A Distributed GC in an Object-oriented Operating System

David Plainfosse, Marc Shapiro

A Distributed GC for Object Oriented Systems

David Plainfossé*†
Marc Shapiro
INRIA, B.P. 105, 78153 Le Chesnay Cédex, France

e-mail: David.Plainfosse@inria.fr, Marc.Shapiro@inria.fr

Abstract

We describe a distributed garbage collector protocol targeted for uncooperative distributed object oriented systems. The protocol has been implemented on distributed Lisp system and a few performance measurements are discussed. Since this implementation, we have refined deeply the protocol to address non FIFO channels. A new reference model is proposed along with the protocol to improve invocation efficiency.

1 Introduction

Our research is concerned with distributed garbage collection. Distributed garbage collection is a difficult problem that has only been addressed partially. All known algorithms that can collect cycles are either not fault-tolerant [Lang 1992a], can delay garbage reclamation [Hughes 1985a] or are not scalable [Liskov 1986a]. On another hand, incomplete techniques [Dickman 1992, Piquer 1991a] are usually more resilient to messages failures, but these benefits rely on the assumption that distributed cycles are relatively rare. Thus, designing a distributed garbage collection poses a challenging problem: reclaiming all kinds of data structures while achieving fault-tolerance, scalability and efficiency.

Our protocol (hereafter called SGP) [Shapiro 1990] addresses this challenge but fails to collect cycles. It is based on a variant of reference counting and bears some similarities with a number of other proposals [Piquer 1991a, Dickman 1992].

*Author's other affiliation: Laboratoire d'Informatique Théorique et Programation (LITP), Université de Paris VI, 75252 Paris Cédex 05

†This work is supported in part by the Ministère de la Recherche et de la Technologie

One of the requirements in designing this protocol was to achieve a high degree of scalability while keeping the overhead as small as possible. The former requirement led us to reject the use of all kinds of global synchronization. In particular, we paid attention not to rely on any kind of termination protocol, which are notoriously costly and not scalable. Our protocol is cheap and efficient and fulfills the latter requirement since it needs no additional foreground messages or systems calls, and rely on any standard communication protocol. In particular, we do not use any expensive communication mechanism such as broadcast or causal protocol.

In order to evaluate the SGP protocol, we have prototyped it on a distributed Lisp system, Transpive [Piquer 1991c], implemented at INRIA, and running on a multi-Transputer board hosted by a Sun server [Plainfossé 1992]. For the purpose of this evaluation, we replaced Piquer's original Indirect Reference Count garbage collector [Piquer 1991a], provided with Transpive, with a prototype implementation of the SGP protocol. This prototype allowed us to make several performance measurements in terms of CPU cost and message traffic. These results are encouraging but needs to be improved in order to minimize the overhead on applications.

The distributed collector interacts strongly with the invocation protocol to track object accessibility. As a consequence, we think that distributed systems should be designed to support such a service. Soul [Shapiro 1991] is the first system to integrate at the design stage, a distributed GC along with uniform lightweight references for either local or remote objects [Shapiro 1992b]. Consequently, we have refined our protocol [Shapiro 1992a] which is now tightly coupled to the reference model of Soul and relies on small number of sub-protocols.

The organization of this paper is the following. Section 2 demonstrates the necessity of providing dis-

tributed GC along with a distributed system. Section 3 overviews the SGP protocol. Section 4 describes our prototype implementation and presents some performance measurements. Finally, Section 5 describes several refinements added to the SGP protocol and the interaction with the reference model.

2 Motivations

Manual reclamation is an error-prone, time consuming activity. Moreover, recent studies [Delacour 1991] argue that it also reduces program re-usability. Consequently, the need for garbage collection in programming languages is now widely accepted. Recent proposals to include GC in C++ [Edelson 1992, Ferreira 1991] are part of this trend. Designing a collector for such a language poses a challenging problem: accessing type information at run-time while achieving acceptable overhead. The former problem is further complicated in languages such as C or C++ where pointers may be subverted. Along with these two majors goals, the collector should not be intrusive nor impose restrictions on the programming language.

Similarly to programming languages, we think that distributed systems should provide transparent storage management. Most of distributed systems provide transparent location and invocation mechanisms but leave users to deal with reclamation of garbage objects. In contrast, we think that distribution transparency necessitates distributed GC.

