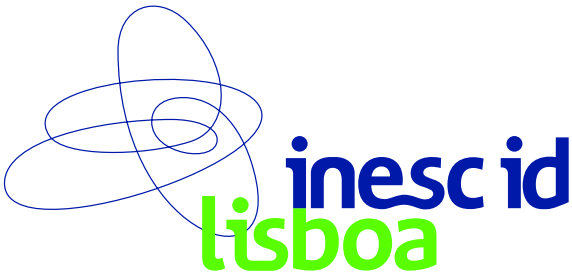


Technical Report RT/07/2005

A Comprehensive Approach for Memory Management of Replicated Objects

Luís Veiga Paulo Ferreira
INESC-ID/IST, Rua Alves Redol N^{o} 9, 1000-029 Lisboa, Portugal
{luis.veiga, paulo.ferreira}@inesc-id.pt

April 2005

Abstract

Replication is a widely-adopted technique for improving data availability and application performance. In the replicated objects model, applications execute methods on objects replicated locally. These objects may contain references to other objects not yet replicated.

The memory management of these distributed (and possibly persistent) graphs of replicated objects is a very difficult task. If performed manually, it leads to memory leaks (useless objects that are never deleted) and dangling references (referencing objects prematurely deleted) causing storage waste and application failure.

Current distributed garbage collection algorithms are not well suited for such systems because either (i) they are unsafe since they do not consider the existence of replication, or (ii) they impose severe constraints on scalability by requiring causal delivery to be provided by the underlying communication layer, or iii) they are not complete, w.r.t distributed cycles of garbage.

We address this problem by developing a comprehensive approach to distributed garbage collection for replicated objects. Our solution is based on: i) an acyclic distributed garbage collector capable of handling replicated objects safely, and ii) a cyclic algorithm that complements the previous one by detecting distributed cycles of garbage involving replicated objects. This twofold hybrid approach is, therefore, complete.

This is the first complete solution of the problem of distributed garbage collection for replicated objects; in particular, the detection and reclamation of distributed garbage cycles does not need any kind of global synchronization. To achieve this goal we introduce the notion of a GC-consistent cut for distributed systems with object replication. We have implemented the algorithm both on Rotor (a free source version of Microsoft .Net, extended to allow object replication) and on OBIWAN (a platform supporting mobile agents, object replication and remote invocation); we observed that applications are not disrupted.

1 Introduction

Replication is a well-know technique for improving data availability and application performance as it allows to collocate data and code. Thus, data availability is ensured because, even if the network is not available, data remains locally available; in addition, application performance is potentially better (when compared to a remote invocation approach) as all accesses to data are local. However, there are several relevant difficulties that must be solved to take full advantage of replication.

In particular, the memory management of distributed (and possibly persistent) graphs of replicated objects is a very difficult task. If performed manually, it leads to memory leaks (useless objects that are never deleted) and dangling references (referencing objects prematurely deleted) causing storage waste and application failure. The reason is that graphs of reachability are large, widely distributed and frequently modified through assignment operations executed by applications. In addition, data replicated in many processes is not necessarily coherent making manual memory management much harder.

Furthermore, the presence of replication increases the need for a complete solution concerning the automatic memory management of replicated objects. As it will become clear in the following sections, it only takes one replica of an object being involved in a distributed cyclic garbage, to consequently encompass all other replicas of the same object in the cycle, and prevent their reclamation. Thus, distributed cyclic garbage involving replicated objects is, arguably, more frequent and wastes more storage, when compared with systems without replication [47, 22, 30].

For these reasons, it is impossible to do manual memory management without generating dangling references and/or memory leaks. Automatic memory management, also known as Garbage Collection (GC), is the single realistic option which is able to maintain referential integrity (i.e. no dangling references or memory leaks) in object-based replicated memory (RM) systems. As a result, program reliability and programmer productivity are clearly improved.

