means of a concurrent mark-and-sweep collector. Multiple overlapping group GCs can be simultaneously active. When a processor or a communication link fails to cooperate, the groups within which it lies are reorganized and continue their work. Eventually all distributed clusters of unreachable objects will belong to a group that reclaims these clusters when it finishes its associated GC.

Our algorithm has some other interesting properties:

- it supports failure of processors but is still able to reclaim unused cells when located on working processors only. It also supports the addition of new processors, or changes in the network topology.
- it does not need a centralized control: for instance, a net may split into two disconnected subnets each of which will continue to scavenge its own space. Multiple group GCs can be simultaneously active, each of which is focusing on a particular subnet where garbage must be reclaimed.
- it can use any kind of tracing GC (mark-and-sweep, copy, etc.) for local collections, provided this local GC transmits the marks used by the group GC from remote entry references to remote

exit references. However no special bits for the group GC are required during the local collections.

- the detection and/or reclamation of unused cells does not require the migration of objects from processor to processor: the algorithm respect the locality of objects as decided by the mutator.

Our algorithm is robust and we think that it can be used not only in the run-time library of distributed symbolic computation languages, but also by distributed file systems or distributed database systems to reclaim unused files or objects. It is particularly attractive for these systems since it makes weak assumptions on their local GCs.

We first establish the context and our terminology in section 2, and then describes in section 3 the working of algorithm for a single group of network nodes. The problem of node failure and of possible recovery strategies from the point of view of both the GC algorithm and the application program are then discussed in section 4. The next section discuss ways of performing cheaply several group collection at the same time. Some comparison with related work and a conclusion end the paper.

2 Terminology

We consider a collection of nodes organized into a *network* and communicating by exchange of messages. We call *node* a processor or a process on a processor able to manage its own memory space. Nodes may contain processes called *mutators* performing independent computations and allocating chunks of memory called *cells*, either for their own need or to serve other nodes. Cells may contain references to cells in the same or other nodes. Each node also contains *roots* which are references to cells it considers useful. In particular, all cell references known by a mutator (e.g. in registers or in an execution stack) are roots. Cells referenced by a root directly or indirectly through other cells are said to be *reachable* or *live*. Other cells are said to be *unreachable* or *dead*, and they constitute the *garbage* memory to be reclaimed by *garbage collection (GC)*. A reference to a cell in the same node (as that where the reference is found) is said to be *local*. A reference to a cell on another node is said to be *remote*. Garbage collection within a node on the basis of local roots and local references is called a *local GC*.

A *remote reference* to a cell γ is represented by a reference to an *exit item* on the same node, which references an *entry item* on another node, which itself references locally the cell γ (see figure 1). To fetch the value of a remote reference, three indirections and some communication time are required; it is of course hoped that the number of remote references is far outnumbered by the number of local references. Exit (resp. entry) items are immutable with respect to the cells they refer to, and thus they can be safely shared: each node has only one exit item for all remote references to a given cell, and only one entry item for each remotely referenced local cell.

A *group GC* is a non-local GC, i.e. a GC which involves more than one node. A group GC operates on a *group* of nodes, and it reclaims any cell of the group that it can prove *inaccessible from any root of any node*. Note that entry or exit items are not part of the set of roots. The creation of a remote reference involves the synchronized creation of the associated entry and exit items.

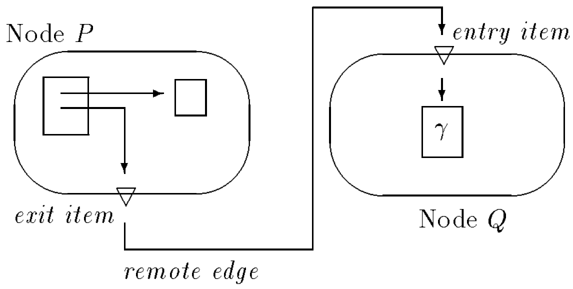

Figure 1: Remote reference

Entry items have a *reference counter* that is equal to the number of exit items referencing them (up to messages in transit). When an exit item is reclaimed, a decrement message is sent to the counter of the entry item it was referencing. If this decrement message brings its counter down to zero, the entry item is reclaimed too. This is the only available mechanism to reclaim entry items; it is safe since non cooperative nodes (or nodes that are down) do not send decrement messages and thus the cells they refer to cannot be reclaimed at all (without an external intervention). Reference counters along with increment or decrement messages do pose some problems in a distributed environment where no global time order exists. Nevertheless weighted reference count [Bev87], generational reference count [Gol89] or indirect reference count [Piq91] can, for instance, be safely used to maintain these counters. The problem with reference counters is that they cannot reclaim dead cycles of cells spanning several nodes.

On each node, a local *collector* reclaims unreachable cells while the computation proper is done by a local *mutator*. The mutator and the collector can interleave their work with a granularity ranging from the usual stop-and-collect mode to the concurrent mode described in [DLM+78]. We restrict local GCs to belong to the "tracing" family, following the terminology of [LD87].

Several nodes can form a *group* in order to perform a GC global to this group. Such a *group GC* will reclaim unreachable cells and, in particular, the unreachable cycles that span nodes within the group. A group GC is *partial* unless the group contains all possible nodes. Groups can *overlap* or form *hierarchies* and therefore can contain smaller groups. Hierarchies are useful when gathering nodes that exhibit some sort of locality. This locality may be for instance topological or geographic: a group can be defined to contain the different processes of a processor, the processors of a local area network, the networks of a country etc. The locality may also be a logical one, for example based on the ratio of mutual remote references.

Though groups can be dynamically created to perform group GCs, we expect that preferred hierarchies of groups can be foreseen, for example based on physical neighbourhood (as above) to minimize communication problems.

