Some Key Issues in the Design of Distributed Garbage Collection and References*

Marc Shapiro
INRIA Rocquencourt and Cornell University
mjs@cs.cornell.edu

David Plainfossé, Paulo Ferreira, Laurent Amsaleg
INRIA Rocquencourt

15 April 1994

Abstract

The design of garbage collectors combines both theoretical aspects (safety and liveness) and practical ones (such as efficiency, inobtrusiveness, ease of implementation, fault tolerance, etc.). Although distributed GC is an instance of a consistency problem, practical designs often use weaker, "conservative" safety conditions, and/or weaker, "incomplete" liveness conditions. We report on our experience designing a number of distributed garbage collection algorithms in different settings, and explore the various design dimensions. The cost of each design alternative depends on the scale of the distributed system.

Garbage collection (GC) has captured the interest of the language community for years [30]. In distributed systems (*i.e.,* subject to partial failures, and where communication is costly), GC has received comparatively little attention until recently. In this paper we describe the problem and report our experience with a few solutions. Generalizing from this experience, we explore various design issues along a number of relatively independent dimensions. This analysis allows a better understanding of the trade-offs and, hopefully, to better adapt future distributed garbage collectors to their environment.

*To be presented at the Seminar on "Unifying Theory and Practice in Distributed Systems," Dagstuhl Int. Conf. and Res. Center for Comp. Sc., Dagstuhl (Germany), September 1994.

1 Distributed Garbage Collection

Garbage collection (GC) considers a program's data as a graph, in which edges represent references and vertices, objects. The *mutator* is the application program that modifies the graph by allocating new objects and assigning reference variables. Objects reachable from a distinguished "root set" of vertices (usually the stack and the variables of the program) are live; the others are *garbage*. The *collector* is the system component that detects and reclaims garbage.

A GC algorithm must be correct: both safe, *i.e.,* it reclaims only garbage objects, and live, *i.e.,* it eventually reclaims all collectable garbage.[1] However, proving correctness is far from enough. Practical issues, such as efficiency, inobtrusiveness, tolerance to faults, and simplicity govern whether an algorithm will be implemented and used.

1.1 GC algorithms and distributed systems

GC algorithms come in two main families. A *counting* algorithm counts the number of references to an object. Counting is *incomplete* because cycles of garbage are not collectable. Counting is easy to distribute because it only performs local operations. A *tracing* algorithm walks the graph to discover which objects are live (the others are garbage). Tracing is complete,[2] but hard to distribute in a large-scale system, because of its global nature, and because it has phases that need to be synchronized.

Many recent distributed garbage collection designs [14, 16, 17, 26] subdivide the system into disjoint *spaces*, each with its own "local root" and its own tracing collector.[3] An object that is the target of a cross-space reference is added to its local root-set, because it should not be collected even if it is locally unreachable.

[1] For some algorithms, which we call incomplete hereafter, not all garbage is collectable. For instance a reference counting algorithm cannot collect cycles of garbage, even though it is live.

[2] For practical reasons, some implementations called "conservative" [6] choose to overestimate the set of reachable objects; hence they may be incomplete.

[3] As we will see in Section 3, the interpretation of what a space is can vary.

1.2 GC as a consistency problem

The local collectors are not synchronized with one another. Therefore inconsistencies may arise.

Consider for instance the following example. Assume a system with asynchronous messages, and no special mechanisms to keep track of sent references. Space A holds the last reference to object x, sends it to space B, then deletes this reference. Suppose B collects before receiving the reference to x, and A collects after removing it. Then it would appear that x is unreachable although a reference to it is in transit.

A consistent snapshot would solve the inconsistency problem, but would be overkill. Because a conservative overestimate of the reachable set is safe, the distributed GC problem is easier than the general consistency problem. Inconsistencies are permissable as long as they are on the conservative side. Practical considerations favor safe but conservative (and/or incomplete) designs over strongly-consistent ones.

In summary, the management of remote references is a specialized consistency protocol, that we will call the Reference Consistency Protocol hereafter. It is composed of two sub-protocols. Recording when an object becomes remotely reachable occurs as the mutator assigns remote references; therefore it is a mutator-level protocol. Detecting when is becomes unreachable is a distributed collector, which can use either counting or tracing, or some combination.

