Complete distributed garbage collection using DGC-consistent cuts and .NET AOP-support

L. Veiga, P. Pereira and P. Ferreira

Abstract: The memory management of distributed objects, when done manually, is an error-prone task. It leads to memory leaks and dangling references, causing applications to fail. Avoiding such errors requires automatic memory management, called distributed garbage collection (DGC). Current DGC solutions are either not safe, not complete or not portable to widely used platforms such as .NET. As a matter of fact, most solutions either run on specialised environments or require modifications of the underlying virtual machine (e.g. rotor, common language runtime (CLR)), hindering its immediate and widespread utilisation. This study describes the design, architecture, implementation and performance measurements of a DGC algorithm for .NET that: (i) is complete, that is, capable of reclaiming both acyclic and cyclic garbage, while (ii) being portable in the sense that it neither requires the underlying virtual machine to be modified, nor source or byte-code modification. The distributed garbage collector was implemented on top of two implementations of the common language infrastructure (.NET virtual machine specification): CLR and shared source CLI, commonly known as Rotor. The implementation requires no modification of the environment, it makes use of the provided aspect-oriented functionalities, and the performance results are encouraging.

1 Introduction

There are several arguments that justify the existence of a system-provided memory-recycling service called garbage collector (GC). These arguments, extensively presented in the context of local garbage collection [1], range from the description of the consequences of errors that result from manual management, namely unreclaimed memory (memory leaks) and premature reclamation (dangling references), to, according to Wilson [2], the classification of the existence of such a service as a fundamental requirement for accomplishing program modularity. Besides programming soundness, GC offers performance benefits since it improves object locality [3]. The strength of the previous arguments has been acknowledged with the inclusion of GC services in platforms with wide industrial usage, such as .NET and Java.

In distributed object systems, the same arguments apply, simply because these systems result from the extension of the programming model offered in non-distributed ones. Considering that a local garbage collector exists, a natural extension would be to provide automatic reclamation of distributed objects. The CLI (Common Language Infrastructure, .NET virtual machine specification [4]) includes a distributed object system, which does not support automatic recycling of distributed memory. Alternatively, it offers a configurable object lifetime management service, based on renewable leases. This approach has the main drawback of being unsafe, that is, it may wrongly reclaim objects still referenced by some process. Furthermore, it places the problem of distributed memory recycling at the application programming level, leading to the previously mentioned errors.

Applications deployed on these scenarios are frequently developed around the abstraction of a single system image with a global object space (or distributed heap [5–9]), built by aggregating memory from each participating process on top of a distributed object system, fully transparent to the programmer. Examples of applications in these scenarios that need to perform navigation and manipulation of complex hierarchies of elements, comprising large distributed object graphs, include ray-tracing [8] and rendering utilities, virtual environment simulation, CAD/CAM project development tools (e.g. cooperative engineering, architecture, urban planning and circuit design), as well as web and social network analysis, and scientific computing (e.g. molecule geometry modelling and computational simulation of proteins). Note that these applications are different from service-oriented distributed applications (e.g. using XML, SOAP), where loose-coupling and interoperability among heterogenous systems are the main issues.

Other recent examples of DGC relevancy and usage include distributed virtual machines [10, 9], parallel computing [11], cluster computing [12, 13 and 5] (also integrating grid infrastructures [14]) and distributed shared memory [15]. It is specially important w.r.t. long-running applications (possibly including object persistence) where garbage accumulates and hinders performance.

This document describes an extension of .NET Remoting with a safe, complete, scalable and portable distributed garbage collection service. .Net Remoting, as the basis of application interoperability within the .Net Framework, is being maintained in Windows Communication Foundation [16] that also includes support for service-orientation. The contribution comprises the design of a distributed garbage

doi:10.1049/iet-sen:20070015

Paper first received 5th February and in revised form 18th September 2007

L. Veiga and P. Ferreira are with INESC-ID / Technical University of Lisbon, Rua Alves Redol No. 9, Lisboa 1000-029, Portugal

P. Pereira is with CCISEL Rua Conselheiro Emídio, Navarro No. 1, Lisboa 1959-007, Portugal

E-mail: luis.veiga@inesc-id.pt

collection algorithm, and its implementation and evaluation in the context of an unmodified widely deployed virtual machine.

The proposed solution resorts to techniques based on the existing built-in aspect-oriented programming (AOP) support, allowing immediate adoption, while minimising the imposed application pause times because of DGC operation. It operates correctly in the presence of temporary failures (node or network). Permanent failures are considered to be dealt with by machine administrators.

Previous approaches to DGC completeness are not portable in the sense that they require modifications in the underlying virtual machines, or are specifically implemented in the context of custom systems, namely research prototypes. Moreover, the authors are not aware of any previous approach resorting to AOP-support for DGC purposes.

The decision of employing an AOP-based approach to DGC is motivated by its soundness, portability and efficiency. Memory management is a cross-cutting concern, which should not be entangled with application logic. Therefore manual memory management was abandoned in favour of garbage collection integrated in the virtual machine. It is therefore natural to consider distributed memory management (i.e. DGC), an aspect in distributed object systems such as .Net Remoting, and employ existing AOP-support. The portability stems from the fact that the solution presented does not require modification of the underlying virtual machine, and is valid in both implementations of the CLI available from Microsoft (CLR, commercially used CLI realisation, and Rotor, shared source CLI implementation), as well as in open-source efforts, such as Mono, which include AOP-support. Although Remoting services (i.e. application domains, proxies and `MarshalByRef` base class) are considered in the actual CLI specification, AOP-support is not and therefore formally not mandatory in every CLI implementation, despite being supported by Microsoft. Finally, in comparison with other portable approaches, such as fully extending remoting by inserting new types of client and server-side sinks in charge of monitoring every message exchanged or, at a lower-level, by extending application source-code or byte-codes to monitor reference-passing, it minimises message processing overhead, redundant programming effort and intrusion w.r.t. application development. Additionally, AOP has been successfully employed in distributed scenarios using .Net [7, 17] with built-in AOP-support, and Java [9] with load-time aspect-weaving.

The rest of the paper is organised as follows. The next section is dedicated to related work, evaluating relevant DGC algorithms found in the literature using a novel approach, w.r.t. their portability (i.e. level of runtime intrusion and coupling between GC components), which is a main goal of our work. Section 3 describes the underlying system model to this work. Section 4 is dedicated to the algorithms used for distributed garbage collection. We present a hybrid DGC algorithm that is safe (reclaims only garbage objects), live (it always makes progress), complete (it eventually detects all garbage objects) and scalable (able to deal with large number of objects and processes). The algorithm is further described using a detailed prototypical example, and its properties are analysed. In Section 5, we describe the features of the .NET runtime; this work is based on (i) remote method invocation (.NET Remoting) and (ii) built-in AOP-support available in the .NET platform (`Context` architecture). Section 6 presents the most important details of our implementation. It is portable, in the sense that it does not require any modifications neither to the .NET runtime, nor to existing libraries, nor to application source or byte-code. The section ends with performance results measuring the critical aspects to algorithm performance and scalability. The paper ends with some conclusions.

2 Related work

Distributed garbage collection has been extensively described in the literature [1, 18–21], and algorithms have been compared based on parameters such as asynchrony, message traffic, space and time overhead.

In these systems, distributed garbage, including distributed cycles, is frequent and has been characterised in [22, 23]. We use the term GC-solution to designate the set of components and algorithms involved in performing garbage collection, both local and distributed, and their actual implementation. Incomplete solutions are typically based on distributed reference-counting and reference-listing [24, 25]. Detection of distributed cycles has been addressed using (i) object migration, explicit [26] and via indirection [27] (train algorithm), (ii) trial deletion [28], (iii) propagation of marks or time-stamps, global [29–31], within groups [32, 33], (iv) distributed back-tracing [31, 34 and 35], (v) centralised detection, loosely-synchronised [36], and asynchronous [37], and (vi) cycle detection algebra [38].

Nonetheless, most of the solutions found in the literature are developed towards very specific systems, namely research prototypes, where it is assumed that the DGC developer has complete control over the runtime. When applied to a widely deployed runtime (as Java and .NET), these solutions, although some could be adapted, frequently require significant modifications to the underlying virtual machine.

Thus, we evaluate existing work on a different perspective, with portability in mind instead. We present a qualitative overview of two main issues that may hinder the adoption of a complete GC-solution to any widely adopted runtime: (i) runtime intrusion, and (ii) coupling between components of the GC-solution. Each of these aspects is decomposed in subaspects and, for each of them, we introduce a scale of approaches with increasing degrees of portability and/or flexibility. Some solutions may be mentioned at different degrees because of the different techniques they employ.

2.1 Runtime intrusion

Runtime intrusion is defined as the need to deviate from an existing runtime, to provide it with a specific garbage collection solution. These deviations may be caused by different GC components, and have different degrees. Naturally, the optimum degree is not requiring any intrusion at all, and this is the case when a specific solution is not explicitly mentioned.

Local GC: The most inflexible technique, with respect to *LGC*, when adopting a GC-solution is to impose an *heterodox LGC* [27, 30], substantially different from those typically included in the runtime. A GC-solution may require the *extension of reachability encoding* of an existing LGC. This is the case in solutions that require the LGC to incorporate in object headers, more colour-bits [32, 33] or additional marks, such as time-stamps [29–31] or reachability-maps [34, 36].

An existing LGC may also be subject to *extension of operation* that is less intrusive than the previous technique, either prepending or appending operations to the ones already performed by the existing LGC, such as generating stub sets [24, 25], or calculating backward references [35].

A solution may impose *direct instrumentation*, in that the existing LGC must be suspended [33, 34] or triggered at specific moments (e.g. when coordinating with other GC components), possibly for a partial collection over a fraction of the object graph [33].

Indirect instrumentation consists of using indirect mechanisms to detect when a local garbage collection has taken place (e.g. using `finalizer` methods on a dummy object). This technique is portable and is used in our solution.

Acyclic DGC: The most inflexible technique to implement a distributed garbage collector is to *modify the communication protocol*, or impose the use of a specific one provided by a non-standard system [25–33] (e.g. Thor [34, 36]). Alternatively, intrusion may be confined to *modifying remoting mechanisms* and its code [31, 37 and 38]. If it is possible and allowed, DGC may be implemented resorting to *interception of library loading* performed by the dynamic linker, either by extending or overriding the functionality of components regarding communication and remote method invocation, without modifying code [35]. Portable techniques include *extended communication mechanism*, resorting to extensions allowed by the runtime, such as custom sockets.

Finally, even non-intrusive extensions may be independent of the communication protocol and restricted to *extended remoting mechanisms*, such as sink chain extensions (as our solution provides).

Cycles detection: Some solutions, depending on the adopted algorithm(s) may require additional *direct intrusion* in the runtime, for the purpose of cycles detection, without possibility of delaying the disruptive operations. Examples include suspending the *mutator* (application) while performing bit-colour propagation [33], and applying barriers to interprocess invocations when back-tracing information is being calculated [34].

In general, most solutions also require information of the *local root set* of each process to differentiate objects targeted by local references or just by interprocess references. This may be achieved by modifying the LGC or indirectly via hints provided by the programmer. This is required because existing runtimes neither inform about different levels of reachability, nor provide reflection services with information about stack variables.

2.2 Coupling of GC components

Coupling is defined as the degree of interdependency among different GC components (namely, LGC, acyclic DGC and cycle detection), in the sense that the adoption of one approach for one component will mandate the adoption of the same or related approach to one, or both the others. In essence, this assesses how monolithic a GC approach is, or how it may be flexibly combined with others. This will determine the difficulty of deploying the solution when modifications to the runtime (namely its LGC) are not an option. Furthermore, this may hinder application performance and/or delay garbage reclamation since garbage of the three kinds (i.e. local, distributed acyclic and distributed cyclic) is not created at similar rates, and thus should be addressed with specialised approaches.