Groups are intended to gather nodes or groups that will cooperate without failure during a group GC. We assume a fail-stop mode where a failing node (or link) ceases to communicate and does not fool other nodes in a byzantine way. If a node fails or, more generally if a node does not want to cooperate, then the groups to which it belongs can exclude it so that the work can be pursued on smaller groups without losing what is already done. Any cycle passing through a non-cooperative node cannot be reclaimed at all, but dead cycles spanning only cooperative nodes in a same group will still be reclaimed by the group GC performed on this group. Some groups must be large enough so that long cycles can be recovered, but the larger they are the longer the group GC takes, and the higher the probability that some node in the cycles fails before the end of the group GC. However, if groups can be defined to exactly cover the cycles that are expected to be reclaimable, a global GC can thus be achieved on all nodes through independent partial group GCs.

3 The basic algorithm

We consider a network of nodes containing cells, some of which are linked by remote references. We first describe how a group may be created, and how a global GC is performed on the group. In section 5.2 we shall discuss in more detail the simultaneous execution of this algorithm on several groups or on all of them.

The steps listed here constitute the single group algorithm. They are further developed in the remainder of this section.

1. **group negotiation:** A node wanting to participate to a group GC sets up a group within which it will be performed.
2. **initial marking:** All the entry items of nodes within the group are marked with respect to the group. The marks on entry items depend on whether they are referenced from inside or from outside the group.
3. **local propagation:** Local GCs propagate the marks of the entry items towards the exit items.
4. **global propagation:** The group GC propagates the marks of the exit items towards the entry items they reference, when within the group.
5. **stabilization:** The preceding two steps are repeated until marks of entry or exit items of the group no longer evolve.
6. **dead cycles removal:** The group GC breaks the unreachable cycles in the group.
7. **group disbanding:** The work is now finished and the group may be disbanded.

Recall that, as announced above, several group GC may actually take place simultaneously, for groups of varying sizes, possibly overlapping.

The single group algorithm can be viewed as the cooperation of a variety of traditional GCs at various levels: local GCs can be any kind of tracing GCs, entry or exit items are managed by means of reference counters, and any dead cycle through these items is eventually broken by a global concurrent mark-and-sweep GC on a group encompassing the cycle.

3.1 Group negotiation

When a node decides to participate to a new group GC, it inspects the other nodes to determine in cooperation with them what group can be set up for that purpose. There are multiple reasons to be involved in a group GC: the node can be idle, or its entry items have not been accessed for a long time, or it is not currently involved in any group GC of some given size range, etc. The node is free to choose the kind of group it wants. It can either choose to form a small group with very close nodes or to undertake a major group GC with all nodes it is aware of. Let us suppose for now that the newly created group is composed of nodes only (we will explain later how to manage groups containing groups). The group is a set of nodes willing to cooperate together until the end of the associated group GC. Once the group is created, there is no such thing as a leader of the group but all nodes of the group are aware of their co-members.

The technique actually used for group formation is not essential to our algorithm. One simple way to implement group negotiation is to predefine a hierarchy of groups, as shown in figure 2, based for instance on neighbourhood criteria. Small dead clusters of cells

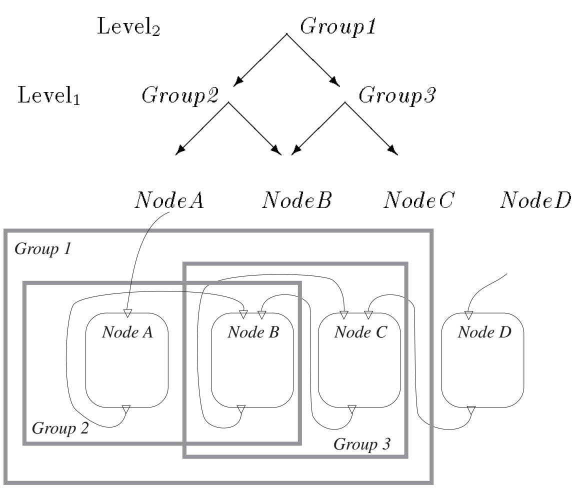

Figure 2: Group hierarchy

are quickly reclaimed by GCs on small groups, while larger ones are ultimately reclaimed by the slower GCs on large groups. If groups are predefined then group negotiation is an instance of a distributed consensus algorithm [LS76, Gra78].

In order to distinguish messages relevant to a specific group, or a specific group GC cycle, a *unique identifier* is associated to each group GC cycle and is made known to the nodes in the group. This can be done concurrently with the group creation and the initialization of marks for this group GC (cf. next section). This unique identifier is necessary to keep track of dynamic reconfiguration of groups during a cycle (for example because of a node failure), or simply for the proper management of messages. Its role is actually dependent on the variant of our algorithm one chooses to implement, and on the need to handle some fault tolerance problems (e.g. late or disordered messages).

3.2 Initial marking

Within each group, entry and exit items have a mark. A key point of our algorithm is that marks are local to groups i.e. are only meaningful with respect to a particular group GC. Different groups may give different marks to a same (entry or exit) item. With respect to a group, an entry item may be marked *soft* or *hard*, and an exit item may be marked *none*, *soft* or *hard*. These marks are strictly exclusive and can only be increased from *none* to *hard* during the local GC for exit items, and from *soft* to *hard* during the group GC for entry items.

Initially, a *hard* mark for an entry item means that it is "needed outside the group", while a *soft* mark means that the entry item is only referenced from inside the group (if at all). Later, a *hard* mark may also mean "accessible from a root of a node in the group". The unused cycles that will be reclaimed in the end only include *soft* marked items (i.e. items that are unreachable both from outside the group and from the roots in the group).