2 Our distributed GC protocols

Before we start exploring the design space, it may be useful to consider some specific examples of collectors. This section briefly overviews two of our designs.

2.1 The SGP distributed GC protocol and SSP Chains

One design combines the Shapiro-Gruber-Plainfossé (SGP) collection protocol with the Stub-Scion Pair Chains (SSP Chains) reference mechanism. It is designed for a classical distributed system, *i.e.*, with no shared memory, partial failures, and unreliable and costly messages.

Sending the reference of some object x from its space A to some other space B creates a reference *incoming* into A and outgoing from B. The "exported" object x is added to the root set of A; exports are counted. The SGP protocol [26] actually uses a conservative, fault-tolerant variant of counting.

We call *stub* the data structure recording an outgoing reference and *scion* the one added to the local root for an incoming reference. When sending a reference, the presentation protocol (the application-level protocol for marshalling arguments into messages) creates a scion; when receiving it creates a stub. When a stub becomes locally unreachable, the local collector reclaims it. Periodically, spaces exchange idempotent live messages that list the set of stubs that are still reachable; the receiver deletes the scions corresponding to stubs that are not listed in the live message.

Message failures are tolerated by a conservative ordering of actions and by idempotent messages; race conditions are avoided by timestamping all messages and data structures, and ignoring messages that are inconsistent with the data structures. Crashes are tolerated by making space termination appear atomic with respect to reference exports.

The SGP protocol works hand-in-hand with a reference mechanism called SSP Chains [25]. This is an efficient and fault-tolerant variant of forwarders [15, 12], meaning that a reference is implemented by a chain of point-to-point links (rather than by a global identifier).

It is interesting to note that an error was found in the SSPC protocol after it had been accepted for publication [24] and a proof written [27]. The error went unnoticed because a certain configuration was implicitly, and wrongly, assumed not to occur. The bug was corrected by making the main data structures immutable, at the expense of a slight complication of the user interface [25].

We have implemented SSP Chains on a Unix system, with a graphical demonstration. Details, lessons learned, and performance results can be found in Plainfossé's thesis [22]. We are currently working on a formal proof.

2.2 Garbage collecting a distributed shared memory

Currently we are working on collecting a distributed shared persistent memory system with local caching of shared chunks of memory and a lock-based coherence protocol [11]. A reference is a virtual-memory pointer, described by auxiliary data

structures (which we continue to call stubs and scions even though they are not used as indirections).

In order to not interfere with coherence, the GC works on old data (because if an object was garbage in the past, it still is). Because a consistent snapshot would be impractical, the GC compensates for inconsistencies by being more conservative. Initially, the GC conservatively assumes that all objects in a mapped chunk are reachable. As better information is received from clients, the GC can refine its view and reclaim unreachable objects.

A global trace of the whole persistent memory (even considering only old data) would also be impractical. Therefore the GC does partial traces: for each chunk, a trace of all the cached copies of the chunk, irrespective of what site has it cached; and for each site, a site-wide trace of all chunks that happen to be cached on it at some particular instant. The site-wide trace thus reclaims cycles of garbage among groups of chunks, somewhat like Lang's mechanism [18]. The grouping is heuristic; we expect that it will be a good heuristic thanks to the well-known locality properties of caching.

3 The design space for distributed GC and referencing

This preliminary experience has us speculate on the dimensions of the design space for distributed garbage collection and referencing.

Our basic model subdivides the universe into disjoint spaces, although the interpretation of the nature of a space will vary (see Section 3.4 hereafter). Distributed garbage collection comprises three interrelated tasks: (i) tracking remote (*i.e.*, inter-space) references; (ii) detecting remotely-unreachable objects; (iii) reclaiming the garbage.

Reference tracking consists of registering references that traverse spaces boundaries. Most systems use Bishop's [4] scheme of an outgoing indirection data structure (a *stub*) at the source side locating an incoming indirection (a *scion*) on the target side.