There are two fundamental problems concerning distributed garbage collection (DGC) in RM systems: i) ensuring safety in presence of replication, and ii) achieving completeness. Obviously, the solutions provided must be feasible, in particular, they must be scalable and not intrusive to applications.

The first issue, safety of the DGC, has been solved in our previous work with an acyclic, reference-listing based, replication-aware DGC algorithm [34]. However, this solution is not complete, i.e. it does not reclaim cycles of garbage. The design of complete DGC algorithms is a problem that has been addressed many times before for systems without replication [28, 1]. However, such solutions, besides other considerations concerning scalability, performance, etc., suffer from a fundamental drawback: they are not safe because they do not take into account the existence of replicated objects.

Thus, we propose a solution that, as others before, follows a hybrid approach: an acyclic distributed collector based on reference-listing [35, 34], improved with rules for safety in presence of replicated objects, and a cycle detector that complements the first, thus providing a complete solution for the problem of DGC.

The novelty of our work is that we provide a solution for the detection and reclamation of cycles of garbage for systems with replicated objects; it does not require global synchronization, it is completely asynchronous, and it does not disrupt applications.

Our algorithm uses a centralized approach. The detection of distributed cycles of garbage works on a view of the global distributed graph that is consistent solely for this purpose. This view results from the combination of graph snapshots taken by each process independently. While this view may not correspond to a consistent cut, it still allows safe detection of distributed cycles of garbage. This view results from a cut named GC-consistent-cut. GC-consistent-cuts can be obtained without requiring any distributed synchronization among the processes involved.

The rest of this paper is organized as follows. The next section characterizes the RM model assumed by the DGC algorithm; it is rather abstract and general, so that the GC solutions provided are widely applicable. Section 3 briefly describes the acyclic DGC algorithm. Section 4 presents the algorithm for cycles detection and reclamation. In Section 5, we present a discussion of the algorithm properties. Sections 6 and 7 address the implementation and performance evaluation respectively. In Sections 8 and 9 we describe relevant related work and final conclusions.

2 Model

The DGC algorithm proposed is rather general given that it applies to systems with minimal requirements. Basically these systems support object replication; the underlying model is called Replicated Memory Model (RMM). This model clearly defines the environment for which the DGC algorithm is conceived.

A RM system is a replicated distributed memory spanning several processes. These processes are connected in a network and communicate only by asynchronous message passing. These messages do not support any kind of remote invocation. As a matter of fact, in a RM system, the only way to share information is by replication of data, which can be done with a DSM based mechanism[20]. Thus, processes do not use Remote Procedure Call (RPC) to access remote data. In other words, application code inside a process never sends messages explicitly. Instead, application code access data always locally; transparently to the application code, the RM runtime system is responsible to replicate data locally when needed.

Each participating process in the RM system encloses the following entities: memory, mutator,[1] and a coherence engine. In our RM model, for each one of these entities, we consider only the operations that are relevant for GC purposes.

2.1 Memory Organization

An **object** is defined to be a consecutive sequence of bytes in memory. Applications can have different views of objects and can see them as language-level class instances, memory pages, data base records, web pages, etc.

Objects can contain **references** pointing to other objects. An **outgoing inter-process** reference is a reference to a target object in a different process. An **incoming inter-process** reference is a reference to an object that is pointed from a different process. Our model does not restrict how references are actually implemented. They can be virtual memory pointers, URLs, etc.

We define **local-root** to be the set of references in a process enclosed in global variables and stacks. An object is **reachable-locally** when it is transitively reachable from the local-root of its enclosing process. An object is **reachable-remotely** when it is referenced by other object(s) in different process(es).

For DGC purposes, objects can be either dead or live. An object is said to be **live** if it is reachable-locally (in some process) or if it is reachable-remotely from some object that is reachable-locally (thus, live objects are **reachable-globally**). An object is said to be **dead** if it is neither reachable-locally nor (directly or indirectly) reachable-remotely from a reachable-locally object. Dead objects are called **garbage**. Thus, objects solely reachable-remotely may be either live or dead: live objects are transitively reachable (through a chain of remote references) from a local-root of some other process; dead objects constitute distributed garbage.

Distributed garbage may be acyclic, cyclic, or hybrid. Every DGC algorithm is able to detect acyclic distributed garbage. Complete DGC algorithms are able to detect and reclaim cyclic distributed garbage. Hybrid garbage may be detected by a single algorithm (one that is complete), or by cooperation of acyclic and cyclic collectors.

The unit for replication is the object. Any object can be replicated in any process. A replica of object X in process P is noted X_P. Each process can hold a replica of any object for reading or writing according to the coherence protocol being used.

2.2 Mutator model

The single operation executed by mutators, which is relevant for GC purposes, is **reference assignment**; this is the only way for applications to modify the graph of objects.

The reference assignment operation executed by a mutator in some process P is noted $X := Y_P$. This means that a reference contained in object X is assigned to the value of a reference contained in object Y.[2] If Y points to an object Z in some other process, this assignment operation results in the creation of a new inter-process reference from X to Z.

Obviously, other assignments can delete references transforming objects in garbage. For example, in process P the mutator may perform $(X := NULL)_P$; this

[1] The term mutator [6] designates the application code which, from the point of view of the garbage collector, mutates (or modifies) the reachability graph of objects.

[2] This notation is not fully accurate but it simplifies the explanation of the DGC algorithm. As a matter of fact, to be more precise we should write $X.ref = Y.ref$ (C++ style notation). However, this improved precision is not important for the DGC algorithm description and would complicate it un-necessarily.

may result on some object Z to become unreachable, i.e. garbage, given that there are no references pointing to it. In conclusion, assignment operations (done by mutators) modify the object graph either creating or deleting references.

2.3 Coherence Model

The coherence engine is the entity of the RM system that is responsible to manage the coherence of replicas. The coherence protocol effectively used varies from system to system and depends on several factors such as the number of replicas, distances between processes, and others. However, the only coherence operation, which is relevant for GC purposes, is the **propagation** of an object, i.e. the replication of an object from one process to another. The propagation of an object Y from process $P1$ to process $P2$ is noted $propagateY_{P1 \rightarrow P2}$.

We assume that any process can propagate a replica into itself as long as the mutator causing the propagation holds a reference to the object being propagated. Thus, if an object X is unreachable-locally in process $P1$, the mutator in that process can not force the propagation of X to some other process; however, if some other process $P2$ holds a reference to X, it can request X to be propagated from $P1$ to $P2$ (more details in Section 3).

In each process, the coherence engine holds two data structures, called **inPropList** and **outPropList**; these indicate the process *from which* each object has been propagated, and the processes *to which* each object has been propagated, respectively.[3] Thus, each entry of the **inPropList**/**outPropList** contains the following information:

- **propObj** - the reference of the object that has been propagated into/to a process;
- **propProc** - the process from/to which the object *p*ropObj has been propagated;
- **sentUmess**/**recUmess** - bit indicating if a *Unreachable* message has been sent or received (more details in Section 3).

In the rest of this paper, for clarity, entries in the **inPropList** and **outPropList** will be referred to as *InProp*/*OutProp* entries, or simply as *InProps*/*OutProps*.

[3]Usually, this information does exist in the coherence engine in order to manage the replicas.

When an object is propagated to a process we say that its enclosed references are **exported** from the sending process to the receiving process; on the receiving process, i.e. the one receiving the propagated object, we say that the object's enclosed references are **imported**.