The initial marks of the entry items of a group can be determined locally to this group by means of the reference counters: this technique was inspired by Christopher [Chr84]. Christopher's algorithm allows, given a group G and after a computation local to G, to know precisely which (and how many times) entry items are referenced from outside G. The initial marking protocol can thus be done in four successive steps:

1. on every node of the group and for every entry item, a copy of the reference counter is created for this group.
2. on every node of the group and for every exit item, if it references an entry item belonging to a node in the same group, a decrement message is sent that will decrement the copy of the reference counter of that entry item.
3. when all decrement messages generated during the previous step are received then on every node of the group, any entry item that has a strictly positive copy of the reference counter is marked *hard*. Otherwise the copy of the reference counter is zero, the entry item is therefore only referenced from inside the group and it is accordingly marked *soft*.
4. the space for the copies of the reference counters is reclaimed (though we shall see that it may be useful to keep these copies).

In practice, many groups may coexist and a node may belong to several groups. Marks and copies of reference counters must be separate for each group. An entry or exit item data structure can contain these additional fields indexed by the group identifier.

A classical termination detection algorithm has to detect that *(i)* no decrement message for this initialization is still in transit, and *(ii)* all nodes in the group have sent their decrement messages. Group negotiation and initial marking can be partly combined. A request to a node for cooperation in a group GC may be accompanied by a bunch of decrement messages.

Once marks have been initialized, every node starts propagating them locally and globally as described by the next two sections.

3.3 Local propagation

A group GC needs cooperation from local GCs. Local GCs are not time-constrained. A group GC simply waits for the local garbage collector to start a new local cycle, and thereby contribute to the group GC. Each local GC cycle is followed by extra steps that propagate its results for the benefit of group GCs, as explained in section 3.4. A local GC cycle may be initiated by local computational need, or may be urged from an external source such as another node noticing it has not contributed for a long time to some group GC. Local GCs are not required to be marking or copying GCs, they are only required to propagate the marks from the entry items to the exit items they locally reference, directly or indirectly. After a local GC on a node, any exit item locally accessible from an entry item (of the same node) is marked at least as hard as this entry item was at the beginning of the local GC.

A node is said to be *stable* w.r.t. a group GC when the propagation of its entry marks to the exit items by a local GC would not change the previously found marks on its exit items, and thus would not contribute any new data to the group GC (cf. section 3.5).

One simple way to achieve the propagation of entry marks is to have a *two-phase marking* for the local GC (see figure 3).

- Initially, all marks on exit items are reset to *none*. If the exit marks resulting from a previous local GC are to be kept (cf. section 3.4), a copy is made in an appropriate location.
- then a first tracing is performed from the *hard* entry items and the internal roots. Any exit item reached by this tracing is marked *hard*.
- finally, a second tracing is performed starting from the *soft* entry items. This second tracing completes the first, i.e., for example in the case of a marking algorithm, the marks of the first tracing are not removed. Any exit item reached by this second tracing is marked *soft*, if it is not yet marked *hard*.

After such a local GC cycle, the following is known:

- Tracing from *soft* entry items is conservative since it is not yet known whether the entry items are useful or not w.r.t. the whole network. Cells that have not been visited at all are not reachable either from outside the node (trough entry items) or from the local roots. They can thus be reclaimed safely (this is implicit in case of a copying algorithm).
- The same is true of exit items that are still marked *none*. They can be safely reclaimed, though the entries they reference must be informed (see below).
- Exit items marked *hard* are reachable either from a *hard* marked entry or from a local root. Hence they are either reachable from a root within the group or referenced from outside the group (up to floating garbage, i.e. the reference may have disappeared since the beginning of the group GC cycle). The same is of course true of the entry items referenced by these exit items.
- Exit items marked *soft* carry no new marking/usefulness information w.r.t. the group GC, since the entries they reference are at least already marked *soft*.

Only a single bit mark is needed for marking the cells within the node, to indicate whether they are (conservatively) live according to the local GC. The distinction between *soft* and *hard* is done by separating the two phases, so that it can be propagated from entry items to exit items, where the two kind of marks are physically distinct.

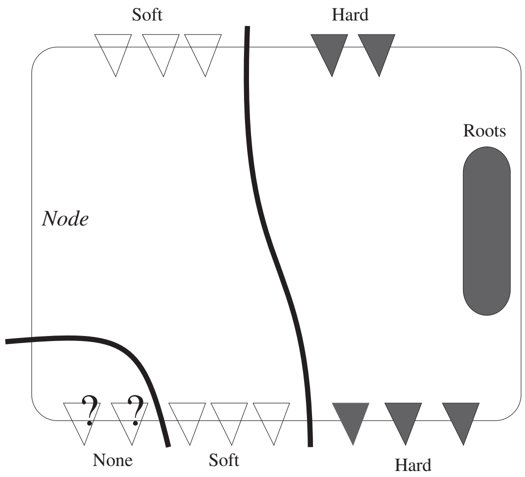

Figure 3: Two-phase marking in a local GC

When an exit item is reclaimed, a decrement message is sent to its associated entry item, which is then reclaimed in turn if its reference count reaches zero, even if it is marked *hard*. In summary, note that *(i)* the two-phase marking may be performed by any kind of tracing GC[1], *(ii)* the work required by the two-phase marking is equivalent to that of traditional tracing GC except for the order of exploration of cells,

[1] The tracing GC must actually trace all the local cells in order to properly propagate marks. This is not done by generational GCs [LH83]. In their case, a simple approach to understanding the problem is to consider generations as distinct nodes, and then to adapt the techniques presented here. In the case of a strictly hierarchical organization of groups (see section 5.2), it may be convenient to privilege generational proximity over geographical proximity for organizing the groups (e.g. by grouping together young generations of different nodes). This has not been pursued in depth by the authors.

