European Research Seminar on Advances in Distributed Systems 1995

Title: On Comprehensive Global Garbage Detection
Subject Area: Distributed Operating Systems

Principal author: Sylvain R.Y. Louboutin
E-mail: Sylvain.Louboutin@dsg.cs.tcd.ie
Phone: +353-1-7021539
Fax: +353-1-6772204
Address: Distributed Systems Group
Department of Computer Science
University of Dublin, Trinity College,
Dublin 2, –Ireland–

On Comprehensive Global Garbage Detection

Sylvain R.Y. Louboutin* Vinny Cahill†
Distributed Systems Group,
Department of Computer Science,
Trinity College, Dublin 2, Ireland

Abstract

The experience gained with centralised garbage collection (GC) techniques has left a legacy of assumptions, expectations and tradeoffs, which may lead one to overlook some Global Garbage Detection (GGD) approaches. We argue that it is not necessary to give up on comprehensiveness in order to achieve a high degree of concurrency and scalability, although this may be at a price which is not palatable in a centralised system. For instance, much higher detection latency or space overhead, at least in terms of worst case scenarios, is acceptable in a large distributed system. We are currently implementing a variation of Schelvis' algorithm on Amadeus (taking advantage of Amadeus' object clustering ability), to conduct an empirical evaluation of the actual impact of such tradeoffs.

1 Redefining Tradeoffs

Automated GC is often advertised as a means of obviating the burden and hazard of explicit resource management, i.e., as a lesser evil which could nevertheless, under appropriate circumstances, be avoided altogether. This might be true in the context of a centralised system where each thread independently manages its own private object graph, i.e., where the visibility/accessibility and lifespan of objects does not go beyond the scope of the thread of control that created them. However, automated GC becomes a necessary and unavoidable component of a system featuring shared and/or persistent objects and a fortiori distribution. These features make manual resource management not only impractical, but impossible. This is because objects are potentially shared among independent threads of control which cannot have a comprehensive view of the overall object graph, and because objects outlive the thread of control that created them.

Although distribution can be made transparent to some extent, a direct adaptation of centralised GC algorithms would lead to unacceptable overhead. Distribution introduces additional costs such as unpredictable and unbounded delays in the delivery of messages across site boundaries and a potentially much larger object space. However it provides more available resources. Not only should the approach be different, but also the expectations put on GGD, leading to different tradeoffs. For instance, a longer latency in the detection of garbage objects could be more easily tolerated, as well as more space overheads, because resources are less likely to be scarce in such environment. In particular it should not be necessary to rule out potential approaches based on unlikely "worst case scenarios."

2 Distributed Cycles

One choice often made in order to cope with the constraints imposed by distribution, is to trade off comprehensiveness, i.e., the ability to detect distributed cycles of garbage, for weaker inter-node synchronisation constraints and a higher degree of concurrency under the assumption that distributed cycles are, in fact, relatively rare [8, 7, 2, 16].

Under this assumption, it can be considered acceptable for instance, to try to detect these rare cycles [13], by heuristically co-locating objects likely to be part of a cycle so that they can be dealt with by some local comprehensive GC algorithm à la Bishop [3].

Instead, we prefer to make no assumption about the topology of the overall distributed object-graph, and more specifically about the likelihood or rarity of distributed cycles. Actually, we contend that distributed cycles of garbage are as likely to occur as local cycles, and that other tradeoffs ought to be made.

For instance, it can be argued that replicated ob-

*E-mail: Sylvain.Louboutin@dsg.cs.tcd.ie
†E-mail: Vinny.Cahill@dsg.cs.tcd.ie

jects, make distributed cycles or cliques even more likely. Of course, the GGD can tackle them separately, but that adds to the complexity of the algorithm and does not guarantee that all other dead cycles are detected. On the other hand, an intrinsically comprehensive GGD could simply consider them as elementary objects, and would not need to have any knowledge of their semantics.

Our attention is therefore focused on algorithms which are intrinsically comprehensive, i.e., inherently able to detect distributed dead cycles. We contend that weak synchronisation constraints and a high degree of concurrency can be achieved without giving up on comprehensiveness.

3 Distributed Tracing

Detecting distributed cycles of garbage involves some form of graph tracing. Three phases can usually be identified in tracing GGD schemes [4, 15].