Moreover, manual reclamation of objects in distributed environment can become intractable. Distributed systems provide a distributed computation model where processes run in parallel on different spaces or machines. Processes share objects through remote references, that may cross space boundaries. In such an environment, deciding if an object is garbage or not is definitely a harder task than in centralized one. A single object can be remotely referenced from a number of different spaces. In this case, reclaiming an object requires cooperation between all the spaces which hold references to it. If the heap is shared by many applications written by different programmers, if it is accessed in parallel, if it also includes disk storage (as in persistent object systems) and distributed access , then manual resource management is simply out of the question.

The problem is further complicated by considering common message failures such as lost, duplicated, or out-of order messages. Each of these failures may invalidate the liveness or safety property of the distributed a GC algorithm. For instance, in order to discard remote objects it is necessary to exchange background administrative messages. The loss of such a message may lead to never collect a garbage object. On the other hand, a duplicated message may lead create a dangling remote reference.

3 Overview of the SGP Protocol

The application (mutator) rests upon two separate layers of object management (see Figure 1). The bottom layer is independent of object semantics, structure, or programming language: this is the distributed GC protocol specified in [Shapiro 1990] and described briefly here. The distributed collector propagates accessibility information supplied by the upper layer.

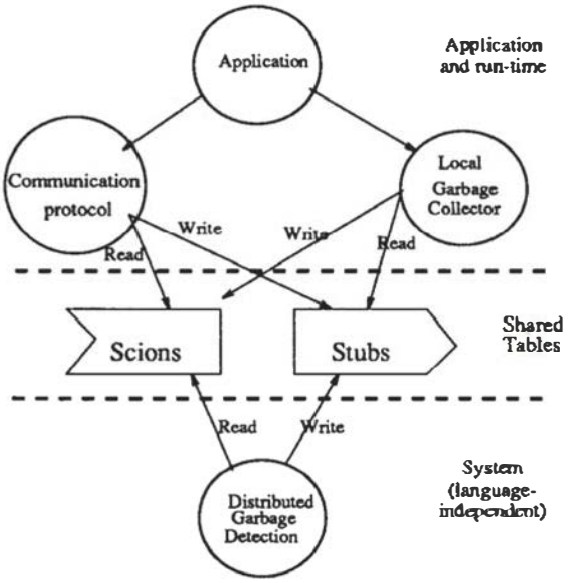

Figure 1: System layering

The upper layer is a language-specific run-time, extended to interface with our distributed GC. In the upper layer, one finds storage management (object allocation, and local tracing garbage collection) as well as remote invocation functions (i.e. communication stubs).

The two layers share information in the form of lists of incoming and outgoing references. Cooperation between layers is limited to simple interactions to maintain safe consistency between those lists.

Mutators in different spaces communicate by RPC-style invocation, i.e. by messages. An invocation is mediated by communication protocol for marshalling

and unmarshalling messages; the communication protocol is the interface between the application and the system, encoding typed information into a typeless form. The arguments and results in an invocation contain any mixture of pure data, references, and migrating objects. When sending or receiving a message, the communication protocol first writes information about the remote references or objects into the tables that are shared between the two layers.

To provide fault tolerance, extra time and ownership information is piggy-backed onto the existing mutator messages. Occasional control messages are exchanged, in the background, to remove inaccessible scions.

The SGP protocol relies on the existence of a standard local tracing garbage collector. The distributed protocol is based on a conservative extension of reference counting.

Each space maintains a list of potential incoming and outgoing references, respectively *scions* and *stubs*. Both the scions list and the stubs list are conservative estimates. If two diff spaces possibly refer to a single object of space A, each will be assigned a scion in space A. This differs from reference counting because we need an entry per remote space to deal with unreliable communication. This policy renders scion deletion an idempotent operation and permit tolerating lost or duplicated message. In the former case any subsequent control messages received will allow us to reclaim previously garbage scions. The latter case will have no effect since all garbage scions would have been been previously collected.

Local garbage collection proceeds from the union of the local root and the scions list and removes objects and stubs. Since local GC starts from the union of the local root with the (conservatively estimated) scions, all non-reachable local objects are true garbage. Each local GC removes useless stubs. In turn, stubs are used to clean remote scions, yielding successively better estimates.