(iii) no extra bits are required on local cells to record the global marks, only entry or exit items differentiate between them. Indeed, the two-phase marking may be performed by a copying collector that does not use marks at all.

It is possible to use a concurrent local GC [Wad76, KS77, DLM+78, Bak78, Yua90], but not a page-oriented one as in [BDS91]. The only difference is that the mutator has to cooperate with the collector by marking some of the cells it touches. The trick is to ask the mutator to always do *hard* marking. During the *hard* phase, this works like an incremental collector. At the end of the *hard* marking phase, all cells that the mutator can access in the future (except for cells it will create and mark *hard*) are already marked *hard*. Hence, during the *soft* phase, continued cooperation from the mutator (still a *hard* marker) will only mark already *hard* marked cells, and this will be totally innocuous though with some time overhead due to this useless marking. Extra floating garbage [Wad76] may also be produced because the exploration order of the two-phase marking may not be optimal. Note however that the working of the algorithm with concurrent collectors is actually very complex to analyze because of incoming messages and interactions with mutators on other nodes.

3.4 Global propagation

When an exit item is known to be *hard* i.e. accessible from a local root or referenced by at least one exit item not belonging to the group, its mark has to be propagated to the entry item it references whenever it belongs to the group under consideration. It is not mandatory to send this information immediately, it may be batched with other messages to lessen transmission cost. Conversely, it is not necessary to wait till the end of the local collection before propagating a *hard* mark, and early propagation may speed up the termination of the group GC. The propagation of a *hard* mark from an exit item can only *harden* the mark of the associated entry item. Note that once an exit item has been marked *hard* with respect to a group, remembering this mark will avoid sending further hardening message from that exit item and with respect to that group. Hence it may be useful to preserve the marks previously found for exit items w.r.t. each group GC.

A local GC may work for one or several group GC at a time. When it finishes a local cycle, it is free to start a new one, provided the *hard* marks on exit items have been propagated or preserved for later propagation. This new cycle may be for purely local use, or also for the same group(s) and/or other groups.

When a new remote reference is created, the associated entry item is marked *hard* since it is (will be) necessarily accessible from a root of the node for which it is created, and to which this remote reference will be sent[2].

3.5 Stabilization

Global propagation of marks is finished when all entry items of the nodes participating in the group GC are marked *hard* whenever they are reachable from a cell outside the group or from a root belonging to the group. Note that is a conservative requirement: some *hard* marks may be due to past reachability that has disappeared since the beginning of the group GC cycle (floating garbage).

Such a situation, called *group stability* is reached when

- All nodes are stable, i.e. they have no new data that could justify *hard*ening more entry items locally or elsewhere *within the group*.
- There are no messages in transit that request the *hard*ening of some entry item.

A node is stable if it has propagated all the (relevant) *hard* marking data it is aware of to the other nodes in the group. This is normally done after a local GC. New *hard* marking data comes mainly as requests to mark *hard* an entry item previously marked *soft* w.r.t. to the current group GC. When this occurs, the node reverts to (or remains in) a non stable state until the marking has been propagated to the exit items by a local GC and (new) *hard* marks on exit items have been propagated to the other nodes of the group.

The stability of a node may also be lost when a new entry item is created on this node. Though the cell referenced by the new entry item must have been live, this fact may not yet have been discovered by the group GC, and there is the possibility that the path that formerly kept it alive will be severed before *hard* marking may be propagated to it.

Finally, stability may be lost when a cell immigrates from another node. Though this cell is obviously live, the knowledge of this liveness may not have been yet propagated to the local cells or the local exit items it references.

Assuming that all messages arrive[3], group stability must be reached since marks can only increase from *none* to *hard* and the total number of entry or

[2] The creation of a remote reference requires some care to keep a consistent reference count on the new entry item, while its intended recipient has not yet received (and acknowledged) a reference to it. This problem, as well as other problems related to protocols to preserve the consistency of distributed reference counts have been discussed in other proposals. They are not considered further in this paper.

[3] Fault-tolerance w.r.t. message behaviour is not discussed in this paper.

exit items is bounded by the total number of possible cells that can be allocated in the nodes of the group. Moreover any entry item created during the group GC is marked *hard*. Group stability can be detected by any distributed termination detection algorithm when no node failure occurs. Examples of such algorithms in the area of garbage collection may be found in [HK82, Aug87, Der90]. The handling of node failure is discussed in section 4.

3.6 Dead cycles removal

After stabilization, all entry items that are directly or indirectly accessible from a root or from a node outside the group are marked *hard*. Entry items marked *soft* can only be part of inaccessible cycles local to the group and can thus be safely reclaimed. *Soft* entry items can be independently reclaimed by each node of the group without group synchronization. This is gracefully achieved by relying on the reference counting mechanisms.

When group stability is detected, each node in the group modifies its soft entry items to now reference `nil` rather than a local cell. This mutation is safe since these entry items are dead. The former offsprings of these entry items, not otherwise accessible, will be reclaimed by the next local GC. Similarly the exit items that were kept alive exclusively by these entries will be reclaimed by the next local GC. The reclamation of such an exit item causes the sending of a decrement message to the entry item it references. In the case of dead loops, (dead) entry items on the loop eventually receive decrement messages from all the (dead) exit items that reference them. Hence their reference counters decrease to 0 and they are eventually reclaimed by the normal reference counting mechanism. This protocol achieves a delayed reclamation instead of a synchronized deletion of useless cells. The latter is difficult, since we cannot brutally reclaim the useless exit items which are still referenced from local cells, and since we cannot either reclaim the useless entry items because they are still referenced by exit items.