The initial phase builds a consistent snapshot of the overall object graph. This snapshot is subsequently traced to detect unreachable objects, while the last phase entails detecting the termination of the trace before another GGD iteration may start and resources used by garbage objects be reclaimed.

This description is meant to identify the key issues rather than trying to capture the actual sequence of events. For instance, the initial phase described above essentially consists in preventing race conditions which would compromise the safety of the GGD (race conditions between messages containing object references and messages used by the GGD). This can be achieved either via a tight synchronisation between the mutator processes and GGD [10] or by actually building a consistent snapshot of the object graph.

A snapshot can be built from scratch during each iteration of the GGD [15] or maintained, on an on going basis by a conceptually centralised service [11] (in which case no explicit termination detection phase is necessary). Alternatively, a distributed, inaccurate but nevertheless consistent snapshot, can be built incrementally via some "log-keeping" mechanism as described in Section 4.

Approaches adapted from centralised, graph-tracing GC are comprehensive and guarantee a bounded GGD latency (in terms of the number of GGD iterations). However, although multiple GGD iterations can be made to proceed asynchronously and GGD iterations interleaved [10, 9], resources cannot be reclaimed until the global mark phase is known to be complete. These approaches do not make it possible to detect the termination of a given GGD iteration based solely on locally available information. To do so, some sort of global consensus must be reached between all the nodes in the system. This constitutes a major bottleneck jeopardizing the scalability of such algorithms.

As a consequence, it may be tempting to discard all tracing-based, GGD algorithms as too costly and cumbersome. The usual taxonomy of GGD algorithms, which emphasizes the "reference counting" versus "tracing" dichotomy of centralised algorithms [1, 6], may lead one to overlook other tracing algorithms which would not be applicable in a centralised environment and therefore not necessarily identified by such a taxonomy. For instance, algorithms could be discarded because of the usual demand on the liveness property, i.e., bounded detection latency, which we contend is not justified (as long as it remains finite) in a distributed system. Algorithms could also be hastily discarded because of space overhead which would be unjustifiable in a centralised environment.

4 Log-keeping

Log-keeping is performed by the mutator and essentially entails keeping track of objects to which references have crossed site boundaries. These objects are locally considered as "alleged roots." Log-keeping makes it possible to maintain locally a conservative approximation of the root set for each individual site, thereby allowing local GC to proceed independently on each site. GGD consists in eventually ridding the alleged root set of objects which are not actually referenced remotely. It is up to the local GC to proceed with the actual collection of garbage objects. This approach has often been adapted to decentralized GGD [13, 14] and can be traced back to Bishop [3].

Log-keeping is orthogonal to the choice of GGD strategy, i.e., it does not dictate the nature of the GGD algorithm *per se*. Moreover, it does not guarantee scalability, nor does it preclude comprehensiveness. However, the choice of strategy used by the GGD to determine which of these alleged roots are actually not referenced remotely, affects the way the log-keeping is performed, as the nature and amount of information which must be logged may be different.

We distinguish two strategies for log-keeping: eager and lazy. The former attempts to update the log-keeping information as soon as possible, at the cost of additional background messages sent by the mutator. When an object reference crosses a site boundary, an

eager log-keeping mechanism attempts to immediately update the log-keeping information maintained for the target object on the site where this object is located. The latter attempts to postpone these updates as late as possible and avoids additional messages, without prejudice to the safety of the GGD (see Section 6).

The information maintained by the log-keeping mechanism constitutes a consistent, although not necessarily accurate, snapshot of the actual object graph, built incrementally as the overall object graph evolves. To guarantee its consistency, race conditions between messages containing references and background messages used for the log-keeping itself must be avoided. Otherwise live objects could erroneously be identified as garbage. This consistency constraint can therefore potentially be both costly (in terms of additional messages for instance) and complex when eager log-keeping is chosen. GGD approaches based on weighted reference counting [2, 16, 7] or reference listing [13] makes it possible to avoid this form of eager log-keeping but are not intrinsically comprehensive.

5 Consensus-free GGD Alternative

Schelvis proposed a GGD algorithm based on the asynchronous and incremental distribution of time-stamp packets [14] which seems to have been overlooked in the literature [1, 13].