When a stub on space A is deleted, the corresponding scion on space B can be removed. To this effect, a *live message* is sent from space A to space B. However this message can be duplicated or lost.To guard against loss, periodic *live messages* are sent from A to B containing the list of all *existing* stubs pointing to B; by comparison space B can deduce scions that are not reachable, and remove them.

One common problem in distributed systems is the message delivery delay. Suppose that one space B sends a message to a space A containing a reference to a given object, say x. At the same time, a delete message is sent from space A to space B to inform that the remote pointer on object x has been discarded. If object x is not locally referenced upon receiving the delete message, it will be remove from the scions list and collected at the next local GC.

To avoid this problem, we keep on each space a vector of highest timestamps and we timestamp entry items. When sending a reference, the stub creates the entry items and store in it the value of the local clock. The same value is used to timestamp the mutator message. Upon receiving a mutator message, the receiver compares the timestamp value extracted from the message with the one found in the vector of highest timestamps. This vector contains a space identifier and an associated timestamp for each remote space. A timestamp is increased each time a message is received. If the corresponding entry in the vector does not yet exist the initial value can be taken from the message. live messages carry the current value of the timestamp vector corresponding to the target space. Upon receiving a live message, the timestamp value found in the message is compared to the value in the entry items to detect messages in transit.

4 Prototype on a Distributed Lisp System

We have experimented with the SGP protocol on Transpive a distributed Lisp system [Piquer 1991c]. The choice of Transpive allowed us to quickly implement the SGP protocol and to learn a few lessons, although the distributed model of Transpive is quite different from SGP's.

In this section, we compare the measured performance of our prototype with Piquer's Indirect Reference Count (hereafter called IRC), in terms of communication and CPU overhead. Our measurements of two applications (merge sort and matrix multiplication) were taken on a Parsytec board composed of four T800 Transputers with one megabyte of memory each, hosted in a Sun. Each application was timed twice in a row; the figures are better the second time because of Transpive's caching policy. The measurements, repeated dozens of times, have shown extremely low variance. Our experiments were able to test resilience to message loss but not to termination, due to lack of a fault-tolerant application. Furthermore, we were not able to quantify how conservative or how scalable our protocol is.

Table 1 shows local execution times. The overhead is due to management of (the Transpive equivalent

Application	CPU time in seconds						Overhead	
	No DGC		IRC		SGP		SGP / IRC	
(sort 100)	3.8	3.2	4.7	3.9	5.5	4.1	1.17	1.05
(sort 200)	5.6	4.4	6.7	5.2	8.1	5.9	1.20	1.12
(mult 20 20)	11.1	7.8	12.1	8.7	13.5	9.8	1.12	1.12

Table 1: Execution Times

Application	Control Messages					
	IRC		SGP		SGP / IRC	
(sort 100)	31	28	10	8	0.32	0.28
(sort 200)	41	39	10	8	0.24	0.20
(mult 20 20)	101	96	20	18	0.20	0.19

Table 2: Message Overhead

of) stubs and scions. Our implementation is on average 10% slower than IRC and 20% slower than with distributed collection turned off. This result is encouraging: our implementation is not optimized and retained some obsolete data structures and processing from Piquer's implementation. Futhermore, the fault-tolerance property of the SGP protocol requires a some additional work compared with Piquer's approach that largely justifies some added cost.

A second measurement concerns the number of control messages sent and their frequency. Our message sending protocol is different from Piquer's and slows down local processing a little because we group stubs into a single structure, instead of sending a unique stub per control message. We have chosen the former policy because it reduces message traffic. As shown on Table 2, this "buffering" strategy reduces dramatically the number of control messages sent in SGP compared with IRC protocol. Note that the number of control messages sent does vary a little between the two executions. Note also that we obtained the same results whatever the size of list in the merge sort application. It shows that our message sending policy is somewhat independent of the number of objects sent between spaces.

The original SGP model did not take into account the replication of objects. Consequently, we have adapted the SGP protocol into the replication model of Transpive. This has the result that the collection of scions is more complex and slower than we expected. This increases the conservative aspect of SGP and can be troublesome if memory is heavily in demand. A way to decrease the delay for collecting scions is to increase the sending frequency of control messages. Consequently, there exists a tradeoff between buffering policy and the frequency of control messages.