As discussed in Section 1.2, distributed garbage detection is the problem of keeping scions consistent with stubs. As such, it is made more complex by concurrency and message and space failures (see Sections 3.1 and 3.2). Therefore, it is necessary to make explicit the failure assumptions.

Reclaiming garbage is typically done by an inferior, *local* GC that traces a single space, unsynchronized with other local collectors. A local GC interfaces with the distributed detector via the reference tracking, by considering scions as part of its root set.

Below we list some of the major dimensions of the design space. This cannot be an exhaustive list. We ignore such secondary issues as persistence, clustering, the design of the local GC, and the influence of GC on the design of applications.

Along each dimension a number of solution points are possible that differ by their functionality and/or cost. The scale of the system influences cost and feasibility of algorithms. Scale is a relative concept that is hard to characterize precisely; rather, we define scalability as a property related of an algorithm: it is scalable if its cost increases much slower than the the number of spaces or of sites in the system. For instance, tracing does not scale because of the need to synchronize with every space.

3.1 Communication semantics

The points along this axis range from instantaneous and reliable, to asynchronous, failure-prone messaging. The cost of such communication depends on the nature of spaces (Section 3.4), on the scale, and on the system configuration.

For instance, on a shared-memory multiprocessor, it is easy to make communication appear instantaneous and reliable. In a large-scale distributed setting on the contrary, the natural communication paradigm is unreliable, asynchronous messages. Furthermore, if a space can crash and recover (Section 3.2), then even so-called reliable transport protocols become lossy.[4]

Asynchronous messages and failures complicate garbage detection by introducing apparent inconsistencies. We believe that any GC algorithm could probably be made tolerant of these problems by the techniques used in SGP, *i.e.*, idempotent messages and timestamps; although to our knowledge, this has not been done for any other algorithm.

An interesting point along this axis is the causally-ordered protocols [2]. They don't scale well, but they make it easier to maintain consistency.

[4]For instance an open TCP connection must be re-established after a crash; the outcome of messages sent while the connection was breaking is unknown.

For a large-scale system, it is most reasonable to choose a communication semantics near the unreliable end of the axis; see also the conclusion of Section 3.4.

3.2 Space failure semantics

We discuss now the failure semantics of spaces. Ignoring Byzantine failures, this axis extends from failure-free spaces (or, equivalently, spaces that fail but recover intact), to fail-silent spaces, with or without consistent failure detection. This dimension is again related to scale: for instance on a multiprocessor partial failures (of a single processor) are assumed not to occur; whereas in a large-scale distributed system, a space can fail independently of the others.

Failure detection is important because objects that were reachable only from a failed space are garbage. Failure detection based solely on time-outs or channel failures is inconsistent because (for instance) a transient overload could cause some space to be considered terminated, whereas it continues to execute. Then, reachable objects could be unsafely reclaimed, as in the Network Objects system [3]. Consistent failure detection is of course harder to achieve [23].

Reference tracking (see Section 3.3) is related to failure detection, since it must distinguish a reference from a failed space. This precludes straightforward reference counting; indeed, a scion must record the name of its client space.

3.3 Reference Consistency Protocol

The Reference Consistency Protocol governs how stubs and scions are created and maintained consistent, *i.e.,* how the mutators create scions and how the partial, asynchronous GCs propagate reachability information. This item is related to Section 3.1, since it is not an issue when the communication protocol is atomic, and a causal protocol makes it simpler.

The two main points on this axis are the standard GC algorithms, counting and tracing. Most published algorithms combine counting and tracing to some degree. For instance in Dickman [7] most, but not all, unreachable scions are detected by counting; the remainder are detected by a complementary global trace.

Counting variants are either non-fault-tolerant [1, 21, 29] or fault-tolerant [26] (see related Sections 3.1 and 3.2). The former are of course simpler than the latter.

They also differ on how they deal with cycles of garbage: by migration [4], by a complementary trace [7], or not at all [1, 21, 29, etc.].