One-size-fits-all: The most inflexible solutions are those that mandate the use of the *same algorithm*, a specific one for all three GC components [27, 29–31], that is, the use of a acyclic DGC algorithm, or cycle detector, effectively mandates the use of the same algorithm for LGC purposes. Naturally, this seriously undermines the adoption of these algorithms to an existing runtime, if one of the components cannot be modified or extended.

LGC and Acyclic DGC: Some solutions demand strong integration of the components that perform LGC and acyclic DGC. They may require the LGC to *propagate information*, through the object graph, received by the acyclic DGC component, namely marks [32] and timestamps [29–31], or otherwise provide interprocess *reachability information* of objects to the DGC [36].

Acyclic DGC and cycle detection: There are solutions that, while avoiding intrusive modifications to the LGC of an existing runtime, use the *same algorithm* for acyclic and cyclic DGC [27, 29–31, 36]. This is not as prejudicial as with the case of LGC, but it may prevent the use of a cycle detector if it imposes changes to an existing acyclic DGC algorithm (e.g. reference-listing) integrated in the runtime. Furthermore, using the same algorithm may delay the identification of acyclic garbage that should be performed more frequently (e.g. [27, 29, 30]). At an intermediate level, the DGC must be able to *cooperate* with the cycle detector for example, performing simulated deletions [28].

Other solutions use *specialised cycle detectors* that do not interfere with normal, more frequent, acyclic DGC operation, namely [26, 32–35, 37, 38].

LGC and specialised cycle detection: The coupling between LGC and cycle detection in the context of solutions that use the same algorithm for acyclic and cyclic DGC was already addressed. With respect to solutions with specialised cycle detectors, those based on migration techniques must be able to *detach objects* from the local graph and create the appropriate interprocess references to preserve their reachability [26, 27]. Trial deletion for cycle detection requires the LGC to provide *tentative reachability information* about the outcome of simulated deletions [28]. Cycle detection with group-merger [33] requires the LGC to *propagate information* throughout the object graph in a process, namely reachability bits (colours, red and green) of ongoing cycle detections. Cycle detectors that need to be informed about local root-sets do not necessarily preclude the use of the runtime built-in LGC [32–35, 37, 38].

In this section, we have analysed the virtues and shortcomings of the most relevant GC-solutions found in the literature, bearing portability in mind, that is, w.r.t. runtime intrusion and coupling among their components. The GC-solution described in this paper offers an increased degree of portability (because of the absence of runtime intrusion and full component decoupling) and can therefore be realistically deployed. Similar concerns (portability) can be found in the implementation (despite some runtime intrusion) of distributed back-tracing cycle detector for CORBA objects [35].

3 System model

We consider systems that offer a distributed memory model based on the work done for the Orca language [39] and further refined for the Modula-3 [24] and Java [40] languages. These systems are commonly known as distributed object systems.

In the considered distributed memory model, the global address space (global space) is partitioned into processes with disjoint address spaces. Each individual process is composed of objects and those that are globally accessible are named remote objects. For an object to be globally accessible it must have a global identifier, usually constructed using both the hosting process identifier and the object local identifier.

Communication between processes is modelled as remote object method calls, or simply remote calls. In each remote call two processes are involved, the caller and called object hosting processes. The former is named client process and the latter server process. These roles are established in a per remote method call basis (i.e. a given process can act both as client and server). For a remote call to be made, the client process must hold a reference to the remote object. This reference contains the remote object global identifier and is named remote reference or interprocess reference.

Furthermore, remote objects are passive (they do not have an internal thread of execution), not persistent (their identity does not survive termination of hosting process) and not mobile (they remain in the same hosting process for all their lifetime). Applications are comprised of threads that invoke both local and remote objects.

For memory-recycling purposes, objects are considered live if they are reachable (through graph traversal) from the running program root set. Otherwise, they are garbage (also called dead). A garbage collector function is to reclaim memory associated with garbage objects while preserving live ones. The root set is composed of all global variables and the local variables (residing at the activation stack and registers) of all existing threads of execution. Once an object is marked as garbage, it will always stay that way, meaning, unreachable objects cannot become reachable.

Finally, garbage collection algorithms are classified as safe if all live objects are preserved and complete if all garbage objects are recycled.

4 Distributed garbage collection

Fig. 1 presents an overview of distributed cycle detection in an example situation, which comprises four processes running a distributed application. We assume there is a preexisting acyclic DGC algorithm deployed (e.g. reference-listing as the one described later in Section 4.1.3) and, thus, each process already has an acyclic DGC component running.

Each process stores its DGC structures and, periodically, sends information about them to other processes (e.g. `NewSetStubs` messages). Using these messages, processes cooperate, by pairwise interaction, to detect acyclic distributed garbage. This pairwise interaction need not be two-sided, that is, a process $P1$ that sends DGC messages to another process $P2$, may or may not receive DGC messages from it.

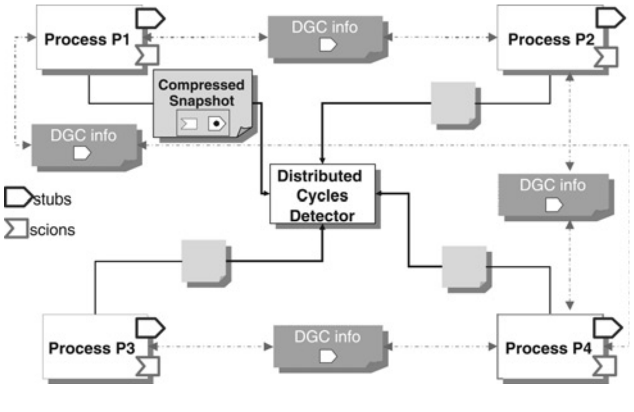


Fig. 1 *Integration of the distributed garbage cycles detector with the reference-listing algorithm*

Since this mechanism based on reference-listing is not complete, there is a distributed cycles detector (DCD), for cyclic DGC. Each process occasionally sends to the DCD a conceptual representation of its enclosed object graph. This takes the shape of a compressed snapshot of the process. A snapshot is a memory-efficient representation of the object graph of a given process (more details in Section 6.3.3). This snapshot also includes information about DGC structures, extended in a way described in the next section.

With this information, the DCD performs the construction of a DGC-consistent cut, upon which it executes a conservative mark-and-sweep (CMS). After the CMS, the DCD is able to identify DGC structures (i.e. scions) keeping cyclic garbage from being collected and to instruct their deletion.

After the DCD receives a compressed snapshot from a process yet unknown to it (thus widening the scope of detection), or an updated version of a compressed snapshot it already holds, it can construct a new DGC-consistent cut and perform CMS on it. The DCD discards older versions of compressed snapshots from each process, no longer in use, when it receives a more recent one. If it receives a message carrying a snapshot from a process that is older than what it already holds, it simply ignores it. Thus, incorporating either newer, or not yet known snapshots, always improves the recency and scope of the DGC-consistent cut, therefore allowing the algorithm to progress, and possibly detecting more distributed cycles.

Individual processes are assumed to have a local garbage collector (LGC) that is responsible for reclaiming dead objects. Each process also contains a DGC component that collaborates with the LGC to prevent reclamation of objects that are only reachable from remote processes (through interprocess references). In practice, this collaboration takes place through local root set extension, meaning, the DGC simply prevents reclamation of an object by referencing it explicitly, leaving the existing LGC unmodified. With this approach, the DGC component function is to detect distributed garbage and update the root set extension accordingly.

The presented DGC algorithm is classified as hybrid, since it combines a reference-listing [24, 41] algorithm for acyclic distributed garbage detection and, to ensure completeness, an extended innovative design and portable implementation of a specialised detector of distributed garbage cycles [37].

4.1 Algorithm for acyclic DGC

In the remainder of this section we present the data structures maintained by the algorithm and the main conceptual aspects of its operation, regarding acyclic distributed garbage detection.

4.1.1 Inter-process reference representation: For distributed garbage detection purposes, interprocess references are represented as stub-scion pairs. The scion resides at the server process and represents an incoming interprocess reference. Each scion points to its corresponding locally hosted remote object. The *stub* resides at the client process and represents an outgoing interprocess reference. Each stub is associated with exactly one scion. A stub–scion pair represents all interprocess references that exist between a given client process and a particular referred remote object.

Stubs and scions are grouped in sets. All stubs in a set have their corresponding scions in the same server process. Conversely, all scions in a set have their corresponding stubs in the same referring process. Each process contains a table of stub sets and a table of scion sets. For simplicity, in the remainder of this document, the former will be called stub table and the latter scion table. The number of sets in each table is determined by the number of referred and referencing processes, respectively.

4.1.2 Local root set extension: To prevent locally hosted remote objects from being reclaimed by the LGC when exclusively reachable through interprocess references, the considered local root set must be extended. This is the purpose of the scion table.

The stub table is a conservative estimate of all remote references held by its enclosing process (outgoing interprocess references). The scion table is a conservative estimate of all existing incoming interprocess references and explicitly references locally hosted remote objects, as long as they are reachable from other processes. Conservative estimations are used to ensure safety, that is, an object is considered live until found otherwise.

The main function of both components (acyclic and cyclic) of the DGC algorithm is to update the previously described data structures, leaving actual object reclamation to the existing LGCs.

4.1.3 Acyclic collector: The creation of scions and stubs is performed incrementally, according to interprocess reference creation. This occurs whenever an application message bearing a remote object reference is exchanged between processes. In this case, the sender process is said to export the reference, and the receiver is said to import it.

Whenever a reference to a remote object located in a given process is exported, the corresponding scion must be created. Every time a reference to a remote object is imported into another process, the corresponding stub must be created. To accomplish this, all application messages must be scanned in search of remote references. This operation is critical since it introduces an additional overhead to reference export and import. If performed carelessly, it would impose application delay (more details on how this issue is addressed are presented in Section 6.3).

Besides the creation of remote references, another potential result of application execution is that of remote objects becoming unreachable. As a consequence, stub and scion tables must be updated to allow the reclamation of such objects.

DGC Protocol: When a LGC execution cycle ends, the stub table is updated according to the outgoing remote references that were dropped. The resulting table contains the stubs corresponding to the remote references that are still reachable from the local root set. The details of how these tasks are performed (i.e. detecting the termination of each LGC execution and determining which stubs should maintained or removed) are explained in Section 6.

Changes in the stub table trigger the creation of `NewSetStubs` messages. The number of such messages is determined by the number of stub sets that were modified (stubs removed) as a result of the LGC cycle execution. Each message contains information related to the surviving stubs on that set. Once created, these messages are sent to the server processes from which the corresponding references have been previously imported.

Upon reception of a `NewSetStubs` message, the DGC component at the process matches the carried information against the corresponding scion set. Scions for which a corresponding stub is not included in the message are removed from the set. Remote objects are recycled when all its scions are removed from the scion table. Naturally, objects are only recycled if they are also unreachable from the non-extended local root set.

If the execution of the previously described tasks implied application suspension, the application performance penalties would be prohibitive. For this reason, the algorithm was designed to enable deferred execution of the tasks related to the DGC protocol, while maintaining its safety. To eliminate race conditions resulting from protocol tasks (e.g. `NewSetStubs` messages in transit) and application work being done concurrently, upon creation, scions are time-stamped using a per-process monotonic global counter, as described in [29, 42]. Additionally, each process maintains a time-stamp vector, or vector clock [43], containing one entry for each referred process. Each entry has the highest known scion time-stamp value for the corresponding process. When a `NewSetStubs` message is sent to a given process, its corresponding entry in the vector clock is also sent.