Each site maintains a set of alleged roots as explained in Section 4 (or "entrance nodes" in Schelvis' terminology). Each entrance node is uniquely identified by its "time-stamp" composed of the value of a local clock at the time when the corresponding entrance node was last accessed, and the host identifier, i.e., a pair {local-time, host-id}.

"Time-stamp packets" are repeatedly and asynchronously sent to the remote entrance nodes which are transitively reachable from local roots, or local entrance nodes, via local objects. A time-stamp packet is made up of the concatenation of time-stamps of relevant entrance nodes and indicates the potential existence or the absence of a live path from some root via these nodes.

Each entrance node maintains the history of the packets it has received, and eliminates the packets which become obsolete every time it receives new packets. The algorithm makes it possible to determine whether a given entrance node is reachable from some root from the history of time-stamp packets it has received, i.e., on the basis of information available locally.

Time-stamps packets are sent from an entrance node identified as garbage before removing it, making it possible to detect dead paths[1]. The packet conceptually sent from an entrance node is derived from the largest[2] packet in its history. Distributed dead cycles (and sub-cycles) can be detected by the node with the highest time-stamp in the cycle which realises that the only packets it receives were sent by itself.

Schelvis discusses the time complexity of the algorithm, and the detection latency which is shown to be unbounded but finite despite potential transient site failures. This latency is a function of the number of GC iterations which result in time-stamp packets being distributed. It depends on the size and structure of the graph. It is for instance proportional to the number k of nodes in the cases of simple structures like a list or a single cycle, but can become $O(k^3)$ in the worst case scenario of detecting a dead distributed doubled linked list.

Although the idea may seem reminiscent of Hughes' algorithm [9], Schelvis' algorithm is different in many respects. Most importantly, time-stamp packets constitute self-contained and idempotent pieces of information about the portion of the object graph these packets have traversed. Moreover, the reachability of an entrance node can be determined based solely on the history of packets it has received, that is, on the basis of information available locally. Therefore, this algorithm avoids the bottleneck common to algorithms requiring some form of global consensus as discussed in Section 3.

However, although this point is not emphasised by Schelvis, it relies on an eager log-keeping mechanism which is potentially quite expensive and which constitutes the weakest point of the algorithm.

6 Lazy per Cluster Log Keeping

We propose a low overhead "lazy, per-cluster log-keeping" mechanism which avoids the race conditions mentioned in Section 4 but nevertheless maintains enough information to make it possible to combine

[1] Although this is not explicitly stated by Schelvis, this algorithm requires a form of eager log-keeping mechanism which goes beyond what we described in Section 4: it ensures that the history of a live entrance node contains at least one packet received when a reference to the object first crossed a site boundary, as well as packets received whenever new remote references to this entrance node were created. Additionally, whenever a reference to a remote object is removed, the target object is notified so as to update its history accordingly.

[2] The algorithm defines a total order relation between time-stamps and hence time-stamp packets.

it with an intrinsically comprehensive GGD. The idea of our log-keeping mechanism is to maintain a trail of "partial back-pointers" along the paths that references to a given object have followed during successive exchanges between sites[3].

On Amadeus [5], the overall system-wide object graph potentially spans both primary and secondary storage. Amadeus uses object clustering as a way of reducing the overhead of managing many fine-grained persistent objects. Objects are grouped within "clusters" which are the unit of (un)mapping between primary and secondary storage, i.e., respectively "contexts" and "containers." A context is a transient address space which contains a set of clusters which may vary dynamically as clusters are created, mapped into the context or unmapped from it. A container is a logically or physically contiguous area of secondary storage which stores a subset of the clusters of the systems.

The information related to the exchange of references between sites is maintained at the cluster level because the cluster constitutes the largest common denominator between both kinds of sites. Keeping information about exchanges of references among objects at the per-context or per-container level would be difficult not only because contexts are transient entities but also because of the very dynamic nature of the global object graph. Objects stored in the same container can be dynamically mapped into different contexts, and objects which were once co-located in the same context can eventually be unmapped into different containers or be migrated to different contexts.