Tracing is feasible only in a small-scale system. Variants within the tracing family include global tracing [16, 28], centralized tracing [17], tracing in groups [18], and timestamped asynchronous global tracing [14]. Although all require some global synchronization, the cost of such synchronization is very variable. For instance, Lang's algorithm [18] requires a number of global barriers; Ladin's [17] works best with globally synchronized clocks (although it is not a requirement); Hughes [14] assumes a global clock and periodically executes a distributed termination algorithm.

A counting, fault-tolerant Reference Consistency Protocol is arguably the best trade-off for a large-scale system.

3.4 Nature of spaces and addressing within a space

This dimension concerns the relation of logical spaces with physical sites. The simplest model is the classical one of a space being a process confined to a single site, with no sharing of memory. A reference within a space just uses a memory address. Inter-space references (Section 3.5) are easy to track, because they are passed explicitly in messages.

A space can be composed of a small set of cooperating processes, possibly on different sites. This is the model used by many multiprocessor collectors [9, 10, 19]. As long as the set uses a single addressing scheme, and assuming no independent failures of one of the processes and reliable communication, this is very similar to the classical model. However, the costs are different because of message-passing between the processes, of synchronization of the trace phases over the set of processes, and possibly of the mapping of addresses to per-process addresses.

We know of no design where a space is a replicated set of processes. This would probably not pose any particular problem, as long as the processes in the group remain consistent.

A space may also be a shared memory region distributed over several processes or sites. If a single process can map more than one such region, the local collector can then trace all the regions currently mapped in a process. Also, the Reference Consistency Protocol must be aware of references created implicitly by reading memory (see Section 3.5). In EOS [13], the regions are logically distributed but

physically centralized. In BMX [11], a region may be cached (either partially or completely) at multiple sites at once; a consistency protocol maintains the caches coherent; the per-process GC algorithm must not interfere with the cache coherence protocol.

Spaces may also be hierarchically nested [18]. Nesting enables to better control the different trade-offs; Lang uses it to do tracing at different rates at each level of the hierarchy. Similarly, different design decisions for each of these dimensions could be used at different levels of the hierarchy.

The most appropriate design for a large-scale system is probably hierarchical. At a small scale, spaces might be distributed shared memory regions; at a higher level of the hierarchy, disjoint domains communicating with causal messages; at yet a higher level, disjoint domains using unreliable messages.

3.5 Cross-space reference scheme

This dimension (related to Section 3.3 and Section 3.4) describes the reference mechanism across spaces. A key requirement is that all remote references must be visible to the collector.

Many well-known distributed systems, for instance OSF-DCE [20], base their references on global, universally unique identifiers (UUIDs). Identifiers that can be arbitrarily manipulated by applications, as in OSF-DCE, or that can be freely stored in files, as in Corba [8], may become invisible, precluding safe GC. Furthermore, global identifiers do not scale well, because of their limited size, and because locating the target object entails a reliable distributed search in the general case. Finally, the semantics of global identifiers is ill-defined [5].

Systems with GC tend to use forwarding chains of stubs and scions [15]. Forwarding is more efficient than global identification in the general case, because a chain locates its target without any search. The rules for forwarding ensure that references are visible to the collector [25]. However, forwarding poses the problem of short-cutting long chains [12, 25].

Some systems supplement the chains with global identifiers to ease short-cutting and equality tests [15, 16], or to detect safety errors [3, 7]. Although they do not have the hiding problem of systems that expose their UUIDs, they do have the same scaling and semantic problems.

The stubs and scions are usually used as indirections [17, 25], but sometimes they

are just auxiliary data structures that describe direct pointers [11]. Although we are not aware of any examples, one could also imagine direct addressing by global identifiers similarly described by stubs and scions.

Note that short-cutting chains can be complicated if an object or a space can have more than one name or if there is no way to derive a canonical name for it; this would happen if an object could be an element of more than one space, or if the naming of spaces is not flat (Section 3.4).

4 Conclusion

We have reported on our experience with a few different designs of distributed garbage collectors and reference systems. Based on this experience, we listed some of the most important design dimensions. Our experience also suggests that, while correctness is necessary, it is far from being the only criterion in designing such systems. Although distributed GC is an instance of a consistency problem, inconsistencies that do not violate safety are acceptable. Indeed, they are close to unavoidable when considering practical issues such as efficiency, scalability, or fault-tolerance. Our experience also suggests that proofs, while necessary, must be taken with a grain of salt. GC being a conceptually simple problem, we have found a small set of example scenarios (not included here for lack of space) that capture the essence of most protocols, and can be used to test them abstractly. Applying these scenarios has been extremely useful, especially in the early phases of a design.