Moreover, the fine grained sharing support of Transpive is definitely an uncooperative environment for distributed GC. In particular, memory consumption is heavy since it requires a huge number of entries in the control data structures. Consequently, it increases the overhead of SGP on application and the frequency of local GC.

The performance results are encouraging but need to be improved , to minimize the overhead on applications. The buffering policy dramatically reduces the number of control messages. The resilience to message failures has been successfully demonstrated. This result validates our design guideline. However, fault-tolerance to space failures and to duplicate messages remains to be investigated. Although, the SGP design relied on very different distributed programming model from Transpive, the prototype behaves correctly with respect to the safety property. That is, it does not reclaim any accessible objects. It demonstrates that the SGP protocol is generic and adaptable. Therefore, it is a good candidate for a system service.

5 Refinements to SGP protocol

Since our implementation, we have added several refinements [Shapiro 1992a] to the SGP protocol. These refinements concern both the reference model and the management of delayed messages. We also improved the design of the distributed GC itself that is now divided into a small number of sub-protocols. Each sub-protocol encapsulates a piece of work and collaborates with others. Figure 2 shows the different sub-protocols and their relationships with the main control data structure.

As shown on Figure 2, we distinguish two kinds of sub-protocols: the mutator protocols and the GC protocols. The former are responsible for maintaining conservative consistency when passing references from one space to another. The latter track global accessibility and tighten the consistency removing unreferenced scions entries.

Application send and receive messages using a low-overhead "presentation layer" protocol, with scions

and stubs created or updated automatically as needed. The marshalled form of a reference is a locator.

The **transport** protocol describes the way messages are handled (under what circumstances messages are discarded for example.

The **presentation** protocol describes the way reference are marshalled into and unmarshalled from messages.

The **invocation** protocol details the way in which a reference is used. That is, how locating of the target object interacts with the activity of invoking its methods.

The **cleanup** protocol is responsible for periodically sending control messages containing the whole list of lived stubs.

In this section certain key features of the mechanism are hilighted and discussed in some detail. Particularly, we describe the main differences with the SGP protocol and we highlight the interactions between the reference model and the distributed GC activity. For the sake of brevity we omit voluntary some details of the protocol.

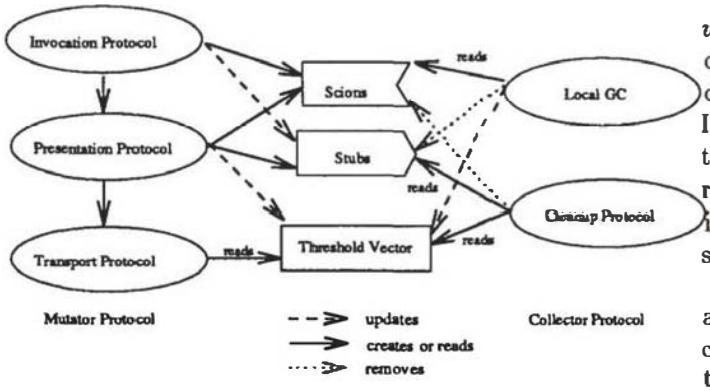

Figure 2: Relationships between sub-protocols and main data structures

5.1 Reference Model

We have modified the reference model in order to improve accessibility and invocation efficiency. In the original model, objects were only accessible through a chain of stubs and scions potentially crossing several spaces (hereafter called indirection spaces). We noticed two drawbacks with this scheme. First, the failure of an indirection space prevented access to objects. Second this scheme was totally inefficient with respect to reference passing. Precisely, passing a reference to a chain of stubs and scions resulted in the creation of a new pair of scion, stub on each indirection space. The new model solves both problems and improves invocation efficiency. A stub contains now two kinds of locators (hereafter called *locator*). The GC invariant enforces that it exist always a chain of strong locator (in absence of space failures) between the source and the target object.

The strong locator serves only distributed GC purpose and is never used for invocation. Instead invocations use always the *weak locator* that shorcuts the chain of stubs and scions. This weak locator improves resilience to space failures since accessibility is not entailled by indirection space failures. It also improves invocation efficiency because it can access objects in a single hop.

5.2 Transport Protocol

The transport sub-protocol is responsible for sending and delivering messages from one space to another. It also determines the circumstances under that a reference may be rejected. This sub-protocol has been improved on the SGP proposal in order to extend the window of acceptance of delayed messages.

We keep now a vector of timestamps, the *threshold vector*, to deal with delayed messages and keep stub creation is idempotent. Each threshold vector entry contains a space name and an associated timestamp. In the SGP proposal, we increased the matching entry of the threshold vector each time a message was received. Messages were only delivered to application if they carried a timestamp greater than the corresponding entry in the threshold vector.

Actually, a message must be rejected only when acting upon it is not idempotent. That is when the corresponding stub has been discarded. Consequently, threshold entries are now updated each time a stub is reclaimed instead of each time a message is received. The local GC sub-protocol is responsible for reclaiming stubs and therefore is in charge of updating threshold entries on the basis of timestamps found in garbage stubs. This new policy requires us to keep timestamp value in stubs. Therefore, it increases a little the space cost of the algorithm.

5.3 Cooperation with Local GC

Since our distributed GC relies on local GC we needed to find an easily adaptable local GC for C++. Local garbage collection for C++ is a difficult problem that has not been successfully addressed yet. All known proposals have majors drawbacks since they are either not efficient enough [Edelson 1991], impose language restrictions over inheritance and polymorphism [Bartlett 1989], or are too intrusive to be widely

used [Ferreira 1991]. A recent proposal [Edelson 1992] matches our requirements: easy to use, efficient, and providing a finalization mechanism.

First, the root set must be extended to take into account the scions list. A straightforward way of extending the roots set consists in keeping a head reference to the list of scions in the address space of each application. A better solution is to store scions in a fixed memory area known to the local GC. This area should be write protected to protect scions against undesirable modifications from applications.

The local GC is responsible for reclaiming garbage stubs. Stubs must be finalized for two reasons. First, as stated in Section 5, the threshold vector must be increased when a stub is reclaimed. Secondly, stubs are linked through weak pointers and the linked list must be updated to remove deleted stubs. This latter work may be handled in two ways. First, the local GC may update the list each time it reclaims a stub. This solution requires either a finalization mechanism or double linked list. The double linked list is space consuming while the finalization is time consuming since it requires to traverse the whole list. A better way is to delay stub reclamation to the next traversal of the list by cleanup protocol. The local GC only marks stubs as deleted and the cleanup protocol update the list when it traverses it. This solution avoids costly mechanisms as well as synchronization problems between the local GC and the cleanup protocol.

6 Related Work

Distributed garbage collection is a difficult problem which has only been addressed partially. One key reason is that while most proposals rely on centralized techniques, adapting such techniques to distributed environments is not a straightforward task. Stop the world algorithms require costly termination mechanisms when facing distribution, whereas reference counting is completely defeated by common messages failures. In order to adapt those techniques to distributed environments, many recent proposals try to relax traditional invariants [Dickman 1992, Piquer 1991b, Watson 1987] whereas others rely on reliable communication protocols [Hughes 1985b, Lang 1992b, Mancini 1991]. The former family algorithms is usually based on reference counting. Therefore they cannot garbage collect distributed cycles and must assume that such graphs are rare. The second family ensure better liveness but all known algorithms are not resilient to message failures [Lang 1992b], may be completely defeated by space failures [Hughes 1985b], or fail to address large network [Liskov 1986b]. Our protocol belongs to the former family and bears some similarities to a number of proposals based on reference counting [Dickman 1992, Piquer 1991b]. Unlike those approaches, however, we maintain an entry item per source space that permits us to tolerate message loss whilst avoiding the dangers of duplicated messages.

Dickman [Dickman 1992] proposes *Optimizing Weighted References Counting* improving traditional *Weighted Reference Counting* [Bevan 1987, Watson 1987] in two aspects: message failures resilience and indirection cells. Resilience to message failures is provided through a weak invariant that requires that each object weight (total weight) is always greater or equal to the sum of all remote reference weights (partial weights). The weak invariant permit tolerating message loss but duplicated message remains problematic. The algorithm avoid the creation of indirections cells when partial weights cannot be split. However, this is enforced through a special `null weight` value. In this case, the total weight is always greater than the sum of partial weights preventing the object from being reclaimed by error. However, liveness is not ensured for *weak* objects which conform only the weak invariant. For this reason, the author assumes than the algorithm is always used in conjunction with a global tracing collector to reclaim garbage distributed cycles and weak objects.

In [Piquer 1991b] Piquer describes his Indirect Reference Count (IRC) algorithm which improves Weighted Reference Count [Bevan 1987] by avoiding indirection cells. The algorithm also eliminates the need for increment messages that may conflict with decrement messages in traditional schemes. Thus, creation and duplication of a remote pointer are performed locally without informing the space where the object is located. In order to achieve local creation/duplication, remote pointers have been extended with a new field, named an indirect pointer. The indirect pointer serves only distributed GC purposes, and refers either to an object or to another remote pointer. The whole set of remote pointers referencing a single object forms a distributed graph which can be traversed using indirect pointers. Mutators never use indirect pointers, instead relying on the direct pointers to access objects in a single hop. As with others proposals relying on reference counting, the IRC algorithm is not resilient to message failures: liveness is not enforced against message loss and safety is not preserved against duplicated message.

Hughes [Hughes 1985b] describes an elegant algo-

rithm based on timestamps and local tracing. The algorithm timestamps objects and relies on the premise that garbage objects' timestamps remain constant whereas non-garbage objects' timestamps increase monotically. A timestamp threshold is computed to distinguish garbage from non-garbage objects. Objects that carry timestamps less than the threshold can be safely reclaimed. Unfortunately, the threshold computation relies on a termination algorithm which is notoriously costly and not scalable. Moreover, the algorithm is not resilient to space failures since a failed space prevents increasing the threshold, hence blocking garbage collection on all other nodes.

In contrast to many proposals that attempt to compute on each space the global accessibility of objects. Liskov and Ladin [Liskov 1986b] rely on their highly available centralized service to compute global accessibility of objects on a single space. This service is physically replicated, hence achieving high availability and fault-tolerance. All objects and tables are assumed to be backed up in stable storage. Clocks are synchronized and message delivery delay is bounded. These assumptions allow the centralized service to build a consistent view of the distributed system. Each local collector informs the centralized service about incoming and outgoing references, and about the paths between incoming and outgoing references. The path computation is expensive but necessary for reclamation of distributed garbage cycles. Based on the paths transmitted, the centralized service builds the graph of inter-site references, and detects garbage (including dead cycles) with a standard tracing algorithm. The centralized service informs LGCs of accessibility of objects.

In a later paper [Ladin 1992] Ladin and Liskov simplify and correct the deficiencies of the above proposal, adopting Hughes' algorithm and loosely synchronized local clocks. Hughes' algorithm eliminates inter-space cycles of garbage, thereby eliminating the need for an accurate computation of the paths and for the central service to maintain an image of the global references. Furthermore, the centralized service determines the garbage threshold date, making a termination protocol unnecessary.

Recently Lang *et al.* [Lang 1992b] describe an original proposal to combine reference count and mark and sweep. The algorithm collect distributed cycles within predefined groups. Groups are dynamic collections of spaces (i.e a space may be removed or added during garbage collection) and may overlap or include other groups. The algorithm relies both on counters and local GC to perform mark and sweep within a group. Reference counts must be kept accurate, hence message failures are not tolerated. Group GC is conservative with respect to inter-group references: any subgraph referenced from outside the group is not collected until a larger group is formed encompassing the entire graph; therefore liveness is not guaranteed. Thus, large cycle reclamation requires extending group size such that the group includes all spaces that hold a cycle vertex. Distributed garbage collection of very large networks is proposed through a hierarchy of included groups. Included groups benefit from larger groups GC that perform some of their work. However, large group GCs are longer than smaller ones and therefore retain more floating garbage. For that reason, the authors assume that large group GCs are rare compared to small group GCs.

In [Lins 1992] Lins and Jones combine *Weighed Reference Counting* with Lins'local algorithm for *Cyclic Reference Counting* [Lins 1991] to address distribution issues. As a result, they propose a simple algorithm to garbage collect cycles in a distributed environment. The general idea of the algorithm is to perform a local mark-scan whenever a reference to a shared graph is deleted. That is, a mark-scan is initiated each time an object is suspected of belonging to a garbage cycle (i.e when its counter is decremented down to one). The mark phase decrements counters each time it visits an object belonging to the subgraph. At the end, all nodes with counters equal to zero are part of a dead cycle and may be safely reclaimed. Lins [Lins 1991] improves the basic idea to perform the mark-scan lazily. Spurious objects are not scanned at once but instead they are queued in a special list. When the allocator fails to supply memory the corresponding list is scanned in order to reclaim potential garbage cycles. Unfortunately, mark-scan of subgraphs must be computed in critical sections. In other words, two different spaces cannot invoke cycle detection concurrently.

7 Conclusion

Most of the proposals for distributed garbage collectors have never been implemented. In contrast, we have experimented with the SGP protocol and the performance measurements are encouraging. We have aslo tested the resilience to messages failures but space failures and duplicated messages remain to be investigated. We have added several refinements to SGP protocol. The design of the protocol now relies on small number of sub-protocols that collaborate with each other to provide distributed GC. We have also described the representation of a remote reference which

is now composed of two locators. The distributed GC invariant preserves access to remote objects through a chain of strong locators. The invocation protocol uses weak locators that shortcut the chain of strong locators. This new design is part of the Soul project that aims at providing efficient and lightweight references along with distributed GC. A prototype implementation of Soul is underway.

References

[Bartlett 1989] Joel F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical Note TN-12, Digital Western Research Laboratory, Palo Alto, CA (USA), October 1989.

[Bevan 1987] 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.

[Delacour 1991] Vincent Delacour. *Gestion mémoire automatique pour les langages de programmation de haut niveau.* PhD thesis, Université Paris-6, Pierre-et-Marie-Curie, Paris (France), jun 1991.

[Dickman 1992] Peter Dickman. Optimising weighted reference counts for scalable fault-tolerant distributed object-support systems. (submitted to publication), 1992.

[Edelson 1991] Daniel R. Edelson and Ira Pohl. A copying collector for C++. In *Proc. of the 1991 Usenix C++ Conf.*, Washington DC, (USA), April 1991.

[Edelson 1992] Daniel R. Edelson. A mark-and-sweep collector for C++. In *Principles of Programming Languages*, pages 51–57, Albuquerque, NM (USA), January 1992.

[Ferreira 1991] Paulo Ferreira. Reclaiming storage in an object oriented platform supporting extended c++ and objective-c applications. In *Proc. of the International Workshop on Object-Orientation in Operating Systems*, pages 100–102, 1991.

[Hughes 1985a] 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.

[Hughes 1985b] 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.

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

[Lang 1992a] 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.

[Lang 1992b] 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.

[Lins 1991] R. D. Lins. Cyclic reference counting with lazy mark-scan. Technical Report TR-77, University of Kent, Computing Labotory Canterbury (England), August 1991.

[Lins 1992] R. D. Lins and R. Jones. Cyclic weighted reference counting. Technical Report TR-95, University of Kent, Computing Labotory Canterbury (England, March 1992.

[Liskov 1986a] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *podc5*, pages 29–39, Vancouver (Canada), August 1986. ACM.

[Liskov 1986b] 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.

[Mancini 1991] L. Mancini and S. K. Shrivastava. Fault-tolerant reference counting for garbage collection in distributed systems. *The Computer Journal*, 34(6):503–513, December 1991.

[Piquer 1991a] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In *Pro. 1991 Principles of Programming Languages*, feb 1991.

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

[Piquer 1991c] José M. Piquer. *Parallélisme et distribution en Lisp.* PhD thesis, Ecole Polytechnique, Massy France, January 1991.

[Plainfossé 1992] David Plainfossé and Marc Shapiro. Experiments with a fault-tolerant garbage collector in a distributed lisp system. (submitted to publication), 1992.

[Shapiro 1990] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, inria, rocquencourt, November 1990.

[Shapiro 1991] Marc Shapiro. Soul: An object-oriented OS framework for object support. In *Workshop on Operating Systems for the Nineties and Beyond*, pages 251–255, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

[Shapiro 1992a] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust distributed references and acyclic garbage collection. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, 1992. (to appear).

[Shapiro 1992b] Marc Shapiro, Julien Maisonneuve, and Pierre Collet. Implementing references as chain of links. (submitted to publication), 1992.

[Watson 1987] 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.