Algorithm safety is maintained by associating the view of outgoing interprocess references with the time it was taken, ensuring its consistency. This prevents the receiving process from incorrectly eliminating scions whose corresponding stubs were not yet created when the `NewSetStubs` message was generated (e.g. a `NewSetStubs` message is generated when the reply of an ongoing remote method call has not yet been received).

The previously described protocol does not require global synchronisation. This is a consequence of scion tables being updated incrementally, according to the reception of `NewSetStubs` messages. It is also fault tolerant, since message loss only leads to delays in garbage detection, not compromising the algorithm properties (safety and completeness).

4.2 Algorithm for cyclic DGC

In this section, we describe the algorithm responsible for creating and processing DGC-consistent cuts to detect distributed cycles of garbage.

4.2.1 Data structures: The DCD uses data structures already present in the acyclic DGC algorithm and defines new ones that are exclusive to cycle detection. The data structures managed by the algorithm are the following (scions and stubs are mentioned for completeness):

- Scion: represents an incoming interprocess reference. It includes a time-stamp. This is a numeric value provided by a monotonic counter global to the enclosing process, when the scion is created.
- Stub: represents an out-going interprocess reference. It also includes a time-stamp field that is equal to the time-stamp of its corresponding scion.
- Vector-clock: each process maintains a record of the highest time-stamp associated to scions from other processes it knows of, including itself, thus constituting a vector-clock [43].
- Snapshot: representation of the object graph of a process, including its stubs, scions and the vector-clock maintained by the process.

- DGC-Consistent cut: a conservative juxtaposition of snapshots taken at uncoordinated times, comprised of, at most, one snapshot concerning each process.

Scions, stubs, vector-clocks and snapshots are created and maintained at application processes. DGC-consistent cuts exist exclusively in the context of the DCD.

The extension of stubs and scions just described is necessary for the purpose of DGC-consistent cut creation. Nevertheless, it is also present in previous approaches to reference-listing [44, 45], in `NewSetStubs` messages, to avoid race conditions, and can thus be leveraged. `NewSetStubs` messages are time-stamped, with the highest scion time-stamp that the sender process was aware of, when the set of stubs was generated (recall Section 4.1).

4.2.2 Messages: The algorithm defines two types of messages: one that is informative and another one that is operative:

- `NewSnapshot`: message sent from the cyclic DGC component of an application process, to the DCD, carrying a snapshot. Actually, there may be more than one DCD process running. This and other optimisations are described afterwards. It can be sent lazily after a new, updated snapshot becomes available in the process.
- `DeleteScion`: message sent from a DCD to the cyclic DGC component of a process, instructing it to delete a specific scion that is found to belong to cyclic garbage.

Upon receiving a `NewSnapshot` message from a process, the DCD is able to infer, from the sender identifier and the enclosed vector clock, whether there was a snapshot from that process previously available. If not, it determines if the one received is indeed more recent than the previous one. This verification ensures correct handling of possible message reordering.

Upon receiving a `DeleteScion` message, a process removes the scion indicated. Once a distributed cycle of garbage is broken this way, the process may still receive `NewSetStubs` messages from the process where the corresponding stub resides, still containing such stub. This is because it may take several iterations of acyclic DGC to reclaim the rest of the broken cycle. Nonetheless, this does not hinder correctness. Since the scion no longer exists, it will not be recreated (as no more references to that object will be created because it is garbage), and the stub enclosed in the message is simply ignored (as it is would normally be by the reference-listing DGC component, because it has no corresponding scion to maintain).

4.2.3 Cyclic collector: Cycle detection is performed by the DCD. It is divided in four distinct phases: (i) snapshot reception, (ii) DGC-consistent cut creation, (iii) conservative mark-and-sweep (CMS) and (iv) sending of `DeleteScion` messages.

Snapshot reception: The DCD is always ready to receive snapshots from processes. When a snapshot is received, it is stored as a new version concerning the sender process. Therefore the DCD may perform snapshot reception concurrently with another phase.

Thus, reception of a newer snapshot from a process never causes any of the other phases to stop, if they are already in progress. Naturally, when the DCD is otherwise idle, the reception of a new snapshot may trigger phase (ii). When a snapshot from a process is no longer involved in phases (ii) nor in (iii), it can be deleted by the DCD if there is a newer version available.

DGC-consistent cut creation: This is the crucial and most novel aspect of the algorithm. A DGC-consistent cut is created by the DCD by assembling a set of snapshots received from application processes. Note that a DGC-consistent cut need not contain a snapshot from all processes. This may affect the scope of detection but not its correctness. Cycles comprising objects in such missing processes are not represented entirely.

The difficulty in creating DGC-consistent cuts stems from the two following facts: (i) snapshots from some processes may not be available and (ii) snapshots available have been created at uncoordinated times.

A DGC-consistent cut is created without requiring a distributed consensus [46] among the application processes that send their graph descriptions to the DCD. It can be used for cycle detection, even if its global view of the distributed object graph is made of local graph descriptions (sent by the application processes) gathered at different and uncoordinated moments. A DGC-consistent cut, although not required to be causally consistent, is still consistent for GC purposes.

In Fig. 2, on the left-hand side, we show in bold a cut that is not consistent for typical DGC purposes; it results from the uncoordinated creation and sending of snapshots from each application process to the DCD. It is clear that this cut is such that the creation of stubs and scions is not consistent for DGC purposes; in addition, this cut does not correspond to a Lamport's consistent cut. Object graphs received by the DCD provide a view of the global graph that does not correspond to a real one; the differences are because of the shaded stubs and scions. LGCs are not represented as they can occur at any time.

However, based on such graphs, the DCD builds a DGC-consistent cut that allows it to detect distributed

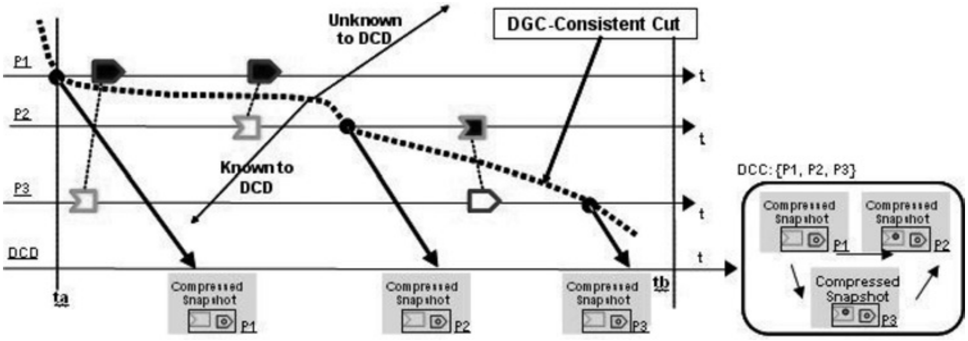


Fig. 2 *DGC-consistent cut and process snapshots as seen by the DCD*

cycles of garbage safely. This cut is consistent w.r.t. the finding of such cycles. The line in bold represents the DGC-consistent cut. The right-hand side of Fig. 2 shows the global graph as perceived by the DCD, based on the graph descriptions received thus far; shaded stubs and scions do not exist in the cut.

Thus, a DGC-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 provide a safe view of the distributed graph as long as the rules to define the root-set of the CMS (performed by the DCD) are respected. In particular, these rules specify which scions are members of the root-set of the CMS, as explained next when CMS is described in detail.

Smaller cuts comprising groups of processes that update their snapshots more often may be created to increase detection promptness. Larger cuts spanning all processes that send snapshots to the DCD will ensure completeness, but need not be created frequently.

When a snapshot is included in a new DGC-consistent cut, it is not copied but just referenced, since the DCD never modifies them. It can thus be shared by several cuts. Once a snapshot is included, newer versions of the snapshot will not be considered by ongoing creations. This ensures liveness even in the improbable situation of a continuous flow of snapshot reception from several processes.

Once a DGC-consistent cut is created, it can be forwarded to the next stage, that is, phase (iii), or stored to be later combined with other DGC-consistent cuts (more details in the remaining of this section when optimisations are discussed).

Conservative Mark-and-Sweep: This phase is performed after a new DGC-consistent cut has been created. Conservative mark and sweep is performed exclusively within the DCD, on the DGC-consistent cut. There is no exchange of messages with other processes, during this phase. The marking is performed on a separate bit-map that includes one bit per entry in the DGC-consistent cut. This way, snapshots are untouched and may be referenced by multiple cuts.

To be safe, CMS must take into account possible, and highly probable discrepancies, among the moments when the snapshots of the different processes are taken and included in the DGC-consistent cut. There is no coordination among them w.r.t. to this activity, as they take snapshots at independent times. Thus, the CMS uses an extended set of global roots that, conservatively, includes:

1. Objects that, in each application process, are directly reachable from the GC local roots (stack and so on) must be obviously considered roots of the CMS.
2. Scions whose corresponding stubs are included in processes whose snapshot is not included in the DGC-consistent cut (it may even be unavailable in the DCD). These scions are members of the CMS root for safety reasons. As a matter of fact, such scions may no longer have a corresponding stub, and be removed after reception of subsequent `NewSetStubs` messages. Nonetheless, the DCD cannot know that for sure, using therefore a conservative approach.
3. Scions with time-stamp greater than the highest time-stamp (regarding the process where the targeted object resides) known by the process holding the corresponding stub. This is also a conservative approach. These scions are those whose corresponding stubs have not yet been created when the referring process recorded its snapshot. These scions verify the following condition.

$$\text{scion.timestamp} > \mathrm{VC}_{P_{\text{stub}}}[P_{\text{scion}}]$$

where:
4. scion.timestamp is the time-stamp given to the scion when it was created,
5. $\mathrm{VC}_{P_{\text{stub}}}$ is the vector-clock maintained by the process holding the corresponding stub, and
6. P_{scion} is the identifier of the process holding the scion.

Note that with the above-mentioned rules, the DCD is able to leverage causality information included in process snapshots to make the cut consistent w.r.t. DGC purposes, even though it is not consistent following Lamport's definition [47].

Thus, CMS is performed starting with the extended root-set, and tracing inter- (from stubs to scions) and intraprocess (from scions to stubs) references. Intraprocess references are expectably more frequent, thus they are subject to an optimised representation (described later). Naturally, tracing an interprocess reference across snapshots from two processes can only be performed if the corresponding stub–scion pair exists in both of them, in the DGC-consistent cut. As the end result of the CMS, scions and stubs belonging to garbage are not marked.

Next, the DCD determines if unmarked scions and stubs belong to distributed cycles and, for each cycle detected, one of the comprised scions is selected and marked for explicit deletion. 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.

Those distributed garbage cycles that already existed when the earliest graph description (included in the DGC-consistent cut being processed) was created, and are totally included in the graph descriptions available at the DCD, are effectively detected and reclaimed. Thus, considering Fig. 2, all cycles that existed before `tb` that are totally enclosed in processes P1, P2, and P3 are detected by the DCD.

Sending of `DeleteScion` *messages*: Once a scion has been selected for explicit deletion, a `DeleteScion` message will be sent to the enclosing process. This message can be sent at any time, without any race condition, because the scion regards a reference that can no longer be transversed by the application, because garbage being a stable property, that is, once garbage, an object can never become live again. Only one message is needed to break each cycle, as the acyclic collector will then be able to reclaim all remaining objects belonging to the cycle. Nonetheless, sending `DeleteScion` messages regarding several scions comprised in the same distributed cycle, will definitely accelerate reclamation of all the objects comprised in the cycle. Messages regarding the same process may be queued and sent in batch.

During the reclamation of the remaining acyclic garbage, after a scion has been deleted, it may happen that its corresponding stub is still included in ulterior `NewSetStubs` messages. According to reference-listing, this does not cause the re-creation of the scion (as this can only happen via exporting a reference). Thus, the stub is ignored and will eventually disappear as the remaining acyclic garbage is reclaimed.

W.r.t each distributed cycle found, the number of `DeleteScion` messages sent only influences the bandwidth used and the speed of cycle reclamation. `DeleteScion` messages that were sent and acknowledged, are recorded, using a best-effort approach, to prevent issuing multiple messages regarding the same scion. This may happen when the same cycle is detected in more than one DGC-consistent cut.

4.2.4 Optimisations: The algorithm presented thus far is subject to three optimisations that are described in this section: (i) snapshot compression, (ii) use of multiple DCDs in parallel and (iii) hierarchical composition of DCDs. These optimisations do not affect algorithm safety and improve both its performance and scalability.

Snapshot Compression: Snapshots containing object graphs of processes, along with DGC information, may be very large, possibly amounting to tenths or hundreds of mega bytes. Sending these snapshots to the DCD process would consume network bandwidth heavily and crumble application performance and communication with other application processes. Furthermore, once received at the DCD process, the cumulative size of all the snapshots would occupy a large amount of memory and disk. Since the DCD would perform the CMS over these large snapshots, cycle detection would become a CPU-intensive operation, slowing it down drastically.

These problems are solved by summarising the object graph (a snapshot) of each application process, prior to sending it to the DCD. This is done in such a way that, from the point of view of the DCD, there is no loss of relevant information. This summarisation compresses a snapshot of an application graph into a set of scions and stubs, with their corresponding reachability associations. As a matter of fact, neither references among objects that are strictly internal to a process, nor object scalar contents, are relevant for the DCD; thus, they are not explicitly represented in the compressed snapshot. In the context of a compressed snapshot, a process may be regarded simply as a set of scions, with each one, possibly referencing one or more stubs. Additionally, each stub is marked if it is reachable from the GC local roots. This is exemplified in Fig. 3.

The previous paragraph contains a simplification that will be used in the remaining of the chapter. For clarity, when we say stubs that are reachable from a scion, or from the GC local-roots, we actually mean: stubs accounting for outgoing references enclosed in objects, that are reachable from a specific object, targeted by an incoming remote reference (this one, represented by a scion), or by a GC local-root.

This summarisation is performed on every snapshot, locally to the process that generated the snapshot. The compressed snapshot is then ready to be sent to the DCD process. Thus, although processes can take snapshots by serialising local graphs, these uncompressed versions never leave the process. The DCD only uses them in their compressed form, that is, after graph summarisation.

Snapshot compression reduces bandwidth usage severely as no actual object content is transferred. The cost of transferring a snapshot is then proportional to the number of scions and stubs in a process, and independent of the actual number and/or cumulative size of the objects located in a process. Furthermore, it also minimises complexity and memory usage at the DCD process. This allows the DCD to detect distributed cycles faster, and manage more snapshots, as opposed to what it would be able to, without snapshot compression. This allows the detection of larger distributed cycles, spanning larger numbers of processes, and comprising larger numbers of objects. Once compressed, a snapshot may be sent to several DCDs.

Multiple DCDs: Distributed cycle detection, for the purpose of cyclic garbage collection, is performed by the DCD, in a centralised manner. Notwithstanding, nothing prevents there being multiple DCD processes running at the same time, on different machines. A DCD may also be co-located in the same machine where other application processes are running, though it runs within its exclusive address space. It need not be a dedicated machine. These issues do not affect the correctness of the solution described. They are only relevant w.r.t performance and scalability.

Hierarchical DCDs: The DCD algorithm achieves completeness and scalability w.r.t. size of distributed cycles, using an hierarchical approach. Without it, a single DCD is clearly not able to manage snapshots from an unbounded

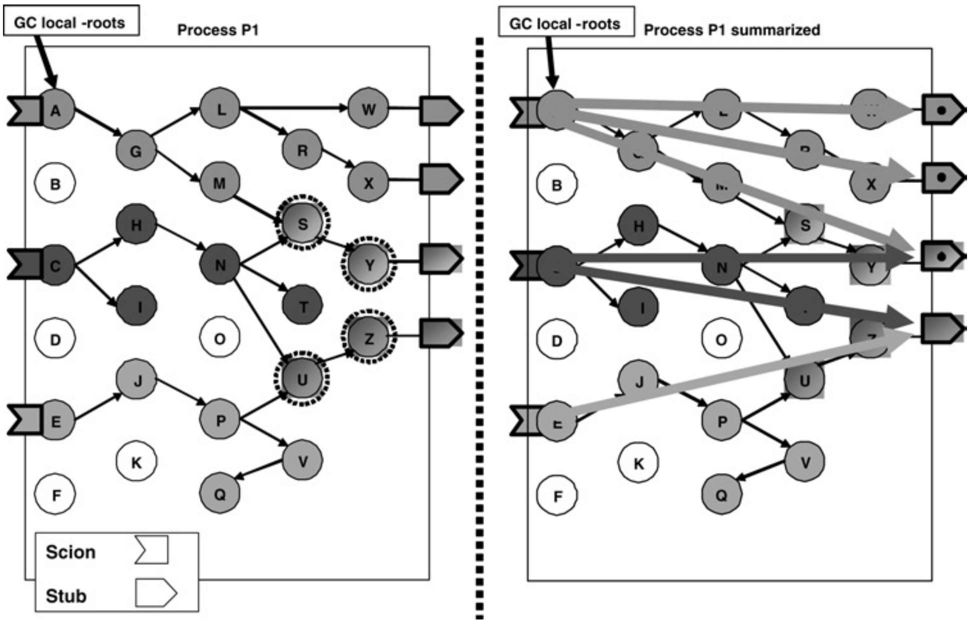


Fig. 3 *Summarisation of an object graph for snapshot compression*

Objects reachable from more than on scion (i.e. *S, U, Y, Z*) are highlighted number of processes. This, at least at a theoretical level, would limit DCD completeness, for example a distributed cycle spanning all processes would not be detectable, because the comprising snapshots would not fit in a single DCD process for CMS.

To circumvent this limitation, every DCD process, besides detecting distributed cycles, is also able to produce a combined, higher-level snapshot (from a DGC-consistent cut), conservatively compressing all snapshots into one that, for DGC purposes, represents all processes as a single one. Every interprocess reference (stubs and scions) involving processes whose snapshots are not available to the DCD is maintained in the combined snapshot. However, interprocess references comprised in the snapshots included in the DGC-consistent cut can be regarded as internal references of the combined snapshot, and can therefore be summarised, as if they belonged to a single process. This is different from the group-based approach described in [32] that requires a single top-level group, containing all existing processes, and tracing all reachable objects individually.

Hierarchical snapshots bound the growth of snapshot size, at the cost of only being able to detect those higher-level distributed cycles that span enclosed snapshots. Thus, distributed cycles fully comprised within a combined snapshot are not detectable at a higher-level DCD. Nonetheless, they are detectable at the lower-level DCD, where the combined snapshot was first created from an existing DGC-consistent cut.

4.3 Prototypical example

In this section we further describe the algorithm operation with respect to prototypical examples. This example portrays how the algorithm operates when information from some process(es) is either absent or outdated, and finally when this information is sufficiently recent to detect a distributed cycle of garbage.

Objects are represented by their name (a letter) and their enclosing process (e.g. A_{P1}). Data structures have the process where they reside mentioned last (e.g. $Stub(F_{P2})_{P1}$, for the stub in $P1$).

Subgraphs of connected objects may be represented in abbreviation (e.g. {{A, C, B}$_{P1}$, {F, G, H}$_{P2}$}), aggregated by its/their enclosing process. References may be also explicitly described when relevant (e.g. $B_{P1} \rightarrow F_{P2}$).

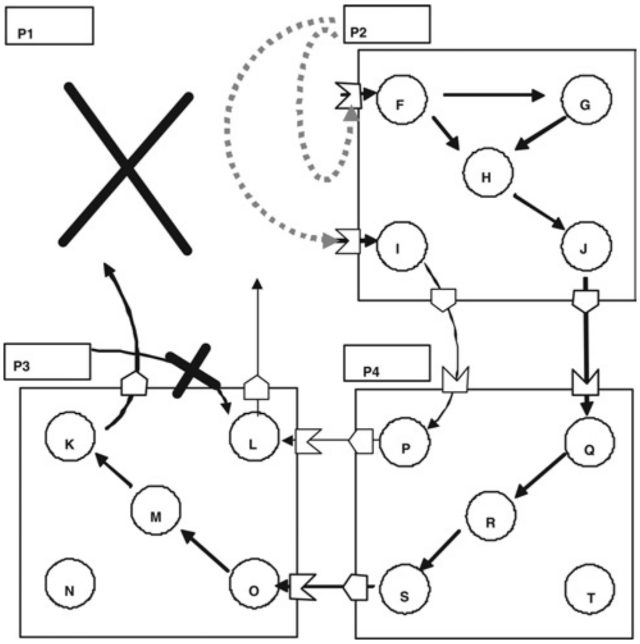


Fig. 4 *Initial situation: no snapshot has been received from process P1*

We now present an example in which the DCD has access to partial or outdated information, about one of the processes holding objects comprised in distributed cyclic garbage. In the initial situation portrayed in Fig. 4, the DCD has no information from $P1$ available. It constructs a DGC-consistent cut with snapshots from processes $P2 \ldots P4$. Thus, even though there are no GC local roots in any of the processes (since the mutator in $P3$ has removed the one referencing object L_{P3}), during the CMS phase, $Scion(F_{P2})_{P2}$ and $Scion(I_{P2})P2$ are promoted to the root-set of the DGC-consistent cut.

Scion promotion is performed according to the safety rules presented earlier, because the scions are targeted by stubs contained in $P1$, of which the DCD knows nothing about. Thus, conservatively, it must consider those scions as roots to ensure safety. This is represented by the dashed grey arrows.

When the DCD performs CMS on the DGC-consistent cut, all objects become marked since they all are reachable from F_{P2} or I_{P2}. Thus, no cyclic garbage is detected, and no `DeleteScion` messages are issued.

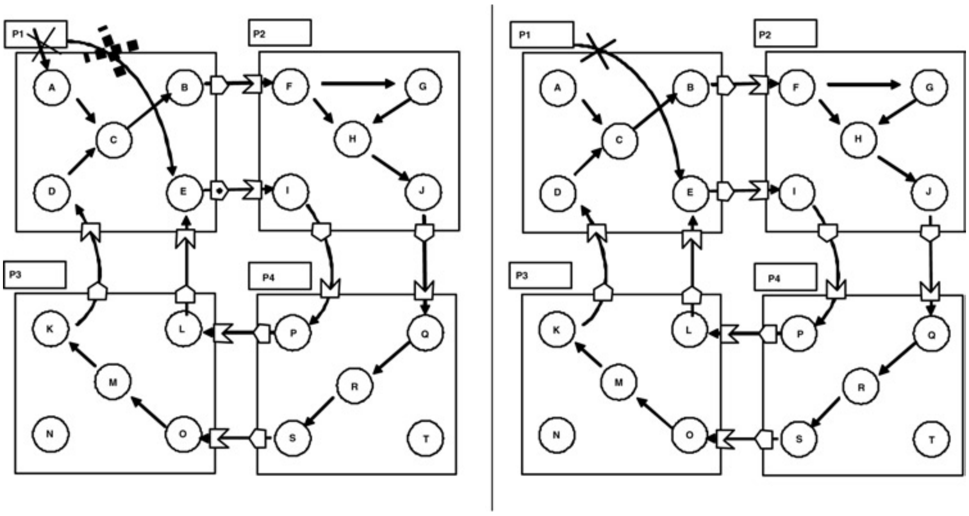


Fig. 5 *Steps (ii) and (iii): the DCD receives increasingly updated information from process P1*

After a period of time, the DCD eventually receives a snapshot from $P1$ (Fig. 5), with the following information:

- $Scion(D_{P1})_{P1} \Rightarrow \{StubsFrom \equiv \{F_{P2}\}\}$
- $Scion(E_{P1}\ )_{P1} \Rightarrow \{StubsFrom \equiv \{I_{P2}\}\}$
- $Stub(F_{P2})_{P1} \Rightarrow \{Local.Reach \equiv false\}$, since object A_{P1} is no longer referenced from the GC local roots in $P1$ (the reference has been deleted by the mutator before the snapshot was taken).
- $Stub(I_{P2})_{P1} \Rightarrow \{Local.Reach \equiv \text{true}\}$, since the other reference from the GC local roots in $P1$, targeting object E_{P1}, has been deleted by the mutator only after the snapshot in $P1$ was taken, as it is indicated by the dashed cross.

The DCD is then able to create a new DGC-consistent cut containing snapshots from processes $P1\ldots P4$. Its root-set includes only $Stub(I_{P2})_{P1}$, marked reachable locally, in the snapshot from $P1$.

After performing the CMS, all objects belonging to the inner cycle $\{E_{P1}, I_{P2}, P_{P4}, L_{P3}\}$ are marked, whereas those belonging to the outer cycle are unmarked. This means the DCD is already able to detect the outer cycle but needs more up-to-date information from $P1$ (though it does not know that explicitly) to detect the inner cycle that is also garbage but not detectable yet.

W.r.t. the outer cycle, the DCD can issue `DeleteScion` messages regarding scions identified as garbage. As already mentioned, these messages may be queued and sent in batch lazily. Therefore process $P1$ may take another snapshot after that, which still contains the same scions that were identified as cyclic garbage. Thus, when the DCD receives the new snapshot from $P1$, it is able to identify both cycles, because of the following difference in the snapshot from $P1$.

$Stub(I_{P2})_{P1} \Rightarrow \{Local.Reach \equiv false\}$, since the previous reference from the GC local roots in $P1$ no longer exists and that fact has been registered in its snapshot.

If the DCD has recorded the `DeleteScion` messages issued previously, it will send `DeleteScion` messages only regarding the inner cycle. Once the `DeleteScion` messages are received by the intended processes, and the corresponding scions removed, the acyclic DGC (reference-listing) will be able to reclaim the objects comprised in the two cycles detected.

4.4 Analysis of algorithm properties

In this section, we address the relevant properties of complete distributed garbage collection, discussing them against the algorithm proposed: safety, liveness, completeness, termination, and scalability.

Safety: The DCD does not reclaim objects that may still be reachable to the mutator, because of the safety rules enforced in the CMS. The DCD is resilient and conservative w.r.t. loss and delay of `NewSnapshot` messages. Replayed `NewSnapshot` messages cause no error as they are idempotent. Nonetheless, for simplicity, they are ignored. Messages sent earlier, and received out of order, are discarded; if considered, they would temporarily prevent algorithm progress as DGC-consistent cuts would go back in time.

Concurrency between cycle detection and the mutator must be analysed w.r.t. two aspects: local and distributed. In local terms, snapshots are taken when the mutator is idle, or created incrementally. Nonetheless, the DCD manipulates snapshots assumed to be coherent w.r.t. each process. Regarding distributed invocations that may swap references to objects among processes, there are no concurrency issues with the DCD; a DGC-consistent cut reveals cyclic garbage that already existed when the enclosed snapshots were created, while the mutator only manipulates live objects, which will be correctly accounted for during CMS.

Liveness: Liveness of the DCD naturally depends on processes updating their snapshots, and the DCD executing CMS on DGC-consistent cuts containing those snapshots recently received. This is guaranteed since DGC-consistent cuts are always created with the most recent version available for each snapshot. Snapshot updates need not be performed often but processes with mutator activity must eventually update their snapshots.

Completeness: To achieve completeness, the DCD must eventually encompass all processes, in the sense that all of them must be included in DGC-consistent cuts. This is achieved using an hierarchical approach, by construction of higher-level DGC-consistent cuts. Thus, all processes are eventually considered, at some level, by a DCD, though not necessarily explicitly included in the snapshots sent to it.

Every snapshot received at each DCD is always included in some DGC-consistent cut, at some level. The DCD ensures that periodically, a top-level DGC-consistent cut is created that directly or indirectly spans all the processes that sent snapshots to the DCD. This higher-level cut may be sent to other DCDs it knows about.

The algorithm does not impose any pattern of cooperation among the participating processes, it is only required that all processes are eventually accounted for, directly or indirectly, in higher-level DGC-consistent cuts that may be exchanged among DCDs. Every DCD can perform at any level, simply by receiving DGC-consistent cuts from other DCDs and creating higher-level cuts, possibly exchanging them with other DCDs as well.

W.r.t. identification of garbage, even in the case where `DeleteScion` messages have not been issued for all scions identified as garbage in one DGC-consistent cut, and individual disjoint cycles have not been specifically identified, the algorithm still achieves completeness eventually. The remaining scions will either be removed by the acyclic DGC (when they belong to the same cycle) or will be explicitly instructed for deletion in ulterior DGC-consistent cuts. This is so because the DCD chooses, in first place, scions for which `DeleteScion` messages have not been issued yet.

Termination: CMS performed on each DCD terminates since it is performed resorting exclusively to information available locally, and the termination of mark-and-sweep algorithm is trivially sound because there is no concurrency with the mutator in the context of the DCD.

Scalability: The discussion of algorithm scalability derives from some of the arguments for completeness. The algorithm can scale to large numbers of processes because it imposes neither synchronisation requirements nor communication among them. There may be multiple DCDs running, possibly cooperating asynchronously in a loose hierarchy. Compressed snapshots are smaller, use limited bandwidth, are only sent occasionally and can be subject to CMS even if stored on disk (as in [48]).

It's worthy to note that with this information, the cycle detector does not perform a full garbage collection (as in Liskov's proposal [36]). Moreover, multiple DCDs are independent and not simply the replicas of a centralised detector.

The deferred nature of the DCD is exclusively used to detect cyclic distributed garbage collection. Snapshot compression and the possibility of multiple DCDs, in parallel as well as in a loose hierarchy, contribute to algorithm scalability.

The hierarchical approach does not require a strict topology (e.g. tree) and any DCD process can perform at any level of the hierarchy if it is contacted by others. There is no single top-level root of the hierarchy.

5 .NET support

This section describes the features of the target system that are fundamental to understand the proposed solution (for more details consider [49]). The target system exposes the distributed memory model described in Section 3. Using .NET remoting terminology, types are categorised as nonremotable or remotable. Only instances of types in the last category are permitted to cross process boundaries. In .NET terminology, a process may be regarded as an application domain. Note that an object is said to cross process boundaries each time it is included in the actual parameters list, or return value, of a remote method call.

Remotable types are further refined into two subcategories: marshal-by-value and marshal-by-reference. The subcategory to which a given type belongs determines its instance behaviour in process boundary crossings. As their names suggest, marshal-by-value instances are passed by value, that is a copy is used. On the other end, marshal-by-reference instances are passed by reference, meaning, the instance global identifier is used.

A type is categorised as marshal-by-reference, if it inherits (directly or indirectly) from `System.MarshalByRefObject`. Global identifiers are represented by instances of `System.Runtime.Remoting.ObjRef`, itself a marshal-by-value type.

For a type to be classified as marshal-by-value it must be annotated with the attribute `Serializable`. This requirement is justified by the mechanism used for copy creation. This mechanism, commonly known as serialisation, produces a byte sequence containing the internal state of all objects included in a given graph.

5.1 Communication infrastructure

Although communication is modelled as remote method calls, it is ultimately achieved through message exchange between client and server processes. When a method call is made, a message composed of the parameters is sent from the client process to the server process (these parameters include object and method identifiers). The method call result is obtained from the received return message.

The previous observation forms the basis of the .NET Remoting communication infrastructure architecture. Communication is supported by a configurable chain of message-processing nodes inspired in the name usually used to refer to computer network participants, generally called message sinks. The chain end-points, transparent proxy and stack builder sink, are responsible for converting between stack frames and messages. The remaining nodes constitute the node set that supports the communication protocol. This node set is named channel.

The transparent proxy function is to represent the server object at the client process. In other words, the client object invokes methods on the transparent proxy which, in turn, delegates to the associated real proxy the conversion of these calls into messages forwarded to the next node in the chain. The method call result is produced based on the received return message.

The stack builder sink represents the client object at the server process. Received messages, passed along the node chain, are converted to local calls on the server object. The return value is converted to a message that is sent back through the chain.

The chain is physically split across participating processes. The half that resides at the server process is constituted upon remote object activation. The other half is created at the client process upon reception of the corresponding `ObjRef` instance. To enable client side chain creation, the received `ObjRef`, sent by the server process, contains all the required information (e.g. location information, remote object identifier).

The previously described architecture offers several customisation opportunities, in particular, through the definition of new channels and new real proxy types.

Lifetime Management: In .NET, remote objects are reclaimed using a lease-based algorithm. The lease value determines if the associated remote object is alive. In each remote call for a given remote object, its lease is automatically renewed. When an object is considered dead (if its lease has expired) the corresponding memory is reclaimed.

The previously described approach is not safe since remote objects may be, erroneously, considered dead. For example, consider a client holding a remote reference and not making use of it for a time interval greater than the lease time. In this scenario, the client process will obtain an error when it tries to use the remote reference (i.e. by calling a method on the referred remote object).

To prevent such errors, the application has the opportunity to register sponsors that will decide about lease renewal. Each process contains a lease manager that tracks leases that are associated with remote objects. Periodically, the lease validity is checked and, if it has expired, registered sponsors are queried about the lease renewal. Although interesting, this solution only shifted the problem of memory reclamation to the application programmer domain.

5.2 AOP support

Kiczales *et al.* [50] argue that object-oriented programming (OOP) failed to deliver the promised clean problem decomposition. In fact, applications tend to be polluted with code snippets not directly related to the problem domain, but instead, related to other concerns, such as system level issues (e.g. security, transaction management, concurrency control). Other non-system concerns include business-related issues, GUI design, pattern integration and so on. A similar observation, with respect to memory management, led to the pro garbage collection argument used by Wilson [2], in which the existence of such a service was considered a fundamental requirement for accomplishing program modularity.

In AOP, system properties are classified as components or aspects. Components contain the implementation of system properties directly related to the problem domain resulting from its functional decomposition. Aspects are the implementation of properties related to system level concerns and are transversal to problem domains. The overall system is the resulting combination. The process of combining components and aspects is named aspect weaving.

Although there are several approaches to aspect weaving, usually performed via code instrumentation (e.g. source code instrumentation), only runtime aspect weaving will be considered in the current document.

The target system in both considered implementations, CLR and Rotor provides support for runtime aspect weaving, performed through method call interception. The provided method call interception infrastructure, described in [51], is based in the .NET Remoting communication infrastructure, described in Section 5.1, and is known as context architecture.

Although this built-in AOP-support and its usage may be regarded as rudimentary (as it only supports method call interception), it is nonetheless useful to our purpose of portability, that is, avoiding modifications to either the virtual machine, application source or byte-code.

Context architecture: In this architecture, processes are subdivided in contexts. A context is the runtime environment that results from the addition of a particular set of aspects to the base system properties. This is performed by extending the call chain, described in Section 5.1, with additional message-processing nodes. These additional nodes are the provided call interception opportunities (also named join points in AOP terminology).

To selectively accomplish the afore-mentioned call chain extension, marshal-by-ref types were further refined into two subcategories: context-agile and context-bound types. Only types in the latter category are considered in the context architecture. A type is classified as context-bound if it inherits (directly or indirectly) from `System.ContextBoundObject`.

The association between component (context-bound type) and aspects (sets of nodes in the call interception chain) is performed by annotating the component with the `Context` attribute. This type of custom attribute (.NET support for metadata extension) is responsible for specifying the required context for the annotated type instances by contributing with the necessary message-processing nodes.

Fig. 6 depicts the chain general composition. It is partitioned into four distinct regions, namely, envoy sinks, client context sinks, server context sinks and object sinks. To help understand the provided interception opportunities, consider the existence of two contexts (client and server) and the propagation in the chain of the message that corresponds to a method call. The sequence of events for the return message is reversed. For simplicity, we assume that each context resides in a separate process.

While still in the client context, two sets of sinks participate in call interception: envoy sinks and client context sinks. Although envoy sinks reside in the client context, they were specified by the server and are target-object-specific. They present an opportunity for collecting information regarding the client, for usage at the server context.

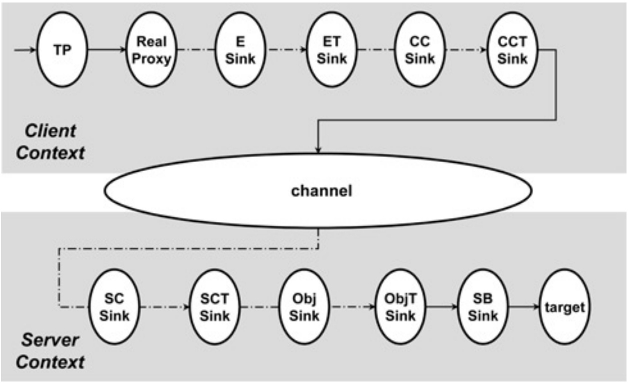


Fig. 6 *.Net remoting context architecture*

The next sinks that are given the opportunity to intercept the call are client context sinks. This set is specified by the context from which the call is originating.

The remaining two sets of sinks will participate in call interception upon arrival at the server context. Both sets are specified by the server and their names, server context sinks and object sinks, suggesting their purpose. The first set is shared by all instances residing in the server context. The second is specific to each instance. As an example, consider the automatic renewal of a remote object lease. In this case, the .NET remoting infrastructure uses an object sink to renew the lease each time the object receives a remote call.

6 Implementation and performance results

In this section we present the implementation details that are fundamental to understand the proposed solution. The implementation follows an architecture based on a clean separation between relevant event detection, required information gathering and the implementation specific to the DGC service. It is composed of two layered modules: (i) the lower module named instrumentation and (ii) the upper module named DGC Service. This layered organisation was adopted to simplify the usage of the DGC algorithm in other implementations of the target distributed object system; the only module that requires modification (if any) is the one named instrumentation. The function of the instrumentation module is to gather the required information for the execution of the acyclic distributed garbage detection.

The functionalities provided by the instrumentation module are (i) detection of remote reference import, for stub table update; (ii) detection of remote reference export, for scion table update; (iii) detection of LGC cycle termination and for stub table update, gathering of information related to unreachable outgoing remote references; (iv) local object graph snapshot creation, for cycles detection.

The DGC service module includes the following elements: (i) an extension of the local root set, to include information about locally hosted objects that are remotely reachable (scion table); (ii) a conservative estimative of locally held remote references (stub table) used to update remote scion tables; (iii) a well known communication endpoint for DGC protocol message exchange.

The previously enumerated functionalities are included in all participating processes. Additionally, one of the participating processes (DCD) is responsible for performing the detection of distributed cycles. This process includes: (i) a well known communication endpoint that receives snapshots from participating processes and (ii) the component responsible for DCD operation.

For the purpose of distributed garbage collection, marshal-by-reference types are classified as dgc types or non-dgc types. Only instances of dgc types are subject to DGC. This classification provides a choice between using the proposed .NET remoting extension, or the default lifetime management service. A type is classified as dgc type if it inherits (directly or indirectly) from `RemoteObject` (`System.Runtime.Remoting.DGC.RemoteObject`).

6.1 DGC service module

Local root set extension is performed by using the default lifetime management service, in particular, by registering a local sponsor. This sponsor is shared by all dgc type instances hosted in the process. When queried about lease renewals, the sponsor responds according to the instance interprocess reachability status, maintained in the scion table.

The conservative estimation of locally held remote references (stub table) is updated upon LGC execution cycle termination. This event triggers the execution of a low-priority background thread that discards stubs related to unreachable outgoing remote references. This is done using weak references, a system-provided mechanism for referencing objects without preventing their reclamation. Each stub holds weak references for all locally held transparent proxies to its associated remote object. Note that, at the client process, and for a given remote object, more than one call chain may exist. The stub is discarded when all its weak references are invalid, meaning, all referred transparent proxies were collected.

The previously described techniques require additional event detection, namely: (i) LGC execution cycle termination, to trigger the low-priority background thread execution; (ii) transparent proxy creation, to refer the created proxy from the corresponding stub, using a weak reference.

The remaining elements in this module are the required well known communication endpoints. They are implemented as stateless remote objects that exist for the entire lifetime process.

6.2 Instrumentation module

The presented solution does not require modifications to the underlying virtual machine. Additionally, this module was used without modifications in two implementations of the CLI, CLR and Rotor.

To detect LGC cycle execution termination, an unreachable singleton instance of type `System.Runtime.Remoting.DGC.LocalGcDetector` (a dummy object) is used. This type redefines the `Finalize` method to generate the event of LGC cycle execution termination. Since the singleton instance is never reachable from the local root set, it is considered garbage. As a consequence, it is eventually collected and its `Finalize` method is called, generating the required event and creating a new unreachable instance.

As described in Section 1, the .NET communication infrastructure provides two extensibility points: channels and real proxies. Although it would be possible to detect remote reference import and export through the definition of a custom channel that would perform the required message scanning, this solution would force the usage (by the application) of that particular communication channel. Alternatively, the DGC service is considered an aspect, and the provided context architecture is used. This approach does not impose a specific communication channel.

The base type for dgc types, named `RemoteObject`, is a `ContextBoundObject` derivate. It is annotated with a context attribute that adds an envoy sink to the call interception chain. Additionally, the context attribute performs proxy creation detection, since it has the opportunity to inject a custom real proxy in the interception call chain.

Although the envoy sink can be used to scan all messages that traverse the call chain, this is not the proposed solution because of the performance penalties that would result from intensive message scanning. Based on the observation that envoy sinks are part of the client side call chain, a conclusion can be drawn; envoy sinks must be an extension of the `ObjRef` instances that represent the remote object. By detecting serialisation and deserealisation of the inserted envoy sink, we can detect remote reference export and import, respectively. Upon serialisation, the envoy sink transports the DGC-required information (e.g. scion identifier). With this solution, detection of remote reference boundary crossings only imposes performance penalties when application messages effectively contain remote references. This is a fundamental characteristic of our implementation since it contributes decisively to the encouraging performance results (see next section).

Finally, and for distributed garbage cycles detection, snapshot creation is performed by specific application request. For that purpose, the application programmer must contribute by registering local application roots with the DGCService (this is necessary because thread stacks are not first-class objects in .NET).

6.3 Performance results

Because of the deferred nature of the implemented DGC algorithm, the majority of the DGC-related tasks are executed in a low-priority background thread to prevent penalising application performance. Nevertheless, it is essential that, eventually, these tasks are given the opportunity to be executed. Failure to comply leads to undetected garbage accumulation.

Note that undetected garbage accumulation is not actually a problem, since its growth (beyond configured limits) can trigger a priority boost of the background thread, forcing its detection and consequent reclamation. Naturally, this has a negative impact (but necessary) on application performance.

The DGC-related tasks that are performance critical are those that impose application pause times (i.e. no application-related work is being done, regardless of the urgency). These pause times occur when DGC bookkeeping is performed in application enrolled threads (i.e. synchronously). Our main performance implementation requirement was the minimisation of these pause times, imposed by the following tasks: (i) remote reference usage, i.e. remote method calls made through the extended call chain; (ii) remote reference import and export detection; iii) local snapshot creation, for cycles detection. For the time being we have focused on measuring the performance penalties imposed by the first two since the latter can be performed when the application is idle. This is a valid approach since distributed garbage cycles are not frequent and thus, the required local snapshot creation can be selectively deferred to minimise application disruption.

In the next subsections, the presented times are the average of 100 samples of the execution time of each sequence of actions to evaluate. Efforts have been made to ensure that no LGC and DGC collections occur while evaluating the costs of each sequence of actions. Although these measurements are presented as absolute times, their single purpose is to underline the working temporal scale. Conclusions were drawn only from the observed variations between measured times with and without DGC-related operations.

6.3.1 Remote reference usage: The goal is to evaluate time overhead imposed by the usage of the extended call chain in remote invocations. Note that an additional node exists in call chains associated with dgc-type instances. Since this additional node is a pass-through, meaning, it just forwards messages to the next node in the chain (it is not actually scanning passing messages), the expected additional cost is low.

In the following scenarios, two processes are used: a client and a server process. The client performs a remote invocation on an object hosted at the server process. The elapsed time is measured at the client, hence it includes round-trip and remote method service times.

Scenario A: This is a worst case scenario. Both processes are hosted in the same computer and the called method does not perform useful work (i.e. empty method body).
Results: non-dgc type = 260.3 μs; dgc type = 310.4 μs; $\Delta = 19.25\%$
Scenario B: Both processes are hosted in the same computer and the called method does perform some simulated work ($\simeq$1 ms long).
Results: non-dgc type = 1021.4 μs; dgc type = 1031.4 μs; $\Delta = 0.98\%$
Scenario C: Each process is hosted in a separate computer and the called method does not perform useful work.
Results: non-dgc type = 6618.8 μs; dgc type = 6719.1 μs; $\Delta = 1.52\%$

With scenarios A and B we intend to show the effects of the inclusion of reality factors (i.e. network latency or useful work execution) in the relative cost associated with the usage of the extended call chain. This cost is masked by each factor introduced. From the previously stated we conclude that, as expected, in real-world applications there is no significant penalty for using the proposed DGC solution.

6.3.2 Remote reference import and export detection: Here, the goal is to measure time overhead associated with the export and import of remote references to a dgc-type instances, as opposed to remote reference export and import to non-dgc-type instances. The measured time includes client and server data structure update times and remote reference propagation.

These performance results represent the cost associated with scanning messages that contain remote references and, as a consequence, updating DGC data structures. Again, the elapsed time is measured at the client end, hence it includes round-trip and remote method service times.

Scenario A: This is a worst case scenario. Both processes are hosted in the same computer and the called method produces a remote reference without performing additional useful work.
Results: non-dgc type = 720.9 μs; dgc type = 1091.5 μs; $\Delta = 51.4\%$
Scenario B: Again, a worst case scenario. Both processes are hosted in the same computer and the called method produces ten remote references without performing additional useful work.
Results: non-dgc type = 5009.9 μs; dgc type = 7013.9 μs; $\Delta = 40\%$
Scenario C: Both processes are hosted in the same computer and the called method, besides producing the remote reference, also performs some simulates work ($\simeq$1 ms long).
Results: non-dgc type = 1161.6 μs; dgc type = 1442 μs; $\Delta = 24.14\%$
Scenario D: Client and server processes are hosted in separate computers. The called method produces a remote reference, without performing useful work. No network payload imposed by the application.
Results: non-dgc type = 9009.2 μs; dgc type = 11331.6 μs; $\Delta = 25.78\%$

Again, the inclusion of reality factors (i.e. network latency, in scenario D, or useful work execution, in scenario C) takes performance penalties to encouraging values, when considering the benefits of the usage of the proposed solution. Note that the measured costs are relative to serialisation and deserialisation of the extended `ObjRef` (with the envoy sink), not to DGC data structure updates. This statement is supported by benchmarks that show costs of data structure update below 100 ns.

Finally, the performance of the evaluated tasks does not depend on the number of scions and stubs in the DGC data structures, since only $O(1)$ operations are used.

6.3.3 Snapshot compression: To evaluate the performance of snapshot compression, we used a synthetic benchmark based on data taken from previous work on memory behaviour in the Java virtual machine. This choice is because of the fact that there is more and earlier work, in this subject, developed in the context of the JVM. Naturally, similar results would apply to .NET, given the similarities between the JVM and the .NET CLR w.r.t. object representation (namely, class and monitor references), and between the Java and C# languages. Thus, memory behaviour of applications should be similar, w.r.t. snapshot compression, in both environments.

The DGC-consistent cuts algorithm requires reachability information among scions and stubs to be stored. The quantities involved, and the calculations performed, to estimate the size of a compressed snapshot, and consequent compression ratio, are described next.

1. *NObj*: number of objects in memory.
2. *NScions*: number of scions (i.e. distinct incoming interprocess references targeting objects in the process).
3. *NStubs*: number of stubs (i.e. distinct objects located in other processes, targeted by interprocess references, contained in objects of the process).
4. $GraphSize = \sum_i^{NObj} size(obj_i)$
5. $ScionSetSize = \sum^{NScions} size(scion)$
6. $StubSetSize = \sum^{NStubs} size(stub)$
7. $size(ScionReach) = size(scion) + (NStubs)/8$ (*since 1 byte = 8 bits*)
8. $size(StubReach) = size(stub) + 1$ *(conservative inclusion of extra-bit for reachability from local-root)*
9. *ScionReachSetSize*

$$= \overbrace{\sum^{NScions} [size(scion) + \lceil (NStubs + 1)/8 \rceil]}$$

10. $StubReachSetSize = \overbrace{\sum^{NStubs} [size(stub) + 1]}$
11. *SnapshotSize = GraphSize + ScionSetSize + StubSetSize*

$$SnapshotSize = \overbrace{\sum_i^{NObj} size(obj_i)} + \overbrace{\sum^{NScions} size(scion)}$$
$$+ \overbrace{\sum^{NStubs} size(stub)}$$

12. *CompressedSize = ScionReachSetSize + StubReachSet Size CompressedSize =*

$$\overbrace{\sum^{NScions} [size(scion) + \lceil (NStubs + 1)/8 \rceil]}$$
$$+ \overbrace{\sum^{NStubs} [size(stub) + 1]}$$

13. $CompressionRatio = \dfrac{SnapshotSize}{CompressedSize}$

The memory behaviour of Java programs has been studied in the literature [52–55], using different (and some times optimised) implementations of the Java virtual machine. In these tests, the average size of the Java heap is 150 MB. Average object size is small in Java (and expectably in .NET as well) as the previous work demonstrates, averaging 30 bytes or less.

However, previous work does not consider character, byte and reference arrays, as private object data. They only consider the cost of storing primitive data and references to other objects. If these indirect costs are attributed to object instances, then the average object size (for most cases) will vary between 36 and 236 bytes [55]. Therefore both 64 and 128 bytes are useful and practical estimations for average object size, occupying respectively, 16 and 32 memory words, 32-bit wide.

As typical heaps in the study have 150 MB in size, this would amount to almost 2.5 million objects of 64 bytes. We assume that the number of objects referenced remotely, in large heaps, will be in the order of thousands; we will consider two boundary scenarios to the ratio between local and remote objects: between 100/1 and 2000/1.

To the best of our knowledge, there are no available measurements w.r.t. the actual density of remote references in application graphs, that is, the fraction of objects involved in remote references when compared with total number of objects. The assumption employed is based on other assumptions from several sources: (i) books on Enterprise Java Beans [56], other discussion fora related to EJB [57], RMI online documentation [58] and RMI debug information available in [59]. All these sources suggest that medium-sized servers contain thousands of objects targeted remote references.

To estimate the size occupied by scions and stubs, consider that a non-optimised representation of a stub or scion, without reachability information, requires at most:

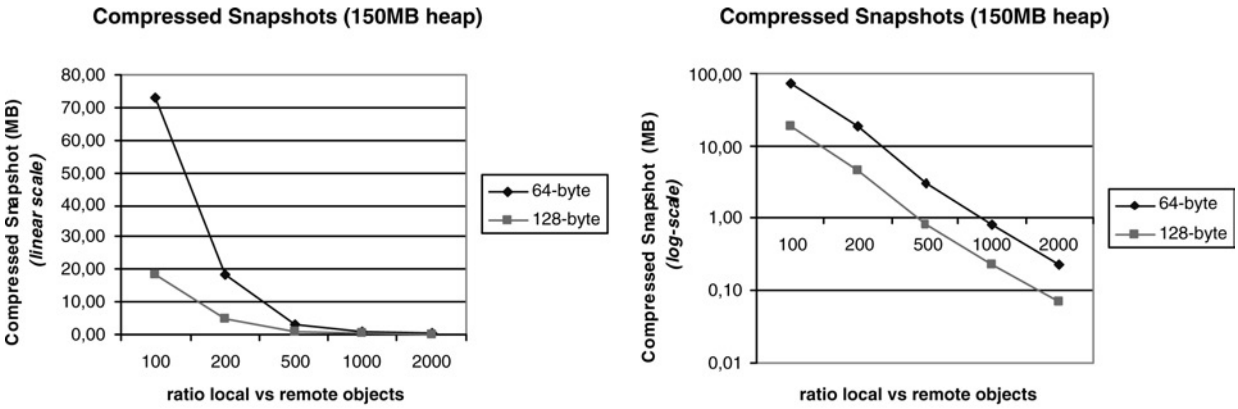
- two 32-bit words for object header (classID, lock, hashCode and so on).
- one word for IP address.
- one word for object ID, within the process.
- one word for invocation counter/time-stamp.

In the compressed snapshot, an additional reachability bit-map is required, whose maximum size in words is $NStubs/32$. Thus, the base size of scions and stubs is 20 bytes, with added reachability information that is dependent on the number of stubs.

We now present the calculation for a synthetic test-case: a 150 MB heap, with average object size of 64 bytes, and a ratio of local to remote objects of 1000 to 1. We assume an equal number of scions and stubs. The results are the following:

1. $NObj = 2\,457\,600$
2. $NScions = 2458$
3. $NStubs = 2458$
4. $GraphSize = 157\,286\,400$ bytes
5. $ScionSetSize = 49\,160$ bytes, with scions and stubs occupying 20 bytes each.
6. $StubSetSize = 49\,160$ bytes
7. $size(ScionReach) = 20 + \lceil(2458 + 1)/8\rceil = 20 + 307 =$ 327 bytes
8. $size(StubReach) = 20 + 1 = 21$ bytes
9. $ScionReachSetSize = 803\,766 = 785$ KB
10. $StubReachSetSize = 51\,618 = 50.4$ KB
11. $SnapshotSize = 157\,286\,400 + 49\,160 + 49\,160 =$ 157,384,720 bytes $= 150.09$ MB
12. $CompressedSize = 803\,766 + 51\,618 = 855\,384$ bytes $= 0.8158$ MB
13. $CompressionRatio = \dfrac{157\,384\,720}{855\,384} = 183.99$ times

Following the previous example, Fig. 7 presents a study of how the size of compressed snapshots varies with changes in average object size, and the ratio of local against remote objects. The graphs show the results contained in the top table, using linear and logarithmic scales. The graphs show that the size of compressed snapshots decreases (and the compression ratio increases) geometrically, with the ratio of local against remote objects. Doubling this ratio produces compressed snapshots with a



Size of Compressed Snapshots (MB)					
Ratio of Local vs Remote objects	100	200	500	1000	2000
64-byte objects	72.96 MB	18.48 MB	3.07 MB	**0.82 MB**	0.23 MB
128-byte objects	18.48 MB	4.74 MB	0.82 MB	0.23 MB	0.07 MB

Compression Ratio					
Ratio of Local vs Remote objects	100	200	500	1000	2000
64-byte objects	2.07	8.14	48.84	**183.99**	656.51
128-byte objects	8.14	31.69	183.99	656.51	2171.24

Fig. 7 *Results of snapshot compression for synthetic 150 MB heaps*

size four times smaller. Conversely, compression ratio increases by a factor of four.

Break-even for 64-byte objects is reached with a ratio of local against remote objects below 70. For 128-byte objects, the break-even is reached with a ratio below 35. However, note that the size of compressed snapshots also varies geometrically with average object size. For a heap with the same size, if the average object size doubles, there will be half the objects. This results in, for the same ratio among local and remote objects, a reduction of stubs and scions to half. Since the size occupied for each scion grows linearly with the number of stubs (while stubs maintain a constant size), the size of the resulting compressed snapshot will be dominated by scion size; thus roughly four times smaller.

When the number of stubs reachable from each scion is 5%, reachability bit-maps are replaced with a vector of indexes. This results in an improvement of 21.25%, thus with a compression factor of 223.10 (instead of 183.99) in the previous example.

Discussion: In the context of actual application execution, the adoption of the DGC solution presented reduces memory usage in participating processes, with consequent performance benefits by detecting and reclaiming distributed garbage. This is achieved while ensuring programming soundness (safety) and preventing memory exhaustion in long-running applications by detecting distributed cycles of garbage (completeness). The performance impact (w.r.t. total time overhead) imposed to distributed applications by the DGC solution proposed can be drawn from the performance results described for managing remote references and snapshot compression. Therefore assuming a ratio of local to remote objects of 1000 to 1, and uniform distribution of invocation of all distributed objects, the overhead presented for remote reference usage, import and export is further mitigated during actual application execution. Moreover, this overhead can be completely masked if distributed invocations exchange larger-sized objects or more complex object graphs, while improving on the built-in lease-based approach that requires communication for periodic renewal of each lease.

7 Conclusions

In this document we presented a solution for extending .NET remoting with a DGC service. The DGC algorithm is characterised as safe, complete (i.e. able to detect all garbage objects including distributed cycles), and scalable to large numbers of objects and processes.

The main contribution is a DGC algorithm design, architecture and implementation based on non-intrusive techniques (i.e. runtime instrumentation), meaning, it does not require modifications neither to the virtual machine, nor to application source or byte-code. To accomplish it, the existing built-in AOP-support in a commercially used runtime (i.e. .NET) is leveraged, allowing immediate adoption.

We find the performance results encouraging, considering the benefits associated with the usage of the resulting programming model. They show the feasibility of the approaches and techniques employed. We highlight (i) the use of method call interception to monitor reference-passing between processes with minimum overhead, as well as (ii) savings in bandwidth and DCD complexity provided by snapshot compression. The combination of these with the possibility of using multiple parallel and hierarchical DCD promotes algorithm scalability.

In conclusion, this work provides an useful extension to a significant runtime environment (.NET with Remoting) as more and more applications are being developed targeting distributed environments, such as clusters (also integrating grid infrastructures), employing distributed virtual machines.

In the future work, we intend to investigate the application of the same techniques to other virtual machines, for example, to provide Java RMI with a complete and portable DGC. We also want to extend our performance study using standard benchmarks (e.g. OO7).

8 References

1 Jones, R., and Lins, R.: 'Garbage collection, algorithms for automatic dynamic memory management' (John Wiley & Sons, 1996)

2 Wilson, P.R.: 'Uniprocessor garbage collection techniques'. Proc. Int. Workshop on Memory Management, Springer-Verlag, Saint-Malo, France, 1992, vol. 637

3 Blackburn, S.M., Cheng, P., and McKinley, K.S.: 'Myths and realities: the performance impact of garbage collection'. Proc. Int. Conf. Measurements and Modeling of Computer Systems, SIGMETRICS 2004, (ACM), New York, NY, USA, 10–14 June 2004, pp. 25–36

4 TC Ecma. 'TG3. Common language infrastructure (CLI)', Standard ECMA-335, 2005

5 Andersson, J., Weber, S., Cecchet, E., Jensen, C., and Cahill, V.: 'Kaffemik-a distributed jvm on a single address space architecture'. Java Virtual Machine Research and Technology Symp., 2001

6 Zhu, W., Wang, C.-L., and Lau, F.C.M.: 'JESSICA2: a distributed Java Virtual Machine with transparent thread migration support'. IEEE Int. Conf. Cluster Computing, 2002

7 Seidmann, T.: 'Distributed shared memory using the .NET framework'. 3rd IEEE/ACM Int. Symp. Cluster Computing and the Grid, 2003, pp. 457–462

8 Frenz, S., Schoettner, M., Goeckelmann, R., and Schulthess, P.: 'Parallel ray-tracing with a transactional DSM'. IEEE Int. Symp. Cluster Computing and the Grid, 2004, pp. 475–481

9 Bonér, J., and Kuleshov, E.: 'Clustering the java virtual machine using aspect-oriented programming'. 6th Int. Conf. Aspect-Oriented Software Development, AOSD.07, ACM SIGPLAN, 2007.

10 Letizi, O.: 'An introduction to terracotta distributed shared objects', Terracotta, available at http://www.terracotta.org/, February 2007

11 Hu, Y.C., Yu, W., Cox, A., Wallach, D., and Zwaenepoel, W.: 'Run-time support for distributed sharing in safe languages', *ACM Trans. Comput. Syst.*, 2003, **21**, (1), pp. 1–35

12 Philippsen, M.: 'Cooperating distributed garbage collectors for clusters and beyond', *Concurrency Pract. Exp.*, 2000, **12**, (7), pp. 595–610

13 Brodie-Tyrrell, W., Detmold, H., Falkner, K., and Munro, D.S.: 'Garbage collection for storage-oriented clusters'. ACSC'04: Proc. 27th Australasian Conf. Computer Science, Australian Conf. Computer Society, Inc., Darlinghurst, Australia, 2004, pp. 99–108

14 Getov, V., von Laszewski, G., Philippsen, M., and Foster, I.: 'Multiparadigm communications in java for grid computing', *Commun. ACM*, 2001, **44**, (10), pp. 118–125

15 Sultan, F., Nguyen, T.D., and Iftode, L.: 'Lazy garbage collection of recovery state for fault-tolerant distributed shared memory', *IEEE Trans. Parallel Distrib. Syst.*, 2002, **13**, (7), pp. 673–686

16 Lowy, J.: 'Programming WCF services' (O'Reilly, 2007)

17 Liu, X., Jiang, H., and Soh, L.K.: 'Exploiting the advantages of object-based DSM in a heterogeneous cluster environment'. IEEE Int. Symp. on Cluster Computing and the Grid, 2005

18 Plainfosse, D., and Shapiro, M.: 'A survey of distributed garbage collection techniques'. Proc. Int. Workshop on Memory Management, Kinross, Scotland (UK), September 1995

19 Abdullahi, S.E., and Ringwood, G.A.: 'Garbage collecting the Internet: a survey of distributed garbage collection', *ACM Comput. Surv.*, 1998, **30**, (3), pp. 330–373

20 Shapiro, M., Le Fessant, F., and Ferreira, P.: 'Recent advances in distributed garbage collection', *Lect. Notes Comput. Sci.*, 2000, 1752, p. 104

21 Ferreira, P., and Veiga, L.: 'Garbage collection curriculum. Msdn academic alliance curriculum repository, object id 6812'. available at http://www.msdnaacr.net/curriculum/pfv.aspx?ID=6182, Microsoft, July 2005

22 Wilson, P.: 'Distributed garbage collection general discussion for faq. GCList Mailing List (gclist@iecc.com)', March 1996

23 Richer, N., and Shapiro, M.: 'The memory behavior of the WWW, or the WWW considered as a persistent store'. POS 2000, 1996, pp. 161–176

24 Birrell, A., Nelson, G., Owicki, S., and Wobber, E.: 'Network objects', *Softw. Pract. Exp.*, 1995, **25**, (S4), pp. 87–130

25 Shapiro, M., Dickman, P., and Plainfosse, D.: 'Robust, dist. references and acyclic garbage collection'. Symp. on Principles of Dist. Computing, Vancouver, Canada, August 1992

26 Bishop, P.B.: 'Computer systems with a very large address space and garbage collection'. MIT Report LCS/TR–178, Laboratory for Computer Science, MIT, Cambridge, MA, 1977

27 Hudson, R.L., Morrison, R., Eliot, J., Moss, B., and Munro, D.S.: 'Garbage collecting the world: one car at time'. Conf. Object-Oriented Programming Systems, Languages, and Applications, Atlanta, USA, October 1997

28 Vestal, S.C.: 'Garbage collection: an exercise in distributed, fault-tolerant programming', PhD thesis, Seattle, WA, USA, 1987

29 Hughes, J.: 'A distributed garbage collection algorithm' in Jouannaud, J.-P. (Ed.): 'Functional languages and computer architectures', vol 201 of Lecture Notes in Computer Science, (Springer-Verlag, Nancy, France, 1985), pp. 256–272

30 Louboutin, S.R., and Cahill, V.: 'Comprehensive dist. garbage collection by tracking causal dependencies of relevant mutator events'. Proc. ICDCS'97 Int. Conf. Dist. Computing Systems, IEEE Press, 1997

31 Le Fessant, F.: 'Detecting distributed cycles of garbage in large-scale systems'. Conf. Principles of Distributed Computing (PODC), August 2001

32 Lang, B., Quenniac, C., and Piquer, J.: 'Garbage collecting the world'. Conf. Record of the 19th Annual ACM Symp. on Principles of Programming Languages, ACM SIGPLAN Notices, (ACM Press), January 1992, pp. 39–50

33 Rodrigues, H., and Jones, R.: 'Cyclic distributed garbage collection with group merger', *Lect. Notes Comput. Sci.*, 1998, **1445**, p. 260

34 Maheshwari, U., and Liskov, B.: 'Collecting cyclic dist. garbage by back tracing'. Proc. PODC'97 Principles of Dist. Computing, 1997

35 Rodriguez-Rivera, G., and Russo, V.: 'Cyclic distributed garbage collection without global synchronization in CORBA'. OOPSLA'97 GC & MM Workshop, 1997

36 Liskov, B., and Ladin, R.: 'Highly-available distributed services and fault-tolerant distributed garbage collection'. Proc 5th Symp. on the Principles of Distributed Computing, Vancouver, Canada, August 1986, pp. 29–39

37 Veiga, L., and Ferreira, P.: 'Complete distributed garbage collection: an experience with rotor', *IEE Proc. – Softw.*, 2003, **150**, pp. 283–290

38 Veiga, L., and Ferreira, P.: 'Asynchronous complete distributed garbage collection'. 19th IEEE Int. Parallel and Distributed Processing Symp., Denver, CO, USA, April 2005

39 Bal, H.E., Tanenbaum, A.S., and Kaashoek, M.F.: 'Orca: a language for distributed programming', *SIGPLAN Not.*, 1990, **25**, (5), pp. 17–24

40 Wollrath, A., Riggs, R., and Waldo, J.: 'A distributed object model for the Java system'. 2nd Conf. Object-Oriented Technologies & Systems (COOTS), 1996, (USENIX Association), pp. 219–232

41 Shapiro, M., Dickman, P., and Plainfosse, D.: 'Robust distributed references and acyclic garbage collection'. Proc. 11th Annual ACM Symposium on Principles of Distributed Computing, Vancouver (Canada), 1992, pp. 135–146

42 Shapiro, M., Gruber, O., and Plainfosse, D.: 'A garbage detection protocol for a realistic distributed object-support system', Technical Report 1320, 1990

43 Mattern, F.: 'Virtual time and global states of distributed systems'. Proc. Int. Workshop on Parallel and Distributed Algorithms, 1989, pp. 215–226

44 Hughes, J.: 'A distributed garbage collection algorithm' in Jouannaud, J.-P. (Ed.): 'Functional languages and computer architectures'. No. 201 in Lecture Notes in Computer Science, (Springer-Verlag, Nancy, France, 1985), pp. 256–272

45 Shapiro, M.: 'A fault-tolerant, scalable, low-overhead dist. garbage collection protocol'. Proc. 10th Symp. on Reliable Dist. Systems, Pisa, September 1991

46 Fisher, M., Lynch, N., and Patterson, M.: 'Impossibility of distributed consensus with one faulty process', *J. ACM*, 1985, **32**, (2), pp. 274–382

47 Lamport, L.: 'Time, clocks, and the ordering of events in a distributed system', *Commun. ACM*, 1978, **21**, (7), pp. 558–565

48 Maheshwari, U., and Liskov, B.: 'Partitioned garbage collection of a large object store'. Proc. SIGMOD'97, 1997

49 Rammer, I.: 'Advanced .NET Remoting' (Apress, 2002)

50 Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J.: 'Aspect-oriented programming'. in Aksit, M., and Matsuoka, S. (Eds.): Proc. European Conf. Object-Oriented Programming, (Springer-Verlag, Berlin, Heidelberg, New York, 1997), vol. 1241, pp. 220–242

51 Box, D., and Sells, C.: 'Essential .NET', The common language runtime, (Addison-Wesley, vol. 1, 2003)

52 Dieckmann, S., and Holzle, U.: 'A study of the allocation behavior of the SPECjvm98 Java Benchmarks'. Proc. 13th European Conf. Object-Oriented Programming, 1999, pp. 92–115

53 Kim, J.S., and Hsu, Y.: 'Memory system behavior of Java programs: methodology and analysis'. Proc. 2000 ACM SIGMETRICS Int. Conf. Measurement and Modeling of Computer Systems, 2000, pp. 264–274

54 Bacon, D.F., Fink, S.J., and Grove, D.: 'Space-and time-efficient implementation of the Java object model'. Proc. 16th European Conf. Object-Oriented Programming, 2002, pp. 111–132

55 Lo, C.T.D., Chang, M., Frieder, O., and Grossman, D.: 'The object behavior of Java objectoriented database management systems'. Int. Conf. Information Technology: Coding and Computing, 2002, pp. 247–252

56 Monson-Haefel, R.: 'Enterprise JavaBeans' (O'Reilly Press, 2000, 2nd edn.)

57 Java Ranch 'Java ranch', available at: http://saloon.javaranch.com/cgibin/ubb/ultimatebb.cgi?ubb=get topic&f=26&t=000522, 2001

58 Sun Microsystems 'Java rmi documentation', available at: http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html, 2004

59 Sun Microsystems 'Sun java bug database, sun developer network', available at: http://bugs.sun.com/bugdatabase/view bug.do?bug id=4403367, 2001