Observe that our algorithm is independent of the nature of inaccessible cell clusters: they can be simple cycles or more complex cycles entangled with subcycles, etc. Occasionally they may even be non cyclic structures, though these are often reclaimed earlier through the reclamation of exit items marked *none* after a local GC. All inaccessible entry items are reclaimed and no heuristics is needed to identify potential cycles. Also note that the deallocation mechanism of entry items is unique and only based on reference counters. Similarly the deallocation of exit items is only a consequence of local GCs. The removal of dead cycles is effectively obtained by cutting all the remote edges of these cycles.

3.7 Group disbanding

When a group GC is finished, its associated group may be disbanded, though it is often convenient to keep the same groups for further group GC. Marks (and possibly other data structures) relative to this group can then be reclaimed. If the delayed protocol of the previous step is used, this reclamation can take place as soon as it finishes mutating to `nil` its *soft* entry items.

4 Failure

If a node fails, i.e. ceases to cooperate, then we assume that this will be detected by some of the other nodes of the group, for example those nodes which precisely require cooperation. For that detection, messages with acknowledgements and time-out can be used. A node which detects such a failure may choose between several non-exclusive options:

- It can decide that this failure is probably only temporary and waits for the failed node to wake up. This only slows down the work of some groups to which the silent node belongs.
- It can reorganize the group i.e. create a new group excluding the failing node (and all other nodes with which communication is now impossible if, for example, relayed by the failing node).

The simplest method to reorganize the group is to build a new group, subgroup of the failed group, and to restart from scratch a new group GC on that subgroup. However the existing information already gathered by the failed group can be used to initialize the new group. In this case the *hard*-marks are kept and transferred as *hard*-marks for the new group. The marking initialization procedure of section 3.2 is used only to promote from *soft* to *hard* all entry items accessible from nodes that have been excluded from the new group. The group GC on the new subgroup is then resumed and since it starts from more advanced marking information, it can stabilize more quickly. However to start from marks inherited from the failed group may increase the ratio of *hard*-marked floating garbage.

The failed group may be aborted immediately. It may also be preserved for some time in the hope that the failure is temporary and the group GC can be resumed later, or until the failure is definitely known as non-recoverable.

A transmission link may fail and divide a group into disconnected subgroups. These subgroups will then reorganize themselves independently, and they will resume independent parts of the group GC that will reclaim all cycles that are local to each of the reorganized subgroup. Of course, cycles spanning through missing links will not be reclaimed by these partial group GCs.

Note that, in all cases, the cells that were reachable from the failed node will never be reclaimed, since the reference counts of the entry items referenced from the failed node can no longer decrease to 0. Hence, even if the group GC is aborted, no remote data used by the failing node will have disappeared when it resumes operations.

When a node has a non-recoverable failure, three types of problems may have to be considered:

1. what happens to cells referenced by the failed node?
2. what is to be done of remote references to cells in the failed node?
3. what is to be done of cells in the failed node that can be recovered by external means (e.g. local operator intervention, or known replication on different nodes)?

Answers to the second question are essentially the responsibility of the application. However, if some cells can be recovered in the failed node together with their network identity (the corresponding entry item), it may be useful to have a mean to migrate these cells to another node which is made known to the network so that corresponding exit items may be updated. A similar action may be first taken when the death of the node has to be decided by an external agent[4].

The first question is the more interesting one for our algorithm. References from the failed node may be determined by Christopher's technique, provided a reference count of references internal to the group has been kept up-to-date on all entry items of the group[5]. Let G be the original group, N the failing node, and G' the group G less the node N. We can run Christopher's algorithm on G' to obtain for the entries in G' the count of references from within G'. By taking the difference with the reference counts w.r.t. group G, we can determine which entries in G were being referenced by the failing node N (or any number of failing nodes taken together). It is then possible for a network or application administrator (either a program or a human being) to take appropriate action with respect to these entry items and the cells they access. Decrement messages may be sent to the concerned entries to consummate the death of the references from the failing node[6]. Alternatively the administrator may decide that the cells that were accessed by the failing node contain important data that should not be allowed to die. In that case he or it may decide to create new live references to these cells.

Note that the recovery facility described above requires keeping up-to-date the internal reference counts for each entry w.r.t. each group it belongs to. This entails both a space overhead on each entry, and a time overhead since several reference counts have to be updated whenever a remote reference is lost or created. The hierarchical scheme proposed in the next section will answer both these concerns.

5 Simultaneous group collections

5.1 Contention between group GCs

Our algorithm makes use of local GCs to perform parts of a group GC. In other words, a group GC delegates to a local GC the propagation of the marks belonging to the group GC. This situation could be replicated at other levels, and in particular a subgroup G' of a group G could be considered as a single node from the point of view of the collection in group G. While this could possibly be useful in some cases (e.g. large variations in network connectivity and/or communication speed), it seems not to be generally advisable. A first reason is that entry items w.r.t. a subgroup G' are still to be located on nodes. Though they can be the same objects as node entry items, not all node entry items are subgroup entry items since some remote references may be fully local to the subgroup. This entails additional management overhead for entries. Similarly, exit items have to be created for the subgroup G' and kept till the end of the group GC for G', i.e. much longer than the duration of a local GC on a single node.

It is instead much simpler and more efficient to consider that several group GCs take place simultaneously, and that a local GC on a node can contribute to several of them. Entry and exit items can then be common to all group GCs, though they must keep separate marks (and possibly separate group reference counts — cf. section 4) for each group.

