A Scalable Cyclic Garbage Detection Algorithm for Distributed Systems

R. Kordale* M. Ahamad

September 8, 1993

Abstract

We present a scalable algorithm for detecting inter-node cycles of garbage in a distributed system. We do this by extending an available scalable and fault-tolerant distributed garbage detection protocol without sacrificing on efficiency and scalability of the original protocol.

*Please direct future correspondence to R. Kordale. e-mail: kram@cc.gatech.edu. Postal address: 33370 Georgia Tech Station, Atlanta, GA - 30332, USA. Tel-no: (404) 894-6169. Fax-no: (404)-853-9378. (My name should appear in the message header.)

1 Introduction

Distributed garbage collection is a difficult problem which has only been addressed partially in that most existing proposals do not hold up to all the requirements of distributed garbage collection such as:

1. low space, communication, and synchronization overheads;
2. tolerance of non-ideal communication channels and node failures and;
3. detection of inter-node cycles of garbage.

Most proposals relying on reference counting cannot collect inter-node cycles of garbage[Piq91]. The version of a fault-tolerant distributed garbage collector[MS91] that is efficient does not collect inter-node cycles either. Some tracing algorithms like those described in [Hug85] and [JJ92](the current prototype) require some form of distributed termination detection which can impact scalability. The technique outlined in [LL92] collects inter-node cyclic garbage efficiently but requires a bound on message delays, and loose synchronization of clocks. Lang et. al.[LQP92] describe an algorithm that groups nodes for purposes of garbage collection. Distributed cycles within such groups are detected. Node failures are handled by either waiting for the node to come up or by negotiating a new group that excludes the failed node(s) and restarting a new group GC from scratch. One of the disadvantages of this algorithm is that large cycle reclamation requires extending group size such that the group includes all spaces that hold a cycle vertex. The larger the groups are, the longer the group GC takes, and higher the probability that some node in the group fails before the end of the group GC. We would like to note here that we are not looking into schemes that address GC particularly in persistent, distributed systems though some of the schemes described above can be used in such systems.

The SGP protocol(as called by the authors in [PS92]) described in [SGP90] does well on most counts and is one of the few ones that have been implemented[PS92]. It can detect garbage in a distributed system subject to common failures such as lost and duplicated messages, network partition, site and disk crashes etc. The protocol uses only information local to each site, or exchaged between pairs of sites. No global mechanism is necessary; so, it should scale well. However, the sub-protocol to collect inter-node cycles of garbage either requires an adaptation of distributed termination detection protocols(might not scale well) or it requires migrating objects belonging to a cycle to other nodes so that cycles will end in a single node and will be reclaimed there. It is clear that moving objects physically as in [Bis77] or logically as in [Ves87] is undesirable. We propose a scheme that extends the scheme given in [SGP90] naturally and without migrating objects; the existing scheme is used except that extra information is passed around and one extra type of message is used.

The paper proceeds as follows. First, we introduce briefly the SGP protocol. In section 3, we provide the intuition behind our solution and finally we present an informal algorithm that collects inter-node cycles of garbage.

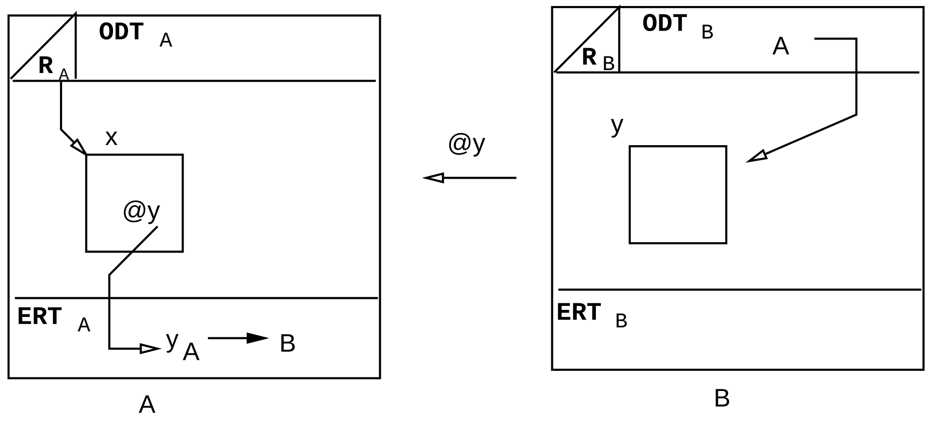

Figure 1: Notation

2 The SGP protocol

The universe of objects is subdivided into disjoint *spaces*. At any time, an existing object is either located in some space or in transit between spaces. Each space performs a standard local tracing garbage detection, independently of the activity of remote or global detectors. The root for local collection($RootSet$) is the union of the local root and the ODT(see below). Communication between *mutators* in different spaces occurs via messages.

They distinguish between a *local* reference(to an object known to be in the same space) and a *remote* one(to an object thought to be in another space). A remote reference is represented as a local reference to a *stub* object, which contains the last known location of the referent. Stubs are collected in a structure called the *External Reference Table*(ERT). Conversely, local objects potentially accessible from other space are listed in an *Object Directory Table*(ODT)(See figure 1).

The detection protocol is based on local marking garbage collectors, and on the interaction of a number of sub-protocols: an object finder protocol, a reference-sending protocol, an object-migration protocol, an interaction protocol between local collectors, and a cycle detection protocol. Furthermore, fault-tolerance is attained by refining the finder with a deletion protocol and a termination protocol. To explain our sub-protocol to detect cycles, we will explain only the interaction protocol between local collectors. We refer the reader to [SGP90] for the other sub-protocols.

Local garbage collection of space A starts from the root which is the union of the local root(R_A) and the ODT(ODT_A). At the end of the local GC phase, all stubs(ERT entries) that are unreachable are removed and to announce stub removal, spaces exchange background control messages such as: $A \rightarrow B$: ERT$\{ERT_{A|B}\}$[1], where $ERT_{A|B}$ denotes

[1] A message from space A to B is noted as $A \rightarrow B$: **type**$\{contents\}$, where **type** is either **mutator** or a specific control-message type.

the subset of stubs of ERT_A pointing to space B. If some ODT entry $ODT_B = (A@x)$ does not appear in $ERT_{A|B}$, then that entry can be removed. Thus, global garbage detection occurs by pairwise cooperation between spaces. Local GC conservatively updates the ERT; each such ERT is in turn used to conservatively create new versions of the ODTs. Local GC contributes to clean up the local ERT, and hence remote ODTs. We will now address the problem of cycle detection.

3 Inter-Space Cycles of Garbage

Detecting inter-space cycles of garbage is a problem because maintaining ODT entries is very similar to maintaining reference counts. Consider the example of references given below: $O_A \rightarrow O_B \rightarrow O_A$. Suppose that O_A and O_B belong to spaces A and B respectively and that these are the only references with respect to these objects. ODT_A contains an entry for O_A and ODT_B has an entry for O_B because of inter-space references $O_B \rightarrow O_A$ and $O_A \rightarrow O_B$ respectively. Even if neither of the objects is a local root of any node, they will not be detected as garbage objects since ODT entries are considered part of the $RootSet$. The protocol we present is similar to the interaction protocol in which ERT entry removals are announced based on which some ODT entries are removed; to detect cyclic garbage, our protocol propagates some more information as shown below.

The basic idea of our sub-protocol is to propagate information about the origin of traversal to all the objects. If an object is reachable only from ODT entries and not from any root object then the object is not *live*. Two obvious optimizations are possible. Firstly, information about the root of traversal can be passed only to objects for which entries exist in some ODT(i.e. passed only at inter-space boundaries). Secondly, root information can be passed during the traversal phase of GC. Consider the following example. Suppose that objects $a_1, a_2, ..., a_m$ are in space A and objects $b_1, b_2, ..., b_n$ are in space B and the set of references $a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_m \rightarrow b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_n$ exists. Suppose that a_1 is a local root of space A. Our proposal is that maintaining this information, henceforth referred to as *root information*, only in the ODT_B entry for b_1(because of the inter-space reference, $a_m \rightarrow b_1$) is enough. This is because traversal from the root object is interrupted only at inter-space boundaries. Thus, when traversal continues on the other space, the root information stored in the ODT can be used.

The only complication is the following. Suppose that objects a, b, and c belong to different spaces and the only references in the object graph is $a \rightarrow b \rightarrow c \rightarrow b$. Suppose a is a root object. Note that when an inter-space reference is created, root information might not be known and thus will have to be left blank in the ODT entry. In the example scenario, root information flows from a to b, then from b to c, during traversal, which are noted in the ODT_B entry for b and the ODT_C entry for c. However, traversal continues from c to b and thus root information flows from c to b; this must be discarded or else it would be wrongly learnt that b is reachable from a on two paths. At the same time, such information cannot be blindly discarded either since c, and therefore b, can be reachable from a on two paths(with an additional reference: $a \rightarrow c$). We describe our solution briefly below. For the complete algorithm, the correctness proof and a detailed example, we refer the reader to [KA93].

During traversal, when a piece of information received by a space is suspected to be a duplicate, a *probe* entry is installed in its ODT and a *probe* message is propagated exactly the same way as root information. If the same message is received by the node later on, it becomes clear that this information is going around a cycle and so the previously installed *probe* entry is removed from the ODT. The mechanics of propagating root information, installing a probe and propagating the probe message is achieved by a change in the information contained in the data structures and handling one extra type of control message.

We show the extensions required to the data structures by showing the formats of the extended data structures. For purposes of comparison, we will show the old and the new formats. If O_A is in space A, O_B is in space B, and the reference $O_A \rightarrow O_B$ exists, then ODT_B contains $< O_B, A >$ and ERT_A contains $< O_B, B >$. With the extensions, the following entries are maintained: ODT_B contains an entry $< O_B, O_r, O_p, O_A, A >$ and ERT_A contains an entry $< O_B, O_r, O_p, O_A, B >$, where O_r is a root from which O_A is reachable and O_p contains the identity of the probing object. Of course, if the root is not known(as can be the case until root information propagates during traversal), O_r is ϕ; also, if no object from O_r down to O_B suspected the information to be duplicate(spurious), then O_p is ϕ.

The local GC algorithm is almost the same. The only differences result because extra information needs to be stored into ERT entries. However, the interaction protocol between local collectors has some changes and so only this is shown in figure 2. For purposes of presentation, we maintain the previous copy of ERT before the latest traversal starts writing into ERT. Thus, at the end of traversal on space A, instead of sending $ERT_{A|B}$ to a node B, a *Remove* message consisting of $\{ERT_{A|B}$[previous] - $ERT_{A|B}$[present]$\}$ is sent(as in the SGP protocol, corresponding ODT_B entries are removed). Additionally, an *Add* message consisting of $\{ERT_{A|B}$[present] - $ERT_{A|B}$[previous]$\}$ is also sent. *Add* messages are used to propagate root information. For example, if a and b belong to spaces A and B respectively, and a reference $a \rightarrow b$ is created, $< b, \phi, \phi, a, B >$ is added to ERT_A and $< b, \phi, \phi, a, A >$ is added to ODT_B. When local GC completes on space A, it contains the entry $< b, a, \phi, a, B >$ instead of $< b, \phi, \phi, a, B >$. This results in the *Remove* message containing $< b, \phi, \phi, a, B >$ and an *Add* message containing $< b, a, \phi, a, B >$. Space B on receiving these messages removes the old ODT entry and replaces it with $< b, a, \phi, a, A >$. Similarly, probes are installed and propagated using *Add* messages. We refer the reader to [KA93] for a detailed explanation of the algorithm, its correctness proof and a detailed example.

Conclusion: We have presented a protocol that uses the same techniques used for inter-space GC in [SGP90] to handle inter-space cycles. No distributed termination protocol is used and objects are not migrated to detect cyclic garbage. Though entries in the tables are larger, the number of entries is itself very small since applications generally exploit

On node i: (At the end of local GC)

for all spaces[a] j
{ Calculate $Remove_i$[j] = { $E \mid E = < O, O_r, O_p, O_i, j >$ and
$E \in (ERT_i$[previous] $- ERT_i$[present]) and $O \in$ space j } ;
/* Until here, the algorithm is the **same** as presented before */
Calculate Add_i[j] = { $E \mid E = < O, O_r, O_p, O_i, j >$ and
$E \in (ERT_i$[present] $- ERT_i$[previous]) and $O \in$ space j } ;
Send $Remove_i[j]$(**message** *Remove*) and
$Add_i[j]$(**message** *Add*) to node j;
}

/* *Remove* messages are handled similarly as in [SGP90](i.e
corresponding ODT_j entries are removed)*/

On node j: (when an *Add* message is received from node i)

for every $< O_j, O_r, O_p, O_i, j > \in Add_i[j]$(j is the node servicing the message)
{ if $(O_p = O_j)$ then /*this is a probe sent by O_j */
Delete corresponding entry $< O_j, O_r, O_p, O_i, i >$ from ODT_j;
else if $< O_j, O_r, D^b, O, D > \in ODT_j$ then/*duplicate suspected*/
Add $< O_j, O_r, O_p, O_i, i >$ to ODT_j;/**probe* installed*/
else if $O_r \neq \phi$ then /* no root information in ODT entry */
Replace $< O_j, \phi, \phi, O_i, D >$ by $< O_j, O_r, \phi, O_i, D >$;
/*add *root* information*/
}

[a] Of course, if no object belonging to a space is referred to from space i, no messages are sent to that space.

[b] D stands for a "don't care" value; basically a value not important to decision making.

Figure 2: Sub-protocol that collects cyclic garbage

communication locality and so there are very few *public*[2] objects[SGP90]. Thus, scalability of the SGP protocol is preserved.

References

[Bis77] P. B. Bishop. Computer systems with a very large address space and garbage collection. *Tech. Rep. MIT/LCS/TR-178, M.I.T. Laboratory of Computer Science*, Cambridge, MA, May, 1977.

[2] An object accessible from at least one remote space is called *public*, as opposed to *private* objects/citePS92.

[Hug85] J. Hughes. A distributed garbage collection Algorithm. In *Functional Languages and Computer Architechtures.* In Lecture Notes in Computer Science, Sep 1985.

[JJ92] N. C. Juul and E. Jul. Comprehensive and Robust Garbage Collection in a Distributed System. IWMM'92.

[KA93] R. Kordale, and M. Ahamad. A Scalable Cyclic Garbage Detection Algorithm for Distributed Systems. *Georgia Inst. of Technology, Tech. Report*(In progress).

[LL92] R. Ladin and B. Liskov. Garbage Collection of a Distributed Heap. *In proceedings of 12th International Conference on DCS, 1992.*

[LQP92] B. Lang, C. Queinnec and J. Piquer. Garbage collecting the world. In Proc. of the 19th Symp. on Principles of Programming Lang., 1992.

[MS91] L. Mancini and S.K. Shrivastava. Fault-tolerant reference counting for garbage collection in distributed systems. The Computer Journal 34,6. Dec 1991.

[Piq91] J. M. Piquer. Indirect reference counting, a distributed garbage colection algorithm. In *PARLE'91*, vol. I of *Lecture Notes in Computer Science*, Springer-Verlag.

[PS92] D. Plainfosse and M. Shapiro. Experience with a Fault-Tolerant Garbage Collector in a Distributed Lisp System. IWMM'92.

[SGP90] M. Shapiro, O. Gruber and D. Plainfosse. A Garbage Detection Protocol for a Realistic Distributed Object Support System. *Rapport de Recherche 1320*, INRIA, France, November 1990

[Ves87] S. C. Vestal. Garbage Collection: An Exercise in Distributed, Fault–tolerant Programming. *PhD thesis, Dept. of Comp. Sc., U. of Washington, Seattle.* Tech. Report 87–01–03.