Complete distributed garbage collection: an experience with Rotor

L. Veiga and P. Ferreira

Abstract: Support for distributed co-operative work implies object sharing. The memory management of these distributed (and possibly persistent) objects is a very difficult task. When done manually, it leads to memory leaks (useless objects that were not deleted) and dangling references (references to objects erroneously deleted), causing applications to fail. These errors have a strong negative impact on a programmer's productivity and program robustness. The authors address this problem by developing a complete distributed garbage collection algorithm. This solution is based on: (i) a reference-listing algorithm; and (ii) a centralised algorithm that complements the previous one by detecting distributed cycles of garbage. The detection and reclamation of distributed garbage cycles does not need any kind of global synchronisation. To achieve this goal they introduce the notion of a GC-consistent cut for distributed systems. They have implemented their algorithms in Rotor. Such an extension of the Rotor capabilities (which already includes a local garbage collector and use of leases for distributed garbage collection) is very important for supporting co-operative work among different users. The performance results obtained are very encouraging.

1 Introduction

Support for distributed co-operative work implies object sharing. The memory management of these distributed (and possibly persistent) objects is a very difficult task. When performed manually, it leads to memory leaks (useless objects that were not deleted) and dangling references (references to objects erroneously deleted), causing applications to fail. These errors have a strong negative impact on a programmer's productivity and program robustness.

However, .Net does not support distributed garbage collection (DGC). The current approach to DGC in .Net is very simple. Leases are associated with remote objects; when an object X is not remotely invoked for a certain amount of time (bigger than its lease) the reference pointing to X is ignored for reachability considerations; therefore, X may be collected even if there is still a remote reference pointing to it. This means that safety is not ensured. To address this problem we have developed a distributed garbage collector and implemented it in Rotor (shared source version of .Net). Such an extension of the Rotor capabilities is very important for supporting co-operative work among different users.

Our solution is based on: (i) a reference-listing algorithm [1] that replaces the leasing mechanism, and (ii) a centralised algorithm that complements the previous one by detecting distributed cycles of garbage. The reference-listing algorithm is safe and live. However, it is not complete as it does not reclaim distributed cycles of garbage. We solved this limitation by developing another algorithm, called cycles detector, capable of detecting such cycles asynchronously. Once a cycle is detected, the cycle detector instructs the reference-listing algorithm to delete one of its entries so that the cycle is eliminated. Then, the reference-listing is capable of reclaiming the remaining garbage objects.

The detection of distributed cycles of garbage works on a view of the global distributed graph that is consistent for its purposes. As explained later, this view may not correspond to a consistent cut (as defined by Lamport [2]) but it still allows us to safely detect distributed cycles of garbage. This view results from a cut that we call a GC-consistent cut. GC-consistent cuts can be obtained without requiring any distributed synchronisation among the processes involved.

Thus, our contributions are the following: (i) a complete distributed garbage collection algorithm, (ii) the notion of GC-consistent-cuts and (iii) an implementation within Rotor.

2 Distributed garbage collection

The only objects effectively available to an application are those that are reachable (either directly or indirectly by transversing one or more references, i.e. in one or more hops) from some considered root-set and, therefore, subject to being read or modified by the application; these objects are called live objects. All remaining objects are garbage and are only wasting memory and should be automatically discarded.

Garbage collectors can be broadly categorised into three main families: tracing collectors, reference counting (with several variations) and hybrids of these two previous families [3]. Tracing collectors are the only ones able to reclaim cyclic garbage, i.e. they are complete. Therefore, most local garbage collectors are of this kind.

However, tracing algorithms do not scale well to distributed systems as they traditionally impose inconvenient disruption (synchronisation and pause times) to

IEE Proceedings online no. 20030993

doi: 10.1049/ip-sen:20030993

Paper received 29th June 2003

The authors are with INESC-ID/IST, Rua Alves Redol No 9, 1000-029 Lisboa, Portugal

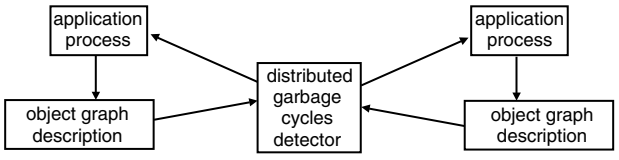


Fig. 1 *Integration of distributed garbage cycles detector in the system*

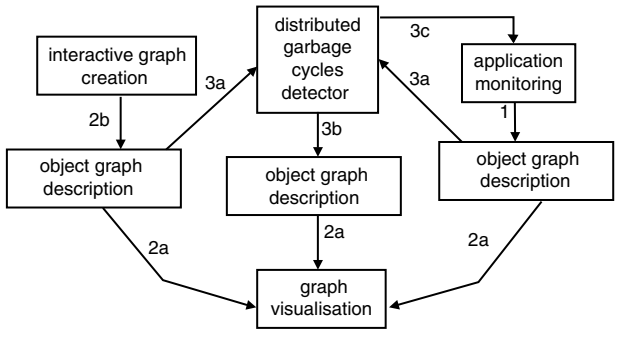


Fig. 2 *System overview*

applications in order to perform correctly. Therefore, most solutions rely on reference-counting algorithms for DGC due to scalability and performance reasons. However, reference counting algorithms are not able to identify and reclaim cyclic garbage, either local or distributed.

This raises the issue of completeness, i.e. how to identify and reclaim distributed cycles of garbage. We address this issue with a combined, hybrid approach, comprising an acyclic reference-listing distributed garbage collector and a centralised, deferred, tracing-based, detector of distributed garbage cycles.

This cycles detector (see Fig. 1) receives from each application process a file describing its enclosed object graph. Note that, with this information, the cycle detector does not perform a full garbage collection (as in Liskov's proposal [4]). As described later, the main task of the cycle detector is to detect distributed cycles of garbage, thus complementing the reference-listing algorithm.

The cycle detector and the reference-listing algorithm mentioned above are both part of the comprehensive approach we followed to extend Rotor with DGC capabilities. Figure 2 provides a general view of our environment:

- application monitoring: implements the reference-listing algorithm and generates the description of the enclosed object graph (1)
- an object graph visualisation interactive tool that:
  - displays the object graph based on an object graph description (2a), and
  - allows the user to draw an object graph and generate its corresponding description (2b)
- a distributed garbage cycles detector that, based on object graph descriptions (3a), detects distributed cycles of garbage, and generates the resulting object graphs (3b), and instructs applications to delete any scions found to belong to a distributed cycle of garbage (3c).

In the following Section we present the reference-listing algorithm that collects distributed acyclic garbage. Then, we describe the distributed cycle detector algorithm.

2.1 Acyclic collector

The algorithm for acyclic DGC is based on reference-listing [1]. This algorithm keeps track of inter-process references by means of data structures called stubs and scions. Thus, while remote references in Rotor are represented by proxies for invocation purposes, scions and stubs (both created and managed by the new code we wrote), are fundamental for GC purposes.

2.1.1 Data structures: A scion represents an incoming reference, i.e. a reference pointing to an object in the scion's process. A scion points to the target object and includes a unique time-stamp, i.e. a numeric value provided by a monotonic counter global to the enclosing process. The usefulness of time-stamps [5, 6] is explained afterwards.

A stub represents an outgoing remote reference, i.e. a reference pointing to an object in another process. A stub points to the remote object and includes the time-stamp of its correspondent scion.

In each process, stubs and scions are grouped in sets, for performance reasons. All stubs in a set have their corresponding scions in the same remote process. Conversely, all scions in a set have their corresponding stubs in the same remote process. There is only one stub-scion pair for each remote object and for each pair of processes. This also holds when there are remote references from various objects in one process to a single object in another process.

Scions and stubs are created according to the creation of inter-process references. These can be exported to a remote process or imported into a process (see Fig. 3). Reference export includes: (i) passing a reference to a local object as an argument of a remote method in a different process; or (ii) returning a reference to a local object as a method result to a different process. Reference import includes: (i) receiving a reference to a remote object as an argument of a method made available by the process, invoked by different processes, or (ii) receiving a reference to a remote object as a result of a remote method invocation in another process. Every time a reference to a local object in a process is exported, the corresponding scion must be created. Every time a reference to a remote object is imported, the corresponding stub must be created.

2.1.2 Messages: Messages exchanged by processes w.r.t. the DGC algorithm are of two kinds: implicit and explicit. Implicit messages are those bearing object references and must be intercepted, in remoting services code, in order to create the corresponding scions and stubs.

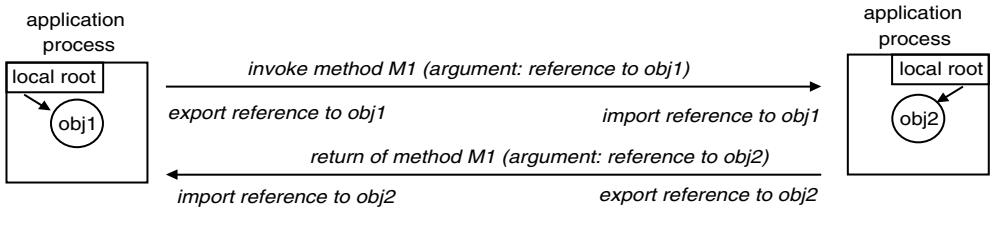


Fig. 3 *Export and import references*

For safety reasons, scions are always created before the corresponding stubs. Thus, scions are created when references are exported and the corresponding stubs are created only when the messages bearing those references arrive at the referring process.

After a local garbage collection (LGC) is complete, a new set of stubs can be generated (not necessarily every time) accounting for every remote object referenced from the local process. This new set of stubs is then sent to remote processes; these processes, based on the set of stubs received, may conclude which scions are no longer reachable so that they can be safely deleted. Objects that are only reachable through these, just deleted, scions are garbage and can be reclaimed by the next LGC. Thus, the only explicit message exchanged among participating processes is NewSetStubs. In particular, there is no use of a special message for scion deletion except to break a distributed cycle of garbage.

To minimise applications' disruption, determination of each new set of stubs and actual sending of the corresponding NewSetStubs messages are both fulfilled lazily; thus, these operations do not increase the amount of time an application is stopped due to the LGC.

Attached to each NewSetStubs message there is always the highest scion time-stamp the sending process knows of. This ensures that the receiving process will not prematurely eliminate scions, recently created, whose corresponding stubs are not yet included in NewSetStubs message just received. Such a situation may occur as follows (see Fig. 4): (i) an invocation takes place from process P1 to process P2; (ii) the message is scanned in P1 for references being exported; (iii) the corresponding scions are created in P1 (e.g. assume that scion SC1 is created); (iv) the invocation message is sent; (v) before the invocation message reaches P2, P2 sends a NewSetStubs message to P1; (vi) the message NewSetStubs received by P1 does not contain the stub corresponding to scion SC1; however, this scion is clearly reachable, and thus should not be deleted.

A set of stubs represents a view of outgoing remote-references in the enclosing process. To be consistent for GC purposes, this view must be associated with the time it was taken so that scions can be safely deleted. This is achieved in the following manner at the process receiving a NewSetStubs message:

- only those scions corresponding to inter-process references originating in the process that sent the NewSetStubs have to be considered
- for each one of these scions - only those with a time-stamp not greater than the time-stamp received in the NewSetStubs message - the existence of the corresponding stub is tested
- if there is a corresponding stub, the scion is preserved (as it is still being referenced); if there is no such stub, the scion is deleted (as it can be safely assumed that the stub no longer exists and, therefore, there are no longer any references to that scion).

Thus, scions can be regarded as assuming one of three states: (i) unreachable scions are those that can be safely removed because they are found to have no corresponding stub and there is no chance of that stub being 'in-transit'; (ii) reachable scions are those with corresponding stubs effectively detected; and (iii) uncertain scions are those that, as far as the referred process is concerned, have no corresponding stubs, but it cannot be safely assumed that they are unreachable; invocation messages leading to the creation of stubs could still be in transit and the algorithm, conservatively, considers these scions as reachable.

In summary, since there are not any competing, i.e. conflicting, types of messages (e.g. create/delete stubs/scions), there are no distributed races during creation and elimination of remote references. It is required that processes maintain a record of the highest scion time-stamp

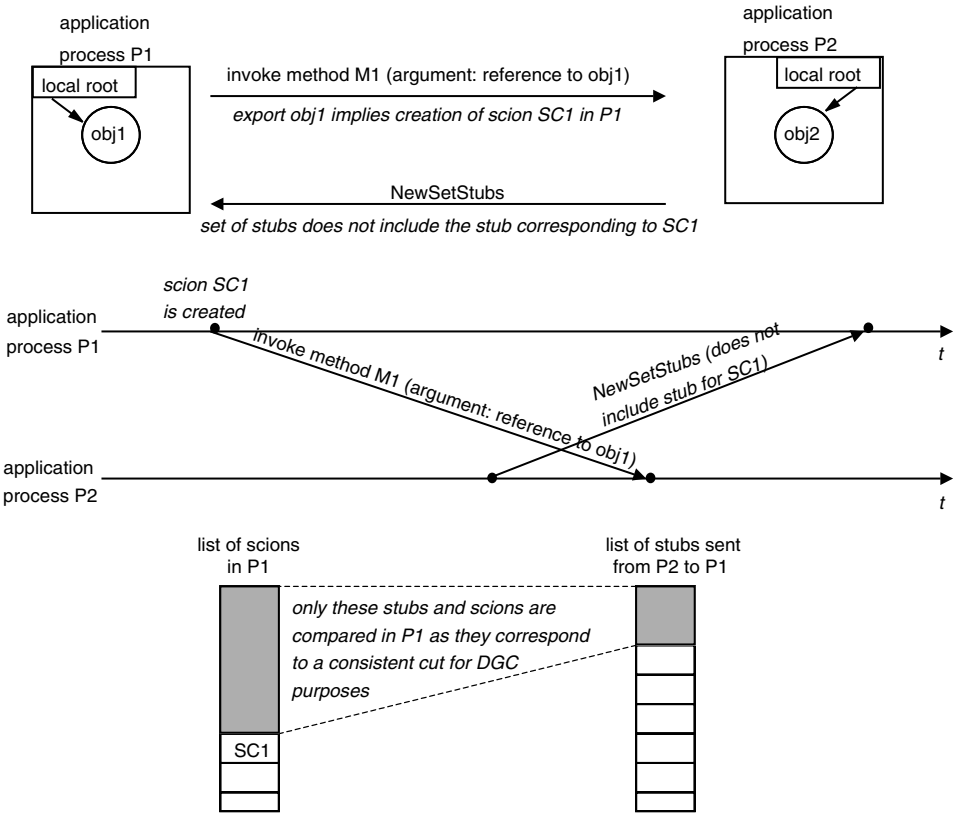


Fig. 4 *Race between scion creation and message NewSetStubs*

received from every other process, constituting, in fact, a vector-clock [2]. As explained in Section 2.2, these vector clocks (one at each application process) also allow establishing cuts of the distributed graph, which are consistent from the point of view of the distributed cycles detector.

2.1.3 Properties: The algorithm described is safe since objects still referenced (locally or remotely) are never reclaimed. Obviously, objects just locally referenced are never wrongly reclaimed as we assume the correctness of the LGC in Rotor.

Objects referenced just by remote references can fall in one of three cases, according to scions pointing to them: reachable, unreachable and uncertain objects. As explained earlier, safety is ensured in all three cases. The algorithm is resilient to lost, duplicate and un-ordered NewSetStubs messages because each set of stubs is matched with the corresponding scions, in a consistent manner, due to time-stamps.

Liveness is, naturally, dependent on processes sending NewSetStubs messages frequently enough. Unreachable scions can only be detected after a NewSetStubs message is received.

Note that the algorithm can be divided into several steps that can be performed lazily:

- after an LGC, determine the set of reachable stubs to include in a new NewSetStubs message with the proper time-stamp; this can be performed lazily and incrementally
- send NewSetStubs message(s) can be done at any time
- matching stubs and scions can be performed lazily and incrementally (see Section 3).

In summary, the distributed algorithm works lazily and with loose synchronisation requirements. This not only increases its scalability but also reduces applications' disruptiveness to a minimum.

The algorithm is not complete w.r.t. distributed cycles of garbage. This issue is addressed by a specific cycles detector that is described in the following Section.

2.2 Distributed garbage cycle detection and reclamation

A distributed cycle of garbage is a partition of the object graph spanning various processes that is completely detached from the set of live objects. However, the reference-listing algorithm is not able to reclaim it due to circular references.

To achieve completeness in DGC it is necessary to detect and delete distributed cycles of garbage. This is a difficult problem that has been addressed in many ways: global tracing, back-tracing, detection within groups, with centralised or distributed approaches; we present the work most relevant to ours in Section 5. Our algorithm makes use of a centralised approach. Such an approach was first introduced in [4]. However, our work has several differences that will become clear afterwards.

2.2.1 Overview: The process of performing the detection of the distributed cycles of garbage (called CDP for cycles detector process) works on a view of the global distributed graph that is consistent for its purposes. As explained in this Section, this view may not correspond to a consistent cut (as defined by Lamport [2]) but it still allows us to safely detect distributed cycles of garbage. We call such a cut, a GC-consistent cut.

GC-consistent cuts can be obtained without requiring a distributed consensus [7] among the applications processes that send their graph descriptions to the CDP. This means that the CDP still performs useful work, i.e. it is capable of detecting cycles, even if its global view of the graph is made of local graph descriptions (sent by the applications processes) at different and unco-ordinated moments.

The CDP is also capable of performing its task without requiring every existing process to send its graph description. The only consequence is that cycles comprising objects in such processes are not detected. However, all other cycles are detected and reclaimed.

2.2.2 Algorithm: The CDP runs the cycles detector algorithm to discover which stubs and scions are part of distributed cycles of garbage. Then, it instructs certain applications processes to delete one or more of their scions. The explicit deletion of such scions is safe because garbage is stable, i.e. once an object is garbage, it stays so. It transforms a distributed cycle of garbage into a set of acyclic garbage objects; then, such objects can be reclaimed by the reference listing algorithm described in the preceding Section.

To perform the cycle detector algorithm, the CDP receives a description of the object graph of applications processes [Note 1]. Note that such object graphs are strictly local to each application process. In addition, as will be made clear later, the CDP does not require the object graph of every existing application process to perform useful work.

The object graphs description received by the CDP can be seen as snapshots of each application process. However, note that our algorithm does not require these snapshots to be taken synchronously by every application involved. In other words, there is no need for a distributed consensus [7], which would clearly be a bad solution for performance and scalability reasons. Thus, as explained now, the CDP analyses the object graphs with special care for consistency and causality from a DGC point of view.

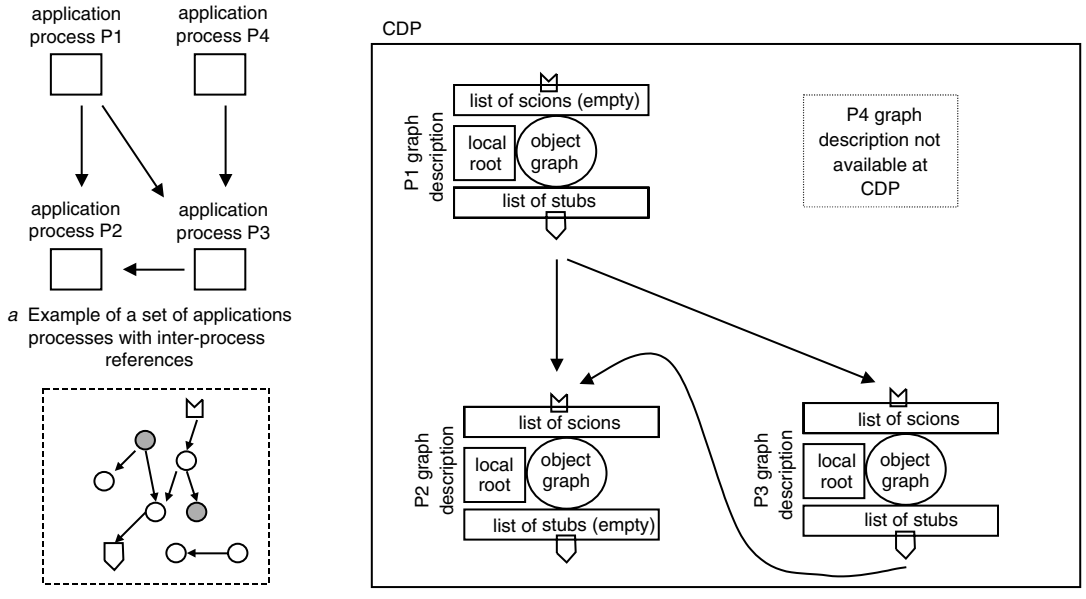
The CDP performs a global mark-and-sweep (GMS) on the graphs description received. This GMS is done in such a way that inter-process references are followed only if the corresponding stub-scion pair exists in the graphs description. Otherwise, the marking on that reference stops.

The roots of the GMS are the following (see Fig. 5):

- Those objects that, in each application process, are directly reachable from the local roots (stack, etc.), must be obviously considered roots of the GMS (in Fig. 5*c* such objects are shaded).
- Scions whose corresponding stubs are included in processes whose graph description is not being considered by the CDP in the GMS are also members of the GMS root (in Fig. 5*b* such a scion is the one in P3 whose corresponding stub is in P4). These scions are members of the GMS root for safety reasons. As a matter of fact, such scions may not have a corresponding stub (so they could be simply discarded) but the CDP cannot say that for sure. Thus, it uses a conservative approach.
- Those scions whose associated time-stamp has a value greater than the value known in the process holding the corresponding stub, are also members of the GMS root.

As for the previous item, this is also a conservative approach. These scions are those whose corresponding stubs have not been created yet when their enclosing application process has created its graph description (then sent to the CDP).

Note 1: A description of an object graph is obtained using a library that, through serialisation, writes a file describing the objects, stubs and scions of the process. This description is subject to a reduction process.

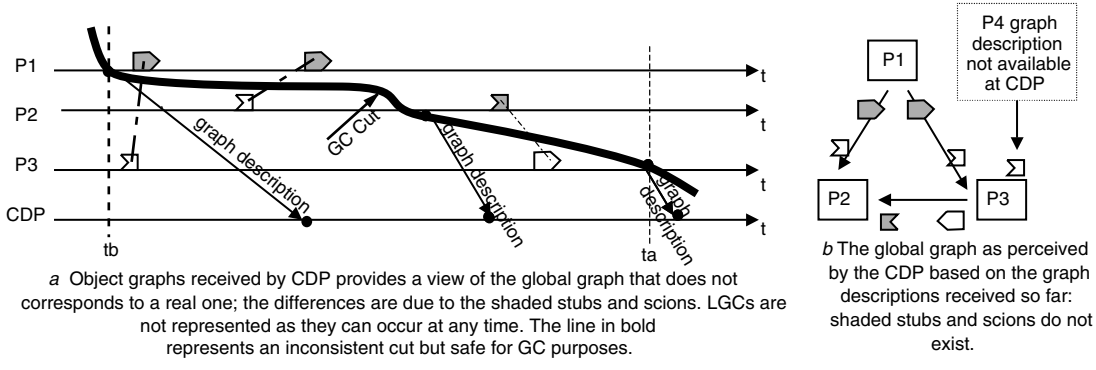


a Example of a set of applications processes with inter-process references

c Example of an object graph description in CDP

b Graph descriptions (at CDP) of applications processes P1, P2 and P3

Fig. 5 *Roots of GMS at CDP*



a Object graphs received by CDP provides a view of the global graph that does not corresponds to a real one; the differences are due to the shaded stubs and scions. LGCs are not represented as they can occur at any time. The line in bold represents an inconsistent cut but safe for GC purposes.

b The global graph as perceived by the CDP based on the graph descriptions received so far: shaded stubs and scions do not exist.

Fig. 6 *View of graphs as seen by CDP*

Note that this may occur because the graph descriptions received by the CDP are snapshots taken at different moments at different processes with no co-ordination at all. Such a situation is illustrated in Fig. 6.

At moment `ta` the graphs are in fact those as illustrated in Fig. 5*a*. However, the view CDP has, based on graph descriptions received so far, is different because the graph descriptions obtained from P2 is older than P3 and the graph description of P1 is even older than P2's. The shaded scions and stubs reflect such differences.

The result from the GMS is a set of garbage stubs and scions. However, not all of these scions belong to distributed cycles of garbage. Some of these scions (those that are not members of distributed cycles of garbage) are reclaimed by the reference-listing algorithm.

The CDP determines which of such scions actually belong to cycles. This is done as follows: only scions that are simultaneously garbage and, still, referenced by stubs, can belong to a distributed cycle of garbage. Then, one, any, or all of them, can be selected for deletion and the message(s) sent to the corresponding process(es). The number of messages sent only influences the bandwidth used and the speed of cycle reclamation. Those distributed garbage cycles that already existed when the oldest graph description (being processed by the CDP) was created, and are totally included in the graph descriptions available at the CDP, are effectively detected and reclaimed. Thus, considering Fig. 6*a*, all cycles that existed before `tb`, that are totally enclosed in processes P1, P2 and P3, are detected by the CDP.

In Fig. 6*a* we show, in bold, a cut that is not causally consistent for typical GC purposes; it results from the unco-ordinated creation and sending of object graphs from each application process to the CDP. However, based on such graphs, the CDP builds a GC-consistent cut that allows it to detect distributed cycles of garbage. This cut is consistent w.r.t. the finding of such cycles. Thus, a GC-consistent cut is a group of scions and stubs that provide a safe view of the distributed object graph. This group of stubs and scions provides a safe view of the distributed graph as long as the rules to define the root-set of the GMS (performed by the CDP) are respected. In particular, these rules specify which scions are members of the root-set of the GMS.

2.2.3 Graph reduction: Objects graphs in application processes may be very large. Consequently, the size of their descriptions may contribute to increasing the load on the network and occupies a large amount of disk space. In addition, such a large amount of data makes the GMS performed by the CDP a CPU-consuming operation as it requires accessing a large amount of data.

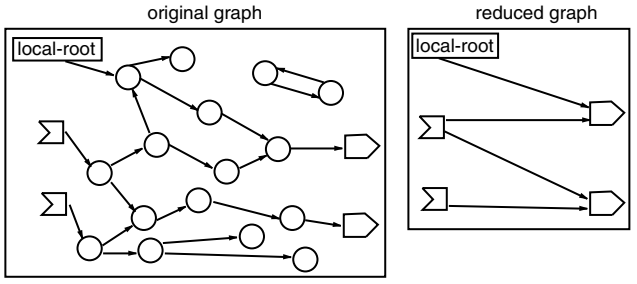


Fig. 7 *Reduction of an object graph*

This problem is solved by reducing the objects graph of each application process in such a way that, from the point of view of the cycle detector algorithm, there is no loss of relevant information. This reduction transforms an object graph of an application process into a set of scions and stubs. As a matter of fact, references strictly internal to a process are not relevant for the cycle detector algorithm.

The reduced graph contains: (i) the set of stubs of the process with information, for each stub, about its reachability from the local root; and (ii) the set of scions of the process with a list, for each scion, of the stubs reachable from that scion. This reduction (illustrated in Fig. 7) is performed at each application process; then the reduced object graph is sent to the CDP. The reduced graph has obvious advantages both in terms of network and disk usage. In addition, this task can be performed lazily, with low priority, with minimal impact on application performance.

We can estimate the importance of cycles based on work done by Richer[8]. In this work, research about the memory behaviour of the World Wide Web, regarded as a persistent store, revealed that a large proportion of objects are involved in distributed cycles; still, most individual cycles are small both in number of objects and in space occupied.

To remain safe, the CDP can only detect distributed cycles of garbage that are fully enclosed in the graph descriptions it holds. This may suit most distributed cycles, that are small, but clearly limits the maximum size of the detectable cycles. However, this limitation can be easily solved because it is possible (and desirable, for scalability and availability purposes) to have several CDPs. These CDPs can be organised hierarchically (or in any other way) so that a CDP at a higher level has a larger view of the global distributed object graph. Such a larger view is obtained as follows. Each CDP applies a graph reduction on the set of graphs it holds and then sends the reduced graph to its parent CDP. With this scheme, it is possible to detect (and reclaim) any distributed cycle of garbage independently of its size.

3 Implementation

The algorithms (reference-listing and cycle detector) were implemented combining different programming languages: C++, C# and Lisp. These languages were chosen to favour integration, simplicity and efficiency. The implementation includes virtual machine modification (for LGC and DGC integration), remoting code instrumentation (to detect export and import of references), distributed cycle detection and graph generation and visualisation.

Virtual machine modifications were implemented in C++, the language Rotor core is implemented in. Remoting instrumentation code was developed in C#, since high-level code of the remoting services is already written in this language. The graph generation, reduction and visualisation tool was also developed in C# but other .Net languages could have been used. The cycle detector was written in Lisp, mainly, for simplicity of writing code for graph manipulation.

3.1 Virtual machine monitoring and modification

The reference-listing algorithm must co-operate with the LGC, essentially in two ways:

- the LGC must provide, in some way, the reference-listing algorithm with information about every remote object referenced by local objects; this is necessary to ensure that all stubs (representing outgoing remote references) are correctly created/preserved
- the reference-listing algorithm must prevent the LGC from reclaiming objects that are no longer locally reachable but are targets of incoming remote references; this ensures that scions actually prevent objects from being reclaimed.

To fulfil these requirements, we had to choose between two main options: (i) to extend Rotor's LGC, modifying its implementation, in order to make it able to create/preserve stubs for every outgoing remote reference or, at least, feed this information to the DGC component that would create them; or (ii) attempt to indirectly detect which outgoing remote references disappeared and eliminate the associated stubs.

Both approaches need instrumentation of Rotor's remoting services. The first approach consists in modifying the code of Rotor's LGC so that every field (possibly containing a reference) in every live object is examined. If this reference happens to point to an object that is a transparent proxy, it is an outgoing remote reference and the corresponding stub must be re-created.

The second approach consists simply of a running thread that monitors existing stubs verifying that they are still valid, i.e. the transparent proxies associated with them still exist. This is achieved using weak references.

Both approaches have their advantages and drawbacks. The first option would determine stub deletion more quickly but it could impose larger pause-time on applications since all stubs would be re-created each time over. This would be implemented with low-level programming, i.e. C/C++ language so that penalty could be minimised. However, the advantage of determining stub deletion more quickly is mitigated by the fact that DGC processing is essentially bound to the exchange of messages, and these are sent in a lazy manner, in order not to disrupt running applications.

Moreover, the second approach is more advantageous for additional reasons: (i) it does not impose relevant modifications on the CLR implementation; (ii) it can be implemented using a high-level language such as C#; (iii) modifications are mainly restricted to the remoting package; and (iv) it does not interfere with the LGC used.

3.2 Remoting code instrumentation

Remoting services code instrumentation intercepts messages sent and received by processes in the context of remote invocation so that scions and stubs are created accordingly [Note 2].

Note 2: In Rotor, messages exchanged by these services are created, intercepted, coded and decoded in several stages, called sinks. A group of different sinks that sequentially process a message constitutes a sink chain.

In our current implementation we only intercept remote invocations in which MarshallByRef references are exported/imported. Thus, every time a reference to an object extending MarshallByRef is exported/imported it must be accounted for DGC purposes. Additionally, the reference-listing algorithm demands scions to be time-stamped when they are created and that the same time-stamp is applied to its counterpart stubs. This implies that scion time-stamps must be included in remote invocation messages bearing remote references.

To accomplish this, three new custom headers were appended to messages: *scionIndex*, *machineId*, *processId* to uniquely identify scions and stubs associated with remote references included in messages. These values must be propagated throughout the entire sink chain. Therefore, adaptations were made on base files as *basetransportheaders.cs*, *corechannel.cs*, *message.cs*, *dispatchchannelsink.cs*, *binaryformattersink.cs*. Higher level files such as *remotingservices.cs*, *tcpsocketmanager.cs*, *binaryformatter.cs* and *activator.cs* were also modified mainly to invoke, when remote references were detected, specialised methods included in the previous files. One specialised file, *gcdata.cs*, implements a new class, *GCManager*, containing all the utility methods and GC state used in all other files.

3.3 Scions and stubs

Stubs and scions are maintained in hash tables within each application process. These are implemented as specific classes, maintained as structured containers with the appropriate selectors, modifiers and with synchronised access.

Code implementing DGC explicit messages is grouped in a specific class *DGCManager*; this code runs as a low priority thread in each application process, and is responsible for managing stubs, and composing and sending NewSetStubs messages lazily. NewSetStubs messages from other processes are delivered when a well known remote method made available by DGCManager is invoked by another process.

3.4 Distributed cycles detection

The distributed cycles detector is implemented in C# and Lisp. The choice of the implementation language of the cycles detector had no constraints, provided that interoperability could be fulfilled between the cycles detector and the graph serialisation/reduction component in each process.

To be consistent, object graph serialisation must be performed while the application code is not running. This occurs, for example, during an LGC. If done immediately after the LGC and before allowing the application to proceed, the extra time taken to create the object graph may cause longer and disruptive application pause times, which is clearly undesirable. However, this object graph is needed only for cycles detection. Thus, a new graph does not have to be created each time an LGC occurs. Furthermore, it only needs to be done occasionally. This allows the creation of the object graph in other more convenient situations, such as when the application stops waiting for input, or is idle.

Graph reduction is performed incrementally, in each process, after a new object graph has been serialised, by a separate thread (which is almost always blocked) or by another offline process. Once reduced, the graph can be sent, lazily, to the CDP.

Table 1: Remote invocation in original Rotor and DGC-extended Rotor (times in ms)

Number of invocations	Rotor	Rotor with DGC
10	1933	2072
100	12417	14731
500	58754	70931
1000	118890	140191

4 Performance

The most relevant performance results of our implementation are those related to phases critical to applications performance: stub/scion creation and object graph serialisation [Note 3].

We measured the creation of stubs and scions when remote references are exported/imported in remote invocations; these operations are always performed and cannot be fulfilled lazily. We tested worst case scenarios that discard potentially long network communications times, that could mask stub and scion creation overhead. Table 1 shows results for different series of remote invocations of a remote method, with 10 arguments (10 different references being exported/imported), where client and server processes execute in the same machine. This forces the DGC to create 10 scions and stubs each time the remote method is invoked. The overhead associated with the creation of DGC stubs and scions, in this worst case scenario, is within 7%-20%, which is acceptable for the functionality provided.

The results regarding graph serialisation, which does not have to be performed frequently, were also encouraging. On average, for graphs with 10000 dummy objects (just references), Rotor serialisation takes 25730 ms [Note 4]. To serialise the same graph, with every object containing an additional remote reference (additional 10000 proxies), takes 34489 ms (34% slower). Thus, serialising remote references is three times faster than serialising an additional object. This result should also be regarded as upper bound values, since, in normal circumstances, application graphs have much higher density of local than of remote references.

5 Related work

Because of space restrictions, and given that the most interesting contribution of this work is the cycles detector algorithm, we focus this section on other proposals for collecting distributed garbage of cycles. Distributed garbage collection has been a field of active study for many years and recently, as well [9–13]. In Fessant [10], time-stamps of stubs and scions are propagated until a global minimum can be computed. (Propagation of time-stamps was first proposed in Hughes [5].) Cycles are detected with the help of optimistic backtracking.

Distributed garbage collection based in cycle detection within groups of processes was first introduced in Lang *et al.* [14]. In Rodrigues and Jones [12] groups of processes are created, with less synchronisation requirements to detect cycles exclusively comprised within them. Groups of

Note 3: Results obtained using a Pentium 4 Mobile 1600 MHz with 512 Mb RAM.

Note 4: Note that, in .Net, serialisation is roughly 100 times faster.

processes can also be merged and ongoing detections can be re-used.

In Liskov [4], distributed garbage collection is performed by a logically centralised server that receives graph information from every process. Requirements on clock synchronisation and message latency are strict. The centralised server performs complete distributed garbage collection and informs processes of objects deletion.

Our algorithm combines properties found in previous work. However, since it uses a combined approach using a DGC algorithm and a specialised cycle detector, it has no need for some potentially costly operations such as mark-propagation, backtracking or distributed consensus. A safe and efficient DGC algorithm based on reference-listing is paired with a cycle detector that detects cycle lazily, in the background, combining local graphs, with very low consistency requirements.

Another important aspect of our solution is the use of a description of distributed graphs that is less restrictive than Lamport's consistent causal cuts (called GC-consistent cuts). Nevertheless, a GC-consistent cut is safe; its and complete as long as the graph description received from each process is eventually updated.

Our notion of GC-consistent cut can be related to GC-consistent cuts in databases as proposed by Skubiszewski and Valduriez [15]. In his work a GC-consistent cut has one or more copies of every page in the database. These copies, possibly inconsistent from a transactional point of view, can be created at different instants. However, all these pages, when combined with knowledge from database locks, may be consistently and safely used for LGC purposes. This work can be applied only to a centralised database system, it is not distributed, and is strongly dependent on the specific information provided by the database synchronisation mechanisms.

Our GC-consistent cuts apply to distributed systems and do not require any kind of synchronisation information about participating applications. Obtaining and managing such synchronisation information, in distributed systems, would be clearly undesirable for scalability and performance reasons.

6 Conclusions

We have presented a comprehensive solution for the problem of distributed garbage collection. The main results of our work are the following: (i) a reference-listing DGC algorithm running on Rotor; (ii) a centralised cycles detector algorithm that requires no global synchronisation, is scalable and makes progress without requiring all processes to participate; (iii) an implementation on Rotor with minimum impact on the source code of the Common Language Runtime; (iv) the notion of a GC-consistent cut in DGC; and (v) a set of tools to monitor Rotor applications and to visualise the object graphs, along with an editor that can be used to specify such graphs.

Finally, although we have implemented the DGC algorithms in Rotor, our solutions are rather general. It is possible to apply the same ideas and, in particular the notion of a GC-consistent cut and the cycle detector algorithm, to other similar platforms. In the future we plan to address the formal correctness proof of the cycle detection algorithm, develop a distributed version of the algorithm and minimise the complexity of the reducing process.

7 Acknowledgments

We thank students Ezequiel Alabaça and Ricardo Mendes for their implementation work. This work was partially supported by Microsoft Research.

8 References

1 Shapiro, M., Dickman, P., and Plainfossé, D.: 'Robust, dist. references and acyclic garbage collection'. Proc. Symp. on Principles of Distributed Computing, Vancouver, Canada, Aug. 1992, pp. 135–146
2 Lamport, L.: 'Time, clocks and the ordering of events in a distributed system', *Commun. ACM*, 1978, **21**, pp. 558–565
3 Plainfossé, D., and Shapiro, M.: 'A survey of dist. garbage collection techniques'. Proc. Int. Workshop on Memory Management, Kinross, UK, 27–29 September 1995, pp. 211–249
4 Liskov, B., and Ladin, R.: 'Highly-available distributed services and fault-tolerant distributed garbage collection'. Proc. 5th Symp. on Principles of Distributed Computing, Vancouver, Canada, Aug. 1986, pp. 29–39
5 Hughes, J.: 'A distributed garbage collection algorithm', *Lect. Notes Comput. Sci.*, 1985, **201**, pp. 256–272
6 Shapiro, M., Plainfossé, D., and Gruber, O.: 'A garbage detection protocol for a realistic distributed object-support system'. Tech. rep., INRIA, Nov. 1990, INRIA 1320
7 Fisher, M., Lynch, N., and Patterson, M.: 'Impossibility of distributed consensus with one faulty process', *J. ACM*, 1985, **32**, pp. 274–382
8 Richer, N., and Shapiro, M.: 'The memory behavior of the WWW, or the WWW considered as a persistent store'. Proc. 9th Int. Workshop on Persistent Object Systems (POS), Lillehammer, Norway, 6–8 September 2000, pp. 161–176
9 Abdullahi, S.E., and Ringwood, G.A.: 'Garbage collecting the internet: a survey of distributed garbage collection', *ACM Comput. Surv. (CSUR)*, 1998, **30**, (3), pp. 330–373
10 Fessant, F.L.: 'Detecting distributed cycles of garbage in large-scale systems'. Proc. 20th Symp. on Principles of Distributed Computing (PODC), Newport, R.I., 26–29 August 2001, pp. 200–209
11 Louboutin, S.R.Y., and Cahill, V.: 'Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events'. Proc. Int. Conf. on Distributed Computing Systems (ICDCS), Baltimore, MA, 1997, pp. 516–525
12 Rodrigues, H., and Jones, R.: 'Cyclic distributed garbage collection with group merger', *Lect. Notes Comput. Sci.*, 1998, **1445**, pp. 260–284
13 Shapiro, M., Fessant, F.L., and Ferreira, P.: 'Recent advances in distributed garbage collection', *Lect. Notes Comput. Sci.*, 2000, **1752**, p. 104
14 Lang, B., Quenniac, C., and Piquer, J.: 'Garbage collecting the world'. Conf. Record 19th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM, Jan. 1992, pp. 39–50
15 Skubiszewski, M., and Valduriez, P.: 'Concurrent garbage collection in O2'. Proc. 23rd Int. Conf. on Very Large Databases (VLDB), Athens, 25–29 August 1997, pp. 356–365