References

[1] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, pages 117–187, Eindhoven (the Netherlands), June 1987. Springer-Verlag.

[2] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[3] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proc. 14thSymp. on Operating Systems Principles*, volume 27 of *Operating Systems*

Review, pages 217–230, Asheville NC (USA), December 1993. ACM SIGOPS, ACM Press.

[4] P. B. Bishop. Computer systems with a very large address space, and garbage collection. Technical Report MIT/LCS/TR-178, Mass. Institute of Technology, MIT Laboratory for Computer Science, Cambridge MA (USA), May 1977.

[5] Andrew P. Black. Object identity. In L.-F. Cabrera and Norman Hutchinson, editors, *Proc. 3d Int. Work. on Object Orientation in Operating Systems*, pages 175–176, Asheville NC (USA), December 1993. IEEE Computer Society Press.

[6] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9):807–820, September 1988.

[7] Peter William Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures*. PhD thesis, Darwin College, U. of Cambridge, Cambridge, England (GB), March 1992.

[8] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Coporation, Object Design, Inc., and SunSoft, Inc. The Common Object Request Broker: Architecture and specification. Technical Report 91-12-1, Object Management Group, Framingham MA (USA), December 1991.

[9] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, pages 113–123, Charleston SC (USA), January 1993.

[10] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, Digital Research Center, Palo Alto, CA (USA), February 1988.

[11] Paulo Ferreira and Marc Shapiro. Garbage collection of persistent objects in distributed shared memory. Note technique SOR–147, INRIA Projet SOR, Rocquencourt (France), March 1994.

[12] Robert Joseph Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proc. 5th Annual ACM Symp. on Principles of Distributed Computing*, pages 108–120, Alberta, Canada, August 1986.

[13] Olivier Gruber and Laurent Amsaleg. Object grouping in Eos. In *Proc. Int. Workshop on Distributed Object Management*, pages 184–201, Edmonton (Canada), August 1992.

[14] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985. Springer-Verlag.

[15] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[16] N. C. Juul and E. Jul. Comprehensive and robust garbage collection in a distributed system. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 103–115, Saint-Malo (France), September 1992. Springer-Verlag.

[17] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Int. Conf. on Distributed Computing Sys.*, pages 708–715, Yokohama (Japan), June 1992.

[18] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, Albuquerque, New Mexico (USA), January 1992.

[19] T. Le Sergent and B. Berthomieu. Incremental multi-threaded garbage collection on virtually shared memory architectures. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 179–199, Saint-Malo (France), September 1992. Springer-Verlag.

[20] Norbert Leser. The Distributed Computing Environment naming architecture. In *OpenForum*, Utrecht (NL), November 1992.

[21] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 150–165, Eindhoven (the Netherlands), June 1991. Springer-Verlag.

[22] David Plainfossé. *Experience with Stub-Scion Pair Chains*. PhD thesis, Université Paris 6 Pierre et Marie Curie, Paris (France), June 1994.

[23] Aleta M. Ricciardi and Kenneth P. Birman. Process membership in asynchronous environments. Technical Report TR 93-1328, Dept. of Comp. Sc., Cornell University, Ithaca NY (USA), February 1993.

[24] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Symp. on Principles of Distributed Computing*, pages 135–146, Vancouver (Canada), August 1992. ACM. Superseded by [25].

[25] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1992. Also available as Broadcast Technical Report #1.

[26] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.

[27] Dennis Shasha. Proof of SSP Chains. Personal communication, October 1992.

[28] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*. PhD thesis, Dept. of Comp. Sc., U. of Washington, Seattle WA (USA), January 1987. U. of Washington Tech. Report 87-01-03.

[29] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, Eindhoven (the Netherlands), June 1987. Springer-Verlag.

[30] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag.