Cyclic Distributed Garbage Collection Without Global Synchronization in CORBA

Gustavo Rodriguez-Rivera (grr@cs.purdue.edu),
Vince Russo (russo@cs.purdue.edu)
Computer Science Department
Purdue University
West Lafayette, IN 47907

1 Abstract

This paper describes an algorithm for cyclic distributed garbage collection and its implementation. The algorithm is an extension of reference-listing that collects cyclic garbage. It uses back-tracing instead of mark-and-sweep to eliminate the need of global synchronization. First, by using a special heuristic the algorithm chooses an object that is likely to be garbage (suspect). Then the algorithm recursively traces back the references to find the transitive closure of objects that reach the suspect. If this closure does not contain any root, the objects in the closure are garbage, and therefore collected.

Back-tracing distributed garbage collection has been covered in previous works, however none of these works mentioned the issues involved in the implementation of this algorithm in a real environment. These implementation issues are addressed and solved in this paper.

To prove the feasibility of back-tracing, it has been implemented using off the shelf software components.

This paper describes the algorithm, the issues, their solutions and an implementation.

2 Introduction

The way networking applications are implemented has evolved since the creation of the Internet. The growth in complexity of these applications has motivated the creation of better networking abstractions to deal with this complexity. This evolution is similar to the evolution of computer languages. Garbage collection is now a required feature in modern computer languages. For the same reasons, distributed garbage collection is also a necessity in modern distributed systems.

Distributed garbage collection algorithms can be grouped in two families: Reference Counting and Tracing[Jon96, PS95].

Reference counting is a garbage collection technique that is rarely used in non-distributed systems because of its high overhead and its incapability to collect cyclic garbage. However, it is a popular technique in distributed garbage collection because it does not require global synchronization. Reference Counting keeps a count of the number of references that point to an object. When this counter drops to zero, the object can be safely removed. Reference counting keeps a relaxed approximation of the reachability graph, i.e. the set of objects that have a reference counter greater than zero is a superset of the true reachable objects. This relaxed approximation eliminates the need of a termination algorithm to collect garbage, but has the disadvantage that cyclic garbage is not collected.

Reference listing is a variation of reference counting that is tolerant to failures [SDP92a, BEN$^+$93, MS91]. In reference listing the server that owns an exported object keeps a list of other servers that have references to that object. In this way, increment/decrement operations are substituted by insert/delete operations.

Since a list of servers is kept instead of a single counter, duplicated messages do not affect the consistency of the reference listing. For example, if an insert message is duplicated, the first message will update the reference list, but the second one will not have any effect since that entry already exists. The same is also valid for delete messages.

There are some implementations that combine reference listing with local garbage collection [BEN$^+$93]. During a local garbage collection, every exported object is considered part of the root set. When a proxy is found to be unreachable from the root set, a delete message is sent to the corresponding remote object to delete that server from the reference list. When the remote object's reference list is empty, the remote object is no longer considered part of the root set, and furthermore collected in the next local garbage collection if it is not locally reachable.

Reference listing is also tolerant of server crashes. If a server detects that another server has crashed, it may remove the entries in the reference lists that correspond to the crashed server. To detect that a server has crashed, the owner of the remote object periodically sends a special message called *ping* to the servers that have proxies to that remote object. If a server does not respond to the *ping* messages after a prespecified number of them, the server is deleted from the reference list [BEN$^+$93].

Reference listing scales up well to large networks. However it is not capable of collecting garbage cycles. Currently reference listing has been implemented in commercial products like Microsoft DCOM [Mic94], and Java RMI [GM95].

Tracing distributed garbage collection is based on mark-and-sweep garbage collection. Tracing distributed garbage collection is able to collect cycles of garbage, however it requires global coordination. The reason is the following: mark-and-sweep defines as garbage everything that is left unmarked after a mark phase. This implies that after a mark-phase, *all the servers in the system* have to agree on the same condition: that there are no more objects to be marked. This problem is equivalent to the costly termination detection problem in a distributed system [TM93]. Examples of distributed garbage collection algorithms that implement mark-and-sweep garbage collection are [JJ92, MA84].

In one hand, reference counting algorithms can be efficient and scalable, but are unable to collect cyclic garbage. In the other hand, tracing distributed garbage collection algorithms collect cyclic garbage but require global synchronization. This paper describes an algorithm that collects cyclic garbage and that does not need global synchronization.

This paper proposes an extension of reference-listing called *back-tracing*, that is scalable and collects cyclic garbage. The idea is the following: starting with an exported object that is suspected of being garbage (e.g. an exported object that is not locally reachable), the collector adds to a set all the objects that contain pointers to the suspect. Then all the objects that contain pointers to any object in the set are also added and so on. If the final set does not contain a root object, then all the objects in the set are garbage. Otherwise, the suspect is not garbage. The process of obtaining this set is called *back-tracing* a suspect. The back-tracing algorithm has the desirable property that back-tracings are localized. For instance, in mark-and-sweep (i.e. forward-tracing), *all* the servers that participate in a garbage collection need to synchronize to agree when the mark phase has ended to be able to sweep the unmarked objects. However, in back-tracing only a subset of the total number of servers, that are the servers visited during the back-tracing of a single suspect will need synchronization. It is shown in this paper that the synchronization needed for back-tracing does not require the servers to stop execution.

Some previous work has been done on back-tracing [Fuc95]. However several important implementation issues were left open: These issues have to be solved before back-tracing can be implemented. First, normal pointers can be followed in only one direction, the *to* direction, and back-tracing needs to follow pointers backwards, the *from* direction. These kind of pointers that can be followed backwards are not usually available in normal programs. Second, the objects and pointers traced during back-tracing can be modified while the back-tracing goes on, and therefore live objects could be prematurely collected. Third, a single object can be visited several times by different back-traces, making it complicated to store the state of the back-trace in the objects visited. Finally, no implementation was available to prove that this algorithm could be successfully implemented with off the shelf software components.

This paper addresses and solves these issues. It also shows how a prototype of the back-tracing algorithm was built using the Orbix distributed system[Ion95], and Boehm's conservative garbage collector [BW88]. Finally, the paper shows directions of future work.

3 The Back-Tracing Algorithm

By definition, live objects are the objects that are reachable from root objects by following a chain of object references. Alternatively a live object is one that when following backwards the chain of references that reach the live object, a root object is found. This first definition is used by mark-and-sweep garbage collection to detect garbage objects. The second one is used by back-tracing garbage collection.

Back-tracing is a trial-an-error algorithm. First, an object that one suspects to be garbage is chosen. Then object references that point to this suspect are followed backwards. References to objects found during the back trace are also followed back until they are exhausted. If a root object is reached during the back-trace, then the suspect is still alive. Otherwise, if no root is found, then the suspect and all the objects back-traced are garbage.

The algorithm sounds simple and easy to implement. However, there are several issues that need to be solved before it can be successfully implemented.

First, the algorithm needs a good heuristic to determine which objects to back-trace. If a non-garbage object is selected for back-tracing, then all the work done during the back-tracing will be wasted. The efficiency of the algorithm therefore depends in great part on how garbage suspects are selected.

Second, not all the back-references necessary for back-tracing are available in a program. Back-references can be divided in two classes: local back-references, and remote back-references. Remote back-references can be provided by a reference-listing algorithm. Since back-tracing is an extension of reference listing, every exported object keeps a list of the servers that have proxies to the exported object. Reference-lists are in fact remote back-references since they point from exported objects back to proxies in other servers. Within the address space of a program however, local back-references do not exist. Pointers in memory have only information about the *to* direction, but not the *from* direction. One approach to implement back-pointers is to modify the language run-time system to keep for every object a list of the locations of the pointers that point to the object. This approach is extremely inefficient, since every pointer operation has to update also the list of back-pointers. An alternative approach is to use the local garbage collector to obtain back-pointer information. This is the approach described in this paper.

Since there is no global coordination, servers may run back-tracings for different garbage suspects that share the same ancestors. The data structures used to represent back-tracings should consider these cases.

Finally, the modification of objects during a back-trace can cause the premature collection of live objects. This is equivalent to the problem of incremental garbage collection. If an object has been back-traced and then a new back-reference to it is created, the new back-reference will not be considered by the back-trace in progress, and therefore the object could be collected prematurely.

The solution to these four issues is described in the next sections.

3.1 Determination of Garbage Suspects

The efficiency of back-tracing garbage collection depends in great part on the heuristic to choose garbage suspects. Back-tracing a live object is a waste of execution time and network bandwidth. Therefore garbage suspects should be chosen carefully. This section describes a simple heuristic. A following section describes an improved heuristic based on object's age.

The simple heuristic chooses as a garbage suspect any exported object that is not locally reachable. These are characteristics that any useless exported object have. This heuristic can be implemented with a flag called *locally-reachable* that is initialized to *True* when the object is first exported. This flag is set to *False* during a local garbage collection if after tracing local roots but before tracing exported objects, the object was not marked and therefore not locally reachable.

But that is not all. An exported object that is not locally reachable can still be accessed remotely. Furthermore, it can become locally reachable again as a result of a remote method invocation. To take into account this cases, every time an exported object *o* is used in a remote method invocation, either as argument or as a target, *o.locally-reachable* is set to *True*.

A garbage suspect is therefore, any exported object with a flag *locally-reachable* equal to *False*. If distributed garbage objects exist, the algorithm makes sure that there are always suspects available by running the local garbage collector periodically.

3.2 Local and Remote Back-References

The second issue is how local and remote back-references necessary for back-tracing are obtained. Back-tracing is an extension of reference listing, and therefore remote back-references are already available in the form of reference-lists. Every exported object o has a list *o.ref-list* of servers that have proxies to o. However, local back-references do not exist and therefore have to be computed in some way. This section describes how the local garbage collector computes local back-references.

It is important to say that not all the local back-references are necessary for back-tracing. Exported objects and proxies are the only ones relevant for distributed garbage collection. Objects that are not exported can be automatically collected by the local garbage collector once they are not reachable from either local roots or exported objects. It is possible therefore, to obtain an approximation of all the back-references hiding all non exported objects, and only representing how exported objects and proxies reach each other.

Every proxy or exported object x stores its back-references both local and remote, in a list *x.back-refs*. The list *x.back-refs* contains pairs of the form *(object, server)*, where a single pair represents either a proxy or an exported object that points to x. If *object* resides in *server*, then the pair represents an exported object. Otherwise, the pair represents a proxy.

To determine *x.back-refs*, the local garbage collector is modified in the following way. First, after tracing all the local roots but before tracing the exported objects, the local garbage collector saves the mark-bits in a temporal area. Then *x.back-refs* is initialized to empty in all proxies and exported objects.

Second, for every exported object o that is not locally reachable, the local garbage collector clears all the mark-bits and traces o. If a proxy or exported object x is marked after tracing o, then o reaches x and therefore the pair *(o, this-server)* is added to *x.back-refs*.

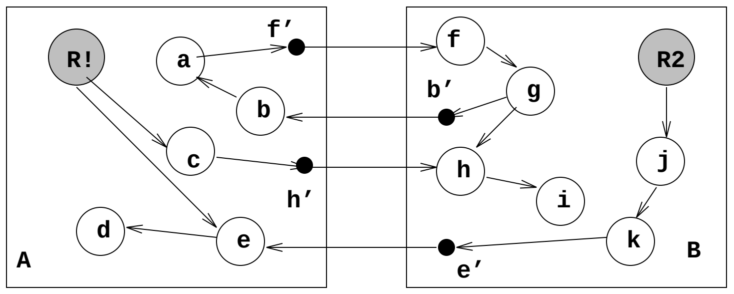
So far only local back-references have been taken into account. To account for the remote back-references, for every exported object o that is not locally reachable, and for every server s in *o.ref-list*, the pair *(o, s)* is added to *o.back-refs*. If an exported object or proxy x is locally reachable, the special pair *(local-root, this-server)* is added to *x.back-refs*.

Finally, the mark-bits are restored from the temporal area, and all the non locally reachable exported objects are traced again to prevent the premature collection of live objects that are only reachable from non locally reachable exported objects. Objects that are left unmarked are recycled during the sweep phase.

An example of how back references are obtained by the local garbage collector is shown in Figure 1.

3.3 Performing Back-Tracing

This section describes how back-traces are performed and the data structures used to support simultaneous back-traces. Given the asynchronous nature of back-tracing, it is possible that two garbage suspects that have the same ancestors may trigger back-traces that trace the same ancestor simultaneously. Therefore the data structures used for back-tracing should support simultaneous back-traces that consider the same objects. Also, it is desirable to have all the state of a back-trace in the same server that triggered the back-trace to make it easier to halt when a



```
f'.back-refs = {(b, A)}     f.back-refs =  {(f', A)}
h'.back-refs = {(R1,A)}     b'.back-refs = {(f, B)}
b.back-refs =  {(b', B)}    e'.back-refs = {(R2,B)}
e.back-refs =  {(R1,A)}     h.back-refs =  {(f,B), (h',A)}
```

Figure 1: Back-references computed by local GC

root object is found. The following algorithm and data structures satisfy these restrictions.

A single back-trace is represented by a queue Q of pairs *(object, server)*, where each pair represents, as before, either a proxy or a exported object that participates in the back-trace. Q also has two indices: *current* and *tail*. *Current* points to the next element to back-trace in Q. *Tail* points to one after the last element inserted in Q.

Initially Q is empty. When a garbage suspect *susp* triggers a back-trace, the pair *(susp, this-server)* is inserted into Q. *Current* is initialized to 0, and *tail* to 1. Then all the back-references of *susp* in *susp.back-refs* are inserted to Q and *current* is incremented. Then the next proxy or exported object *(x, server)* in Q pointed by *current* is back-traced by inserting to Q all the back-references in *x.back-refs*. Then *current* is incremented, and so on. Only unrepeated elements are inserted in Q and therefore the back-trace is guaranteed to end at some point in time (Figure 2 and Figure 3)

The back-trace terminates when: (a) An object in Q is a local-root, in which case the suspect is also reachable, and therefore the back-trace is halted; or (b) *current* and *tail* have the same index, in which case all the proxies and exported objects that reach the suspect have been back-traced and exhausted, and therefore none of them is reachable from any local root. Furthermore, the proxies and exported objects in Q can be recycled.

Q is maintained in the same server that starts the back-trace. Several back-traces can be performed simultaneously in several servers by keeping different Qs in each server.

To obtain the back-references of a remote object or proxy *(o, server)* during a back-trace, a special procedure *getBackRefs(o)* is provided in each server. When this procedure is called, either locally or remotely, it returns the list *o.back-refs*.

A back-trace that ends without reaching a local root means that the suspect and all the proxies and exported objects in Q are not reachable from any local root. If that is the case, the servers that own these proxies and exported objects are informed. These exported objects and proxies will not be traced in the next local garbage collection and hence collected.

A back-trace that halts because some exported object or proxy in Q is reachable from a local

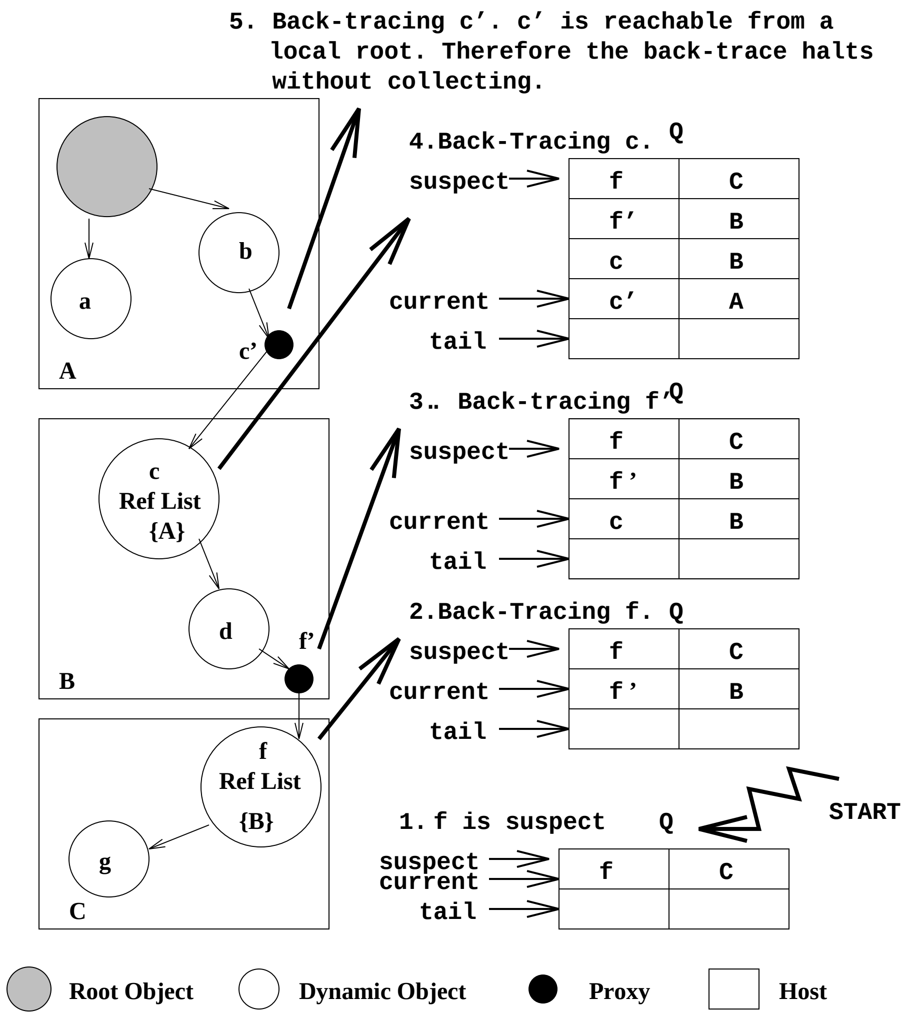


Figure 2: Back-tracing a reachable suspect

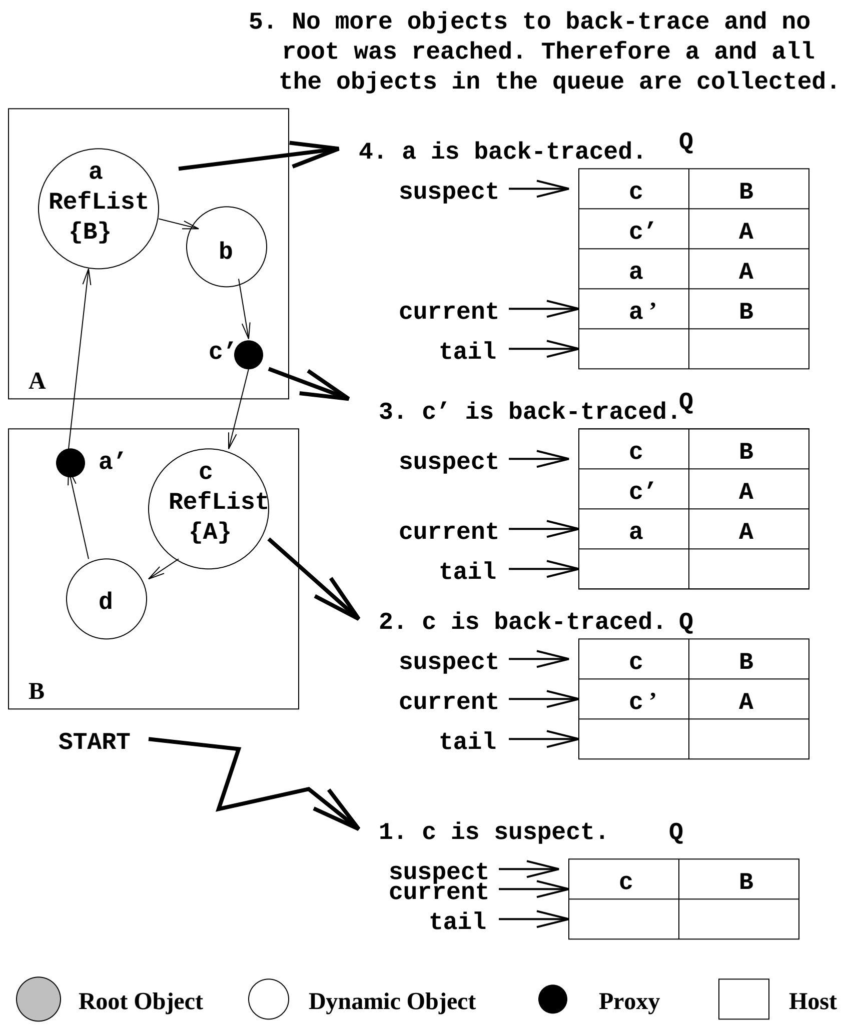


Figure 3: Back-tracing an unreachable suspect

root means that the suspect is also reachable from a root object. If that is the case, no action is taken and Q is discarded.

3.4 Incremental Back-Tracing

There is a problem in back-tracing that is similar to the problems of incremental garbage collection. Objects and back-references that participate in a back-trace can be modified while a back-trace is taking place and therefore, the risk of prematurely collecting live objects.

For instance, lets assume that some exported object has been back-traced and this exported object and all its proxies have been already inserted in Q. While the back-trace is going on, a new proxy for this exported object is created. Since the exported object has been already back-traced, this new proxy is not inserted in Q, and therefore not back-traced. Assuming that this new proxy is reachable from a root, but no other object in the back-trace was found reachable from a root, then the exported-object will be prematurely collected.

This problem can be described in terms of Dijkstra's tricolor marking used for incremental garbage collection [DLM+78], and now adapted for back-tracing. Exported objects and proxies in Q that have been already back-traced are *black*. Exported objects and proxies in Q that have not been back-traced are *grey*. All the objects that are not in Q are *white*.

Alternatively, exported objects and proxies that have an index in Q less that *current* are *black*. Exported objects and proxies that have an index in Q greater or equal than *current* but less than *tail* are *grey*. All the other objects are *white*.

Using this tricolor marking it is possible to explain the problem of premature collection in the following way. An exported object or proxy is missed during a back-trace if both the following two conditions occur during the back-trace.

1. A new reference is created that goes from a *white* to a *black* exported object or proxy.
2. All the paths from root objects to *grey* exported objects or proxies are deleted. The *white* exported object or proxy is left reachable from a root object.

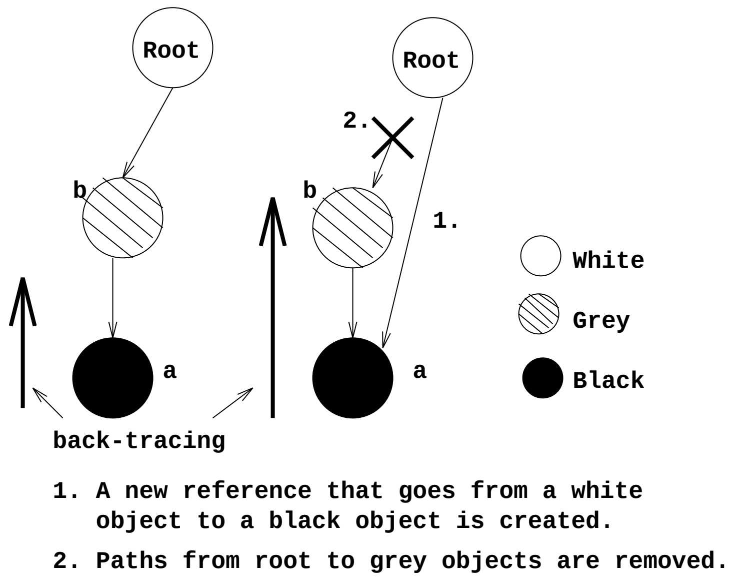
A situation when the two conditions occur is shown in Figure 4.

In order to avoid the premature collection of live objects either one of these conditions has to be prevented. To prevent the first condition it is necessary to catch with a barrier every operation that creates a new reference for a *black* object or proxy and either turn the *black* object to *grey* by removing it from Q and inserting it again, or by turning the *white* object to *grey* by inserting it to Q.

Preventing the second condition means to always keep a path from the root objects to the *grey* objects if such path existed when the back-trace started.

The algorithm described in this paper detects the first condition. However, instead of trying to fix the colors of the *white* and *black* objects and proceed with the back-trace, the algorithm just halts the back-trace. That is because it is not worth resuming the back-trace since the *black* object was already reachable from a root, and therefore very likely to continue being reachable when the back-trace ends.

The barrier that detects when a new reference is created for a *black* object is implemented in the following way. Because of the way the local garbage collector is implemented, a *black*



Root
Root
2.
b
b
1.
White
Grey
Black
a
a
back-tracing
1. A new reference that goes from a white object to a black object is created.
2. Paths from root to grey objects are removed.

Figure 4: Two conditions for premature collection.

exported object or proxy x in Q is not reachable from a local root when its back-references in Q were computed. Therefore, there are only two ways a reference can be created for x and therefore be missed: (a) Passing x back to the server where x lives as argument or result of a remote method invocation. or (b) Invoking a remote method in an exported object that locally reaches x.

To detect this two cases, a new field called *version* is added to every exported object o. *o.version* is increased every time o is used in a remote method invocation either as a argument, result, or target. By looking at the changes in *o.version* it is possible to determine if o or any of the objects that o reaches locally, may have a new reference.

The back-references stored in *x.back-refs* are now represented by the triplet *(object, server, version)*. As before, this triplet can represent either a proxy or a exported object. When back-references are computed by the local garbage collector, if an exported object o is found to locally reach x, then the triplet *(o, this-server, o.version)* is inserted in *x.back-refs*. When adding the remote back-references to an exported object o, for every *server* in *o.ref-list*, a triplet *(o, server, o.version)* is inserted in *x.back-refs*. This triplet *(object, server, version)* is also used for the entries in Q.

The version number in a back-reference indicates what *version* of the object reaches a given proxy or exported object. The main idea behind incremental back-tracing is that a *black* object or proxy x in Q can have a new back-reference that is not in Q only if the *version* number of any of the triplets that were inserted in Q when x was back-traced is different than the current *version* number in the original object. Therefore, if at the end of a back-trace, there is a triplet *(o, server, version)* in Q such that *version* and *o.version* are different, then the back-trace has to be halted because there may be a new reference to a *black* object that was not considered in the back-trace.

Incremental back-tracing has two phases: a back-trace phase and a validation phase. The first phase back-traces the objects to determine if the suspect is garbage. The second phase validates that no changes occurred in the back-traced objects while the back-trace was taking place. There is a special call *getVersion(o)* in any server that returns the value *o.version* when called remotely or locally. During the second phase, for every exported object *(o, server, version)* in *Q version* is compared with the result of the function *getVersion(o)* executed in *server*. If these numbers are different in any of the triplets, the back-trace is halted. Otherwise, the servers that own entries in Q are told to collect these proxies and exported objects in the next local garbage collection.

Some locking is necessary to make the computation and access of the back-references atomic. A mutex lock is used when the local garbage collector computes the back-references, when new remote back-references are created, and in the function *getBackRefs()*.

3.5 Generational Back-Tracing

Previous sections have explained that choosing the right garbage suspects can increase the performance of back-tracing. This section explain a heuristic that takes into account the age of the objects.

In generational garbage collection [Wil95, LH83, Ung84, Sha88, Zor90, DeT90, Hay91], new objects are collected more frequently than older ones because they are the most likely to be garbage. By using the same idea, generational back-tracing performs back-tracing of recently allocated objects more frequently than old objects.

The back-tracing algorithm described before performs back-tracing of every single non locally reachable object every T seconds. In the new heuristic, the more time an object has survived collections, the less frequent it is back-traced. The frequency function that is used in the implementation is shown in Figure 5. This figure shows that if an object has survived for 5 minutes, then it is back-traced every 12 minutes. If an object has survived 10 minutes, then it is back-traced every 22 minutes and so on. The maximum interval between back-traces for any object in this case is every 60 minutes. The smallest interval is 1 minute. The parameters of the exponential function shown here are the ones used in the implementation but they are configurable.

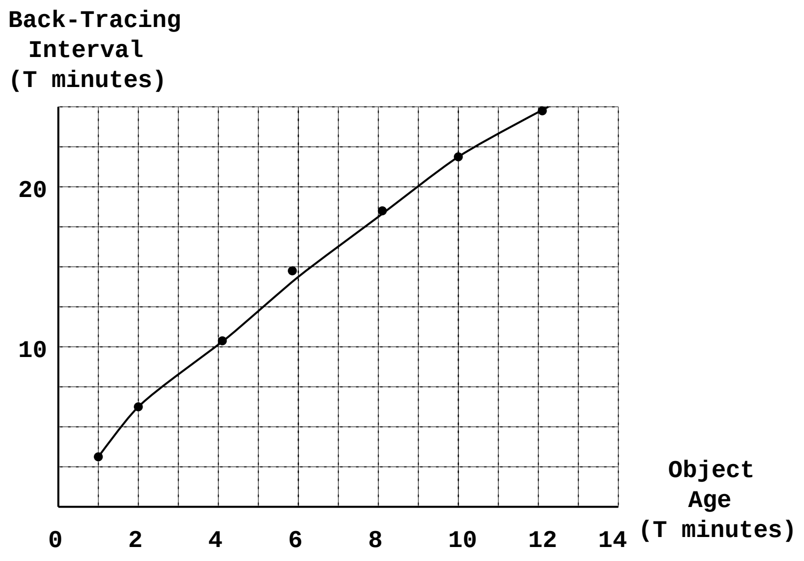
This new heuristic benefits the common case when an object is created and then kept alive during the entire execution of the program. The old heuristic back-traces the object every 1 minute. With generational back-tracing, the object is back-traced every 60 minutes in the steady state.

3.6 Back-Trace Factoring

Back-Trace Factoring is another optimization that improves the efficiency of back-tracing distributed garbage collection by reducing the number of redundant back-traces.

The last section showed that if a suspect is found reachable after a back-trace, the suspect's back-trace interval is increased. However, the back-trace interval of other objects involved in back-trace is not affected even when they are also found reachable. Hence, these objects may trigger redundant back-traces almost immediately after they have been used in a back-trace.

For example, assume a list of 4 objects a, b, c, and d, where a is locally reachable. Assume also that d is chosen suspect. Then c, b, and a participate in the back-trace. Since a is locally reachable the back-trace halts, d is determined reachable, and its back-trace interval incremented.



$$\text{back-tracing interval} = 60\left(1 - e^{\text{age}\frac{\log(.5)}{15}}\right)$$

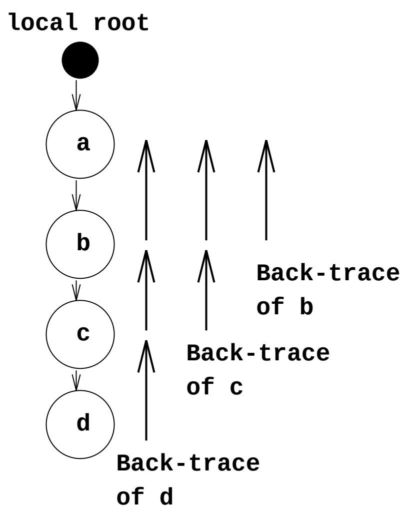
Figure 5: Back-Tracing Interval vs. Object Age.

However, the back-trace intervals of a, b, and c are not modified. Hence, b and c are likely to be chosen suspects and trigger back-traces after a short period of time, even when they were recently found reachable.

In general, a list of n elements where the first element is reachable from a local root triggers $n-1$ back-traces before all its elements have their back-trace interval incremented. In each back-trace $n/2$ objects in average are visited. Therefore the total number of object visits is $(n-1)n/2 = O(n^2)$ (Figure 6).

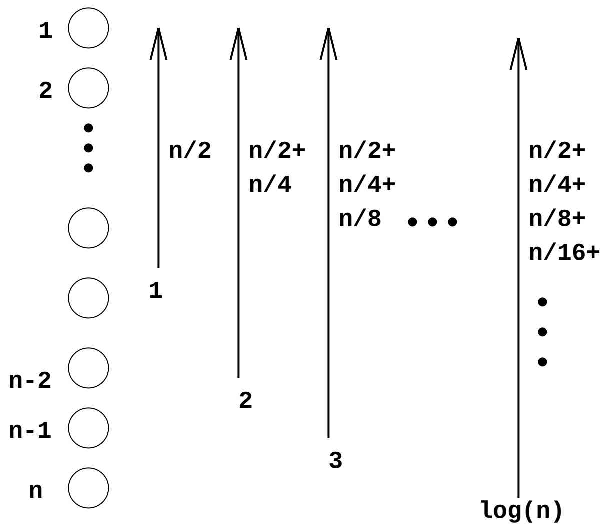
To reduce the number of object visits, every time an object is visited during a back-trace, the back-trace interval of the object visited is also incremented. In this way, if the suspect is found reachable, all the objects involved in the back-trace will have the back-trace interval also incremented. This mechanism is called *back-trace factoring*

By using this mechanism, the number of back-traces performed before all the objects in the list of n elements have their back-trace interval incremented is computed as follows. Assuming that suspects are chosen randomly in the list, the expected number of objects visited in the first back-trace is $n/2$ (Figure 7). Since the back-trace interval is incremented in the objects visited, they will not trigger back-traces immediately afterwards. For the second back-trace, one of the remaining $n/2$ objects in the list is chosen suspect. In this back-trace, the expected number of objects that have their back-trace interval incremented and that had not been incremented before is $n/4$ (Figure 7). Following the same reasoning, the expected number of back-traces is $log(n)$, and the number of object visits is $n/2 + (n/2 + n/4) + (n/2 + n/4 + n/8)... = O(nlog(n))$.



The total objects visited before all the back-trace intervals of a, b, c, and d are incremented is (n-1)n/2 = 3*4/2 = 6. No back-trace factoring is used.

Figure 6: Redundant back-traces.



The expected total objects visited before the back-trace intervals of the n objects are incremented is O(n * log(n)). Back-trace factoring is used.

Figure 7: Back-trace factoring.

4 Implementation

The back-tracing algorithm described in this paper has been implemented using off the shelf software components: the Solaris operating system. Boehm's conservative garbage collector [BDS91], Orbix [Ion95], Solaris, and C++.

The distributed garbage collector (DGC) is implemented as a library that can be linked with any Orbix program. The DGC automatically collects Orbix objects when they are no longer reachable from any local or remote root. The library implements in Orbix a special exported object called the distributed garbage collector object (DGC object). Every server that participates in the distributed garbage collection needs to have a DGC object. Servers that are not linked with the DGC library still can import collectible objects. However, the roots of servers without DGC's are not considered during distributed garbage collections. In the other side, servers linked with the DGC library can import/export both collectible and non collectible objects. This gives backwards compatibility to existing Orbix programs.

The local garbage collector is based on Boehm's conservative garbage collection library. Some extra functionality was added to the original collector to trace Orbix's roots in a special manner and to compute the back-references.

In Orbix theres is an Object Table (OT) that contains pointers to all the proxies and exported objects that exist in the server. Since the local collector considers the OT and all the Orbix data structures part of the root set, an unmodified local collector would always mark all the exported objects and proxies. As a result, exported objects and proxies would be always locally reachable and therefore never collected. To solve this problem, the local garbage collector traces Orbix's data structures in a special to eliminate the unnecessary retention of exported objects and proxies.

When Orbix data structures are traced, only objects that are neither proxies nor exported objects are pushed to the mark stack. After finding out which exported objects are locally reachable and which ones are not, the local garbage collector traces all the exported objects and proxies in the system to prevent the premature collection of other objects reachable from exported objects and proxies.

This solution is not ideal. The ideal solution would be to modify the data structures in Orbix and to use *weak* pointers instead of normal pointers when referencing exported objects and proxies. Weak pointers are a special kind of pointers that do not keep alive the objects they point to. However, since Orbix's sources are not available, this solution is not feasible in the current implementation.

The local garbage collector was modified to obtain the local back-references using auxiliary tracing in the way described in the last section. After tracing all roots but before tracing exported objects, each non locally reachable exported object is traced independently. If a proxy or exported object x is reachable from exported object o , then o is added to the list of back-references of x.

There are two main issues in the interfacing of the distributed garbage collector and Orbix. First, collectible and non-collectible objects have to be differentiated, so the distributed garbage collector can process collectible objects in a special manner both in the client and in the server side. And second, since no Orbix's source code is available, the distributed garbage has to intercept and reimplement the creation/deletion operations on object references.

The first issue comes from the fact that there may be some servers without distributed garbage collection in the system, i.e. some of the exported objects and proxies may not be garbage collectible. Furthermore, it is necessary to differentiate garbage collectible objects and non garbage collectible objects.

In order to keep the garbage collectible property orthogonal to object type, the collectible property is linked to object names. When a collectible object is created, a special suffix is added at the end of the object's name by using the function *setGarbageCollectible(o)*. Only exported objects and proxies that have this special suffix are handled by the distributed garbage collector. All the other objects and proxies that do not have this special suffix are ignored by the distributed garbage collector.

The second issue is solved by intercepting and reimplementing some Orbix function calls. In the same way that Boehm's collector can intercept *malloc/free* and *new/delete* operations to control memory allocation, the distributed garbage collector intercepts the *duplicate/release* reference operations defined in Orbix to control remote object allocation. These operations are always called when marhsaling and unmarshaling object references. If the marker of the object passed as argument to these functions corresponds to a garbage collectible object, then the distributed garbage collector handles the object in a special way. Otherwise, it just calls the corresponding *duplicate/release* operation defined in Orbix. Some help from the dynamic linker is necessary to have access to both DGC's and Orbix's implementations of the *duplicate* and *release* functions.

Orbix does not implement reference listing. Instead, Orbix uses a mix of explicit deallocation and local reference counting to keep track of local pointers to exported objects and proxies. In Orbix, every proxy and exported object has a reference count. The programmer has to explicitly call *duplicate* or *release* when duplicating or removing pointers to proxies or exported objects. If the reference count goes to 0, the exported object or proxy is removed. The reference counter is only kept locally and therefore does not influence any other reference counters of corresponding exported objects or proxies in other servers. This limited functionality of reference counting is not enough for the requirements of the back-tracing garbage collection algorithm. Therefore, it is necessary to implement reference listing on top of Orbix.

Reference listing is implemented on top of Orbix by inserting special code in the *duplicate(o)* and *release(o)* operations. These operations are called every time an object reference is marshaled or unmarshaled. For every exported object the distributed garbage collector maintains a list of servers that have a proxy to the exported object. When a new proxy is created, the server that owns the exported object is informed, and then it adds the proxy's server to the reference list.

The distributed garbage collector is a library that can be linked to any Orbix program. The distributed garbage collector is itself an exported object implemented in Orbix. Every server that participates in the distributed garbage collection exports a similar DGC object. DGC objects in different servers communicate with each other through Orbix. Currently the DGC object is implemented in Solaris and C++. However nothing prevents the implementation of DGC objects in other languages such as Java and Visual Basic, and other operating systems such as Windows NT and other Unix flavors, to furthermore create heterogeneous distributed garbage collection.

5 Directions for Future Work

There are still several open issues in back-tracing garbage collection. Here it is a list of just a few of them.

More work has to be done to reduce the number of remote method calls needed for back-tracing. It would be possible to reduce the number of remote method calls by using a better heuristic for choosing garbage suspects. One possibility is to combine Hughes algorithm [Hug85] with back-tracing and perform back-tracing only in those objects that have a time-stamp older than some prespecified value.

Also the distributed garbage collection implementation shown here is homogeneous in the sense that only Orbix applications in Solaris can use it. However nothing prevents other servers that use other operating systems or languages to use it as long as they also implement a distributed garbage collector object with the same IDL interface and characteristics. In this way it is possible to have heterogenous distributed garbage collection. A good candidate for implementing distributed garbage collection is Java. The current Remote Method Invocation (RMI) in Java uses reference listing for distributed garbage collection, andtherefore is unable to collect cyclic garbage.

Another future direction is to analyze the performance of back-tracing using real applications. Unfortunately, since cyclic distributed garbage collection is not currently available, there are no too many applications that make full use of distributed garbage collection. Hopefully the number of these applications will start growing once distributed garbage collection is widely spread.

Finally, it would be worth exploring the possibility of using back-tracing for garbage collecting persistent stores.

References

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, Ontario, June 1991. ACM Press.

[BEN+93] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed Garbage Collection for Network Objects. Technical Report 116, Digital Systems Research Center, 1993.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[DeT90] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, Palo Alto, California, August 1990.

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

[Fuc95] Matthew Fuchs. Garbage collection on an open network. In Henry Baker, editor, *International Workshop on Memory Management*, Lecture Notes in Computer Science, Concurrent Engineering Research Center, West Virginia University, Mor gantown, WV, September 1995. Springer-Verlag.

[GM95] James Gosling and Henry McGilton. The JAVA Language Environment: A White Paper. Available from http://www.javasoft.com/whitePaper, 1995.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In Andreas Paepcke, editor, *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[Hug85] 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.

[Ion95] Iona Technologies, Dublin, Ireland. *Orbix Programmer's Guide*, July 1995.

[JJ92] 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.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, New York, 1996. With a chapter on Distributed Garbage Collection by R. Lins.

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

[MA84] Khayri A. Mohamed-Ali. *Object Oriented Storage Management and Garbage Collection in Distributed Processing Systems.* PhD thesis, Royal Institute of Technology, Stockholm, December 1984.

[Mic94] Microsoft Corporation. *OLE2 Programmer's Reference*, volume 2. Microsoft Press, 1994.

[MS91] Luigi Mancini and S. K. Shrinivastava. Fault-tolerant reference counting for garbage collection in distributed systems. *Computer Journal*, 34(6):503–513, December 1991.

[PS95] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), September 1995.

[SDP92a] 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 [SDP92b]: corrects a bug, more elegant, more informative.

[SDP92b] 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), nov 1992. Also available as Broadcast Technical Report #1.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Palo Alto, California, February 1988. Technical Report CSL-TR-88-351, Stanford University Computer Systems Laboratory.

[TM93] G. Tel and F. Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. ACM Press, April 1984. Published as *ACM SIGPLAN Notices 19*(5), May, 1987.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[Wil95] Paul R. Wilson. Garbage collection. *Computing Surveys*, 1995. Expanded version of [Wil92]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps`. In revision, to appear.

[Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Conference Record of the 1990 ACM Symposium on LISP and Functional Programming*, pages 87–98, Nice, France, June 1990. ACM Press.