If we consider as above a group G and a subgroup G' of G, a *hard* mark w.r.t. G on an entry item means that it is accessible from some root in G or from cell outside G. Hence this entry item must also be marked *hard* w.r.t. G' since it must be also accessible either from a root in G' or from outside G' (because G' is included in G). Hence, any local collection that con-

[4] Migration with reference counter is taken into account by [Piq91].

[5] Maintaining group based reference counts is also useful to initialize the marking of entry items when a new GC cycle is decided for the same group. Then Christopher's algorithm need be used only when creating a new group.

[6] This must be remembered to inform the failing node, when there is a chance of later partial or total recovery, so as to avoid dangling references. Such a mechanism does not properly belong to the garbage collection algorithm, but it must be available to allow the application programs to handle this situation.

tributes to the group GC of G can also contribute to the group GC of G' by propagating a *hard* mark w.r.t. G' to every entry item for which it does it w.r.t. G.

However the *hard* marking phase of a local GC w.r.t. G is not a complete one for G' since some entry items may be marked *hard* w.r.t. G' and *soft* w.r.t. G. Thus if a local GC works for for the group GC of G, this may slow down the progression of the group GC for G'. Conversely, if a local GC works for for the group GC of G', its *hard* marking cannot be used at all for the group GC of G, which is even more slowed down.

Contention between two embedded groups is far from a clear-cut issue. Each slows the other down, but not to the same extent. Smaller groups can terminate faster, with less floating garbage, but they recover less storage and never the large dead cycles, and they do not contribute to the group GCs of larger embedding groups. Conversely, larger groups may take much longer to terminate, thus leave more floating garbage, but they usually recover more storage and only they can recover larger dead cycles of cells. Additionally a large group GC contributes to some extent to the GCs of smaller embedded groups, and when a group GC on a larger group finishes its work, it reclaims all its dead internal storage, thereby also completing the work of all its subgroups (up to floating garbage).

The situation is simply worse when a node belongs to two overlapping groups (i.e. groups with a non trivial common intersection). Then the speed of the two group GCs is halved on the average since the local GC can work only for either group, as there is no simple relation between the marks for the two group GCs.

This discussion hints first at the necessity of having a strictly hierarchical embedding of groups (no partial overlap) to avoid group contention over the services of the local GCs. We show in section 5.2 how to deal with the contention problem in the case of embedded groups, and thus have each local GC of a node contribute to the group GCs of all groups to which the node belongs. Hence all GCs can terminate faster, dead cells are recovered earlier, and less floating garbage is produced.

5.2 Hierarchical cooperation of group GCs

We now consider that all groups are organized in a strictly hierarchical order by inclusion. If two group overlap, then one contains the other. We further assume that there is one group containing all the node in the network, which we call the *universal group*.

Each group can then be assigned a *level index* which is the number of groups it is strictly embedded in. Hence the universal group has level 0. Note that the level of a group may change as groups embedding it may be destroyed or newly created. This problem must be taken in consideration when implementing the algorithm below since it uses level related data. At any given time, on a given node, group levels uniquely identify the groups that contain the node. Each node will keep a table relating the unique identifier of each containing group to its group level.

Our objective is to have each local GC contribute precisely to the marking of entries w.r.t. to the group GCs of all groups containing the node. We have seen in the previous section that when an entry is marked *hard* w.r.t. some group G, it can also be marked *hard* w.r.t. any subgroup G' of G. Hence, for an entry x in a node N, we can define a marking level $Mark_x(N)$ as the least level (i.e. the level of the largest group) such that the entry x is marked *hard*. Similar marking levels can be used on exit items. *Remember that a lower marking level actually means more* hard *marks since level indexes are in reverse inclusion order.*

Now, instead of propagating an ordered binary marking (*hard* or *soft*) from entry items to exit items with a two phase tracing algorithm, the local GC must propagate an integer marking by means of a multiphase tracing algorithm (or any equivalent algorithm). Multi-phase marking is performed by propagating the $Mark()$ of entries in increasing order, so that each exit item get the lowest marking level of all entries that can reach it. The tracing from the roots is done with the 0-level trace and thus marks exit items *hard* w.r.t. all groups. This multiphase tracing is very similar to the time-stamp propagation used in Hughes' algorithm [Hug85]. Some care has to be exercised with the multiphase tracing (and with the 2-phase tracing as well) because the marking level of some entries could decrease while it is taking place, and these entries should not be skipped.

Since levels are meaningful only locally to a node, the marking levels on exit items are changed to the unique identifier of the corresponding group before they are sent to the node they reference. This unique identifier can be turned back into a local marking level upon arrival.

Stability is also detected by means of group levels. A node is stable w.r.t. level ℓ when it has not received any data that could justify decreasing the former marking level of some exit item down to or below ℓ (cf. section 3.5). When global group stability is known at level ℓ on a node, all entry items with a level marking higher than ℓ may be reset to `nil` (cf. section 3.6). Recall that this may not be true at level ℓ for all nodes, since their level ℓ group may be different. level).

For the next GC cycle in this group, marking levels of entry items must be reinitialized. Whenever, for a given entry, the count of references external to the group is non-zero, the marking level of that entry item is set to the group level, unless it is already smaller.

Multiphase tracing can be extended to concurrent local collectors (as it can be in the case of Hughes' collector). As explained in section 3.3, the mutator can just act as a 0-level marker. Actually it just contributes to the tracing in the usual way, of parallel collectors. This contribution is naturally taken as 0-level one during the 0-level phase (the first phase), and is useless but innocuous during the later phases.