Using clusters as the log-keeping unit makes it possible to reduce the overhead of keeping the log itself by sharing the space overhead among several objects. It also potentially takes advantage of the locality of reference within clusters, which should contribute to minimizing the amount of information the log has to keep, and reducing the complexity of the resulting graph of partial back-pointer paths rooted at this object.

Therefore the aforementioned partial back-pointers are maintained as a set of logs, one log per cluster, and must contain enough information (see Section 4) for the GGD to be comprehensive. Logically, each entry in the log maintained by each cluster, is indexed by the identifier of some object, and associated with a list of cluster identifiers. Such an entry means that this object is "known" by each of these associated clusters. A cluster "knows" an object if it either contains a reference to this object, or has an entry in its own log indexed by the identifier of this object. Thus index objects may or may not belong to the cluster where the log resides.

The log is updated whenever a reference crosses a site boundary[4]. The log can be updated either when references are exchanged as parameters in some cross-context object invocation, or when some objects that contains references, are (un)mapped.

The first time a reference to some target object crosses a site boundary, an appropriate entry can always be logged in the target's cluster. This initial partial back-pointer identifies the target as an alleged root. The log-keeping mechanism ensures that every object in the "alleged root set" (see Section 4) of some site, has an entry in a least one log located at this site. Our mechanism ensures that when this reference subsequently crosses another site boundary, that there is a co-located cluster, already belonging to the partial back-pointer path, where an appropriate entry can be logged.

The log-keeping mechanism does not try to eagerly maintain a situation whereby each entry gives a complete list of clusters which contain a reference to a given object. In other words, it does not attempt to update remote third party logs even in the case of exchanges of third party references. Thus it avoids the race conditions mentioned in Section 3. The complete list of the clusters which hold a reference to a given object can nevertheless be gathered by transitively tracing these partial back-pointer paths.

7 Related Work

Ferreira and Shapiro [8] describe a system based on a Distributed Shared Memory (DSM) model rather than the Remote Procedure Call (RPC)/object-swapping model adopted by Amadeus; it features fine-grain objects, clustered into fixed-size and disjoint, "segments." These segments are themselves logically grouped into "bunches." Bunches can be replicated and shared via the underlying weakly consistent DSM system. Garbage detection is performed at two levels; a per bunch comprehensive GC and a "scion cleaner[5]" which is not comprehensive. A heuristic is used to group bunches at one site so that a comprehensive GC can tackle cycles locally.

This approach relates to ours in the sense that both systems use some form of object clustering, and

[3] It is a *back* pointer because it leads to whatever cluster or clusters "know" the given object. It is a *partial,* back-pointer because it does not point to each individual object which holds such a reference, but to their clusters.

[4] Our mechanism traps (un)swizzling operations.

[5] A "scion" could be described as an alleged root for the bunch where it is maintained.

that log-keeping is done on a per cluster basis. However, when an inter-bunch reference is created and the bunch of the target object is not local, a "scion-message" must be sent to the target. Race conditions involved by this eager log-keeping approach, are avoided by piggy-packing the log-keeping "control" messages with the messages used by the underlying consistency protocol. Our system uses instead a lazy log-keeping approach, and is of course comprehensive.

8 Work in Progress

Amadeus differs from the model described by Schelvis [14] in many respects. For instance the explicit duality between primary and secondary storage, the transient nature of the contexts, and the fine granularity of the objects (and hence the necessity for clustering).

The per-cluster logs is used to log the history of received packets for the (public) objects of that cluster. Packets are propagated along the partial back-pointer paths so as to appropriately and progressively prune these paths of obsolete entries (for more details, see Louboutin and Cahill [12]). Contrary to Schelvis' original algorithm, time-stamp packets are not propagated, "down-stream," i.e., along the edges of the "entrance graph" (which is an abstraction of the actual object graph), but "up-stream," i.e., along partial back-pointer paths.

Our adaptation of Schelvis' algorithm requires that the GGD generates more messages than the original algorithm would require, as a result of the different strategy used for our log-keeping mechanism. We contend that shifting the overhead from the log-keeping mechanism, i.e., from the mutator processes, to the GGD is in itself beneficial to overall system performance even if it does not decrease the number of messages exchanged globally. It is, however, expected that the coarser granularity of the information carried in these messages should contribute to actually reducing the number of messages necessary for the GGD operation. Instead of indicating the potential existence or the absence of a live path from some root via a list of relevant objects/nodes, our packets indicate paths via relevant *clusters*, i.e., similar information, but with a different granularity.

We chose to not give up on comprehensiveness but nevertheless achieve a high degree of concurrency and scalability. To avoid the usual bottleneck and pitfall of tracing-based GGD algorithms, associated with having to maintain a consistent abstraction of the object graph, and having to reach a global consensus before any resource can actually be reclaimed, we are combining a low-overhead, lazy, log-keeping mechanism with an algorithm for GGD inspired by that of Schelvis. This choice is made at the cost of potentially larger space overhead, i.e., larger amount of information maintained by the log-keeping mechanism, larger contents of control messages and unbounded (but finite) detection latency, which we consider to be acceptable in the framework of a large distributed system of persistent objects. The actual implementation will make it possible to conduct empirical evaluation of this approach, and of the adequacy of our choices.

The implementation of our adaptation of Schelvis' algorithm on Amadeus is being carried out, focusing initially on primary-storage GGD. The correctness of our adaptation of this algorithm remains to be proven.

References

[1] Saleh E. Abdullahi, Eliot E. Miranda, and Graem A. Ringwood. Collection schemes for distributed garbage. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, pages 43–81, St-Malo, Brittany, September 1992. Springer Verlag – Lecture Notes in Computer Science. LNCS 637.

[2] D.I. Bevan. Distributed Garbage Collection using Reference Counting. *PARLE (Parallel Architectures and Languages Europe)*, pages 176–187, June 1987. LNCS 259.

[3] Peter B. Bishop. *Computer systems with a very large address space and garbage collection.* PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, USA, May 1977. MIT/LCS/TR-178.

[4] Anders Björnerstedt. *Secondary Storage Garbage Collection for Decentralized Object-Based Systems.* PhD thesis, The Royal Institute of Technology and Stockholm University, Electrum 230, S-164 40 KISTA, Sweden, June 1990. Available as Report No 77.

[5] Vinny Cahill, Seán Baker, Chris Horn, and Gradimir Starovic. The Amadeus GRT – Generic Support for Distributed Persistent Programming. In *OOPSLA*, 1993.

[6] André Couvert, Aomar Maddi, and René Pedrono. Partage d'Objects dans les Systèmes Distribués – Principes des Ramasse-Miettes. Rapport de Recherche 963, INRIA-Rennes, Campus

Universitaire de Beaulieu, F-35042 Rennes, June 1989.

[7] Peter William Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures.* PhD thesis, Darwin College, Cambridge University, September 1991.

[8] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, California, USA, November 1994.

[9] J. Hughes. A Distributed Garbage Collection Algorithm. In *ACM Conference on Functional Programming Languages and Computer Architecture*, pages 256–272, Nancy, France, September 1985. Springer Verlag – Lecture Notes in Computer Science. LNCS 201.

[10] Niels Christian Juul. *Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System Emerald.* PhD thesis, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Kobenhavn 0, February 1993. Rapport Nr.93/1 ISSN 0107-8283.

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

[12] Sylvain R.Y. Louboutin and Vinny Cahill. Escaping the Legacy of Centralised Garbage Collection – Towards comprehensive and scalable global garbage detection in a distributed system with fine grain persistent objects. submitted for publication.

[13] David Plainfossé. *Distributed Garbage Collection and Referencing Management in the Soul Object Support System.* PhD thesis, Université Pierre & Marie Curie – Paris VI, Paris, France, June 1994.

[14] M. Schelvis. Incremental Distribution of Timestamp Packets: A New Approach To Distributed Garbage Collection. In *Proceedings OOPSLA'89*, pages 37–48, New Orleans, October 1989. ACM.

[15] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable distributed garbage collection for systems of active objects. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, pages 134–147, St.Malo, Brittany, September 1992. Springer Verlag – Lecture Notes in Computer Science. LNCS 637.

[16] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In J.W. de Bakker, A.J. Nijmaan, and P.C. Treleaven, editors, *PARLE (Parallel Architectures and Languages Europe)*, pages 432–443, Eindhoven, The Netherlands, June 1987.