However, there is one drawback to hierarchical cooperation of GCs as described above. The assertion that " an entry item that is marked *hard* w.r.t a group G has to be *hard* w.r.t. any subgroup G' of G " is not quite correct because of the existence of floating garbage. The entry items on a dead cycle may have a low marking level because they were formerly reachable in some large group G. This reachability may have disappeared, but the marking remains low until the end of the GC cycle for group G, which make take a long time since the group G is large. During all that time, the GC of any subgroup G' containing the cycle will have to assume that this dead cycle is live, because the marking level mechanism will keep the low marking level of G on the entry items of the cycle.

Hence, if completing GC cycles on large groups takes much longer than on small ones, it may be advisable to occasionally suspend the work on the larger groups, and perform hierarchical GC only for the smaller groups to avoid the above effect[7]. Afterwards, hierarchical GC on all groups can be resumed from its saved state. Remember though that this problem concerns only dead cycles since other garbage is always reclaimed by the reference count mechanism.

5.3 Keeping group reference counts

Keeping reference counts up-to-date w.r.t. each group can be useful for two reason:

- When groups do not change, it allows a faster initialization of the marks on entry items (thus avoiding the cost of Christopher's algorithm).
- It is necessary to enable the recovery procedures described in section 4

One obvious solution is just to keep them all in the entry items, and update them when the number of references change. In the case of a strictly hierarchical organization of groups, one can use an array $Count_N[i,x]$ giving the reference count of entry x of node N for the level I group (i.e. the number of references from inside that group). However this entails a significant overhead, both in space for keeping the counts, and in time for updating all of them.

Instead we propose to keep the usual global reference count for the whole network, and for each of the other groups a *difference count* which is defined as follows:
$\forall i \geq 1,\ Diff_N[i,x] = Count_N[i-1,x] - Count_N[i,x]$

This has several advantages:

- Difference counts are very convenient for initializing marking levels. The initial marking level is the smallest i such that $Diff_N[i,x] \neq 0$, or 0 if they are all zero. However, during the reinitialization of marking levels for the next GC cycle of a group, it may be that the marking level of entry item x is already smaller, in which case it is not changed (cf. section 5.2).
- It requires less updates: when counts have to be updated (incremented or decremented) only the global reference count and one difference count have to be modified.
- Difference counts are usually much smaller, often equal to 0. Hence it is easier to use storage-saving techniques such as 1 or 2 bits reference counts [DB76, WF77].

6 Related works

Our algorithm, like many others, is based on the concept of multi-area collection which was pioneered by Bishop [Bis77]. Distributed [Hug85, LL86, Rud86, Bev87, Gol89, Der90, Piq91] or fault-tolerant [Ves87, Sch89, SGP90] or real-time [Bak78, QBQ89, Yua90, Bak91] or concurrent GCs [KS77, DLM+78, HK82, vdS87] (among others[8]) have been studied for long.

To mix increment and decrement messages through asynchronous communication links to maintain reference counters raises some difficulties. However elegant solutions [Bev87, Gol89, Piq91] have been proposed based on variants of reference counters that avoid the need of increment messages used by traditional reference counters. Their schemes have various properties but the "generational reference counting" scheme of Goldberg allows a node to simply obtain the total number of references on entry items. This is useful when initializing the marks of a group.

Liskov and Ladin [LL86] is one of the first published fault-tolerant distributed GC. The graph of remote references is reconstructed on a single node and scavenged there. The results are sent back to all participating nodes which can then erase useless items. This

[7] Actually new GC cycles may be started on the full hierarchy with no additional cost. However, for the larger groups, these GC cycles should not be terminated. Their partial results, i.e. the markings obtained, should simply be merged with those of the suspended GCs of the corresponding groups when the latter are resumed. The merging on each entry is done by taking the minimum of the marking levels attained by both GCs.

[8] Some of these algorithms belong to more than one category.

induces a logical synchronization of all nodes; this synchronization is made safe through assumptions on delays and global time but is not scalable to large numbers of nodes. Fault-tolerance is only with respect to the failure of the central service but does not seem to account for failures of normal nodes involved in the group GC. The algorithm makes very strong assumptions on the ability of the local GCs to detect connectivity between entry and exit items without giving references to an actual algorithm satisfying these constraints.

Hudak and Keller proposed a real-time distributed GC in [HK82]. Their algorithm performs a group GC on the whole space, operates in real-time and is furthermore able to reclaim irrelevant tasks. On the other hand, it is not fault-tolerant, imposes extra fields on every cell, and requires local mutators to strongly cooperate. Except for the reclamation of tasks, our algorithm can simulate theirs: a single group exists containing all objects, and any cell is considered to form a node by itself. Since there is a single group, reference counters are no longer useful and can be abandoned as well as the initial marking which is useless since there is nothing outside the group.

Rudalics presented a real-time distributed GC in [Rud86]. It is strongly based on a copying local GC and tolerates out-of-order message transmission. It operates on the whole space and is not fault-tolerant, on the other hand it propagates global marks faster than ours.

In [Hug85], Hughes proposed an algorithm to achieve a global GC. His algorithm is not fault-tolerant but separates well local and global GCs. His basic algorithm is similar to ours, but with a single group encompassing the whole network. On the other hand, his use of time-stamps allows him to achieve at low cost simultaneous global collections shifted in time. This keeps a continuous flow of freed memory, and does not let garbage float for very long. The drawback is that completing a GC cycle on a very large network still takes very long, and that is the time needed to actually reclaim garbage cells (it takes roughly half a GC cycle on the average). Our hierarchical algorithm also uses strictly ordered stamps, but they indicate spatial multiplexing rather than temporal multiplexing of global collectors. Though we lose on the frequency of network-global GCs, and hence on the freeing of very large cyclic clusters, we have faster and more frequent GCs for small groups, and can reclaim quickly clusters with good locality. In addition, this spatial multiplexing is the basis for the ability of our algorithm to tolerate and recover from failures. Hence we believe our algorithm to be more scalable to very large networks.

We were unable to compare our algorithm with that of [Sch89] which we do not understand enough from his paper.

Shapiro, Gruber and Plainfossé [SGP90] presented an extension of Vestal [Ves87] with many improvements and refined protocols to deal with various types of failures. They considers a single global group, and reference counters of their entry items are represented by the list of referencing nodes (for better tolerance to data transmission faults). At the end of a local collection, *soft* entry items as well as their *soft* local offspring are migrated towards other nodes so that cycles will end in a single node and will be reclaimed there (our algorithm does not need to migrate objects). The ability of their algorithm to handle various types of communication failures can be used in the reference count part of our algorithm. Its tolerance to duplicated messages would in particular increase the resilience of our algorithm against nodes where it could be erroneously programmed, for example in the case of erroneous sending of decrement messages.

7 Conclusions

This paper gives the principle of a distributed garbage collector that can collect all garbage, and in particular cyclic garbage with an efficiency proportional to the locality of the structures to be reclaimed. It has minimal interaction with the computing processes (mutators) and uses little synchronization. Its hierarchical structure makes it potentially usable for very large distributed systems.

Though we implicitely rely on many techniques that have been adequately developed in the context of other algorithms, it is clear that many points still need to be made more precise, more than was possible within the limits of this paper. In particular, several options and design choices have been left open. Further choices and refinements should depend on the characteristics of the network and (most importantly) of message transmission, the type of application programs considered (e.g. importance and size of cycles, ratio of remote references, size of cells, etc.), and on experimental results and measurements. Several aspects also need further analysis and/or proof, for example the use of concurrent collectors in local collections.

Bibliography

[Aug87] Lex Augusteijn. Garbage collection in a distributed environment. In *PARLE '87 – Parallel Architectures and Languages Europe*, pages 75–93. Lecture Notes in Computer Science 259, Springer-Verlag, June 1987.

[Bak78] Henry G Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bak91] Henry G Baker. Cantabile immobile—real-time garbage collection without motion sickness. ©1991 Nimble Computer Corporation, 1991.

[BDS91] Hans J Boehm, Alan Demers, and Scott Shenker. Mostly parallel garbage collection. In *PLDI '91 –ACM SIGPLAN Programming Languages Design and Implementation*, pages 157–164, Toronto (Ontario, Canada), June 1991. SIGPLAN Notices Vol 26 N 6.

[Bev87] D I Bevan. Distributed garbage collection using reference counting. In *PARLE '87 – Parallel Architectures and Languages Europe*, pages 176–187. Lecture Notes in Computer Science 259, Springer-Verlag, June 1987.

[Bis77] Peter Bishop. *Computer Systems With a Very Large Address Space and Garbage Collection.* PhD thesis, MIT, May 1977.

[Chr84] Thomas W Christopher. Reference count garbage collection. *Software–Practice and Experience*, 14(6):503–507, June 1984.

[DB76] Peter L Deutsch and Daniel G Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

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

[DLM+78] Edsger W Dijkstra, Leslie Lamport, A J Martins, C S Scholten, and E F M Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[Gol89] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *ACM SIGPLAN Programming Languages Design and Implementation*, pages 313–321, Portland (OR), June 1989.

[Gra78] J N Gray. Notes on database operating systems. In R Bayer et al., editor, *Operating Systems: An Advanced Course*, pages 394–481. Lecture Notes in Computer Science 60, Springer-Verlag, 1978.

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

[Hug85] John Hughes. A distributed garbage collection. In *Functional Programming and Computer Architecture*, pages 256–272. Lecture Notes in Computer Science 201, Springer-Verlag, September 1985.

[KS77] H T Kung and S W Song. An efficient garbage collection system and its correctness proof. In *Proceedings of IEEE 18th Symposium on Foundation of Computer Science*, pages 120–131, Providence (RI), October 1977.

[LD87] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN '87 – Symposium on Interpreters and Interpretive Techniques*, pages 253–263, Saint Paul, MA, 1987.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LL86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *PODC '86 – Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 29–39, Vancouver (Canada), August 1986.

[LS76] B Lampson and H Sturgis. Crash recovery in a distributed data storage system. Technical report, Palo Alto Research Center, Xerox, Palo Alto, CA, 1976.

[Piq91] José Miguel Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE '91 – Parallel Architectures and Languages Europe*, pages 150–165. Lecture Notes in Computer Science 505, Springer-Verlag, June 1991.

[QBQ89] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING sweep rather than mark THEN sweep. In *PARLE '89 – Parallel Architectures and Languages Europe*. Lecture Notes in Computer Science 365, Springer-Verlag, June 1989.

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

[Sch89] Marcel Schelvis. Incremental distribution of timestamp packets: A new approach to distributed garbage collection. In *Object-Oriented Programming Systems and LAnguages*, pages 37–48, October 1989.

[SGP90] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Research Report 1320, INRIA–Rocquencourt, November 1990.

[vdS87] Jan L A van de Snepscheut. "algorithms for on-the-fly garbage collection" revisited. *Information Processing Letters*, 24:211–216, March 1987.

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

[Wad76] Phil L. Wadler. Analysis of an algorithm for real time garbage collection. *Communications of the ACM*, 19(9):491–500, September 1976.

[WF77] David S. Wise and Daniel P. Friedman. The one-bit reference count. *BIT*, 17(3):351–359, 1977.

[Yua90] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of System Software*, 11:181–198, 1990.