It's worthy to note the following aspects concerning the creation of inter-process references. The only way a process can create an inter-process reference is through the execution of only two operations: (i) reference assignment, which is performed explicitly by the mutator (as described in Section 2.2), and (ii) object propagation, which is performed by the coherence engine in order to allow the mutator to access some object.[4]

3 Acyclic Distributed Garbage Collection

The overall solution for the problem of DGC in a RM system, is constituted by the following algorithms: i) a local tracing-based garbage collection (LGC) algorithm running in each process, ii) a replication-aware reference-listing acyclic distributed garbage collector (ADGC) algorithm [34], and iii) a distributed cycles detector (DCD) algorithm. The LGC, ADGC and DCD algorithms depend on each other to perform their job, as explained afterwards.

Common to both the ADGC and the DCD (and used also by the LGC), there are two main data structures:

- Stub - A stub describes an outgoing inter-process reference, from a source process to a target process (e.g. from object X in a process P1 to object Y in P2).
- Scion - A scion describes an incoming inter-process reference, from a source process to a target process (e.g. to object Y in a process P2 from object X in P1).

Thus, a scion represents an incoming reference, i.e., a reference pointing to an object in the scion's process; a stub represents an outgoing remote reference, i.e., a reference pointing to an object in another process.

[4]For example, in some DSM-based systems, when the mutator tries to access an object that is not yet cached locally, a page fault is generated; then, this fault is automatically recovered by the coherence engine that obtains a replica of the faulted object from some other process.

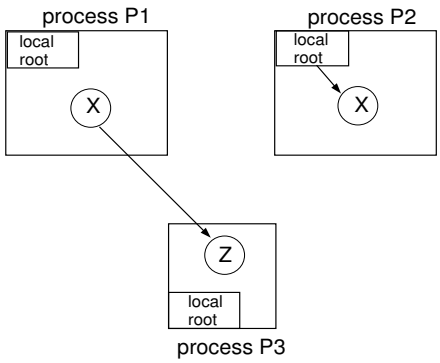


Figure 1: *Safety problem of current DGC algorithms which do not handle replicated data: Z is erroneously considered unreachable.*

Note that stubs and scions do not impose any indirection on the native reference mechanism. In other words, they do not interfere either with the structure of references or the invocation mechanism. They are simply GC specific auxiliary data structures. Thus, stubs and scions should not be confused with (virtual machine) native stubs and scions (or skeletons/proxies) used for remote method invocations (RMI).

In the remaining of this section we describe the LGC and ADGC algorithms. These algorithms are (briefly) presented mainly for completeness of the solution and as a means to explain the usefulness of stubs and scions, and how replication is safely dealt with. Then, in Section 4 we focus on the DCD algorithm.

3.1 Replication Awareness

Both the ADGC and the DCD algorithms follow a set of well-defined rules (presented later) so that they are safe and live in presence of replication. This means that these algorithms solve the safety problem that is not addressed by other DGC algorithms [28, 1]; as a matter of fact, such algorithms do not take into account the existence of replicated objects, as explained now.

Consider Figure 1 in which an object X is replicated in processes $P1$ and $P2$. Now, suppose that X_{P1} contains a reference to an object Z in another process $P3$, X_{P1} points to no other object, X_{P1} is not reachable-locally and X_{P2} is reachable-locally. Then, the question is: should Z be considered garbage? Classical DGC algorithms (designed for function-shipping systems) consider that Z is effectively garbage. However, this is wrong because, in a RM system, it is possible for an application in $P2$ to acquire a replica of X from some other process, in particular, X_{P1}. Thus, the fact that X_{P1} is not reachable-locally in process $P1$ does not mean that X is unreachable-globally; as a matter of fact, according to the coherence model, X_{P1} contents can be accessed by an application in process $P2$ by means of a propagate operation.

Therefore, in a RM system, a target object Z is considered unreachable only if the union of all the replicas of the source object, X in this example, do not refer to it. This is the **Union Rule** introduced in Larchant [10, 11]: *a target object Z is considered unreachable only if the union of all the replicas of the source objects do not refer to it.* The next sections show how this rule is enforced.

3.2 LGC

Each process has a local garbage collector (LGC); it works as any standard tracing collector with the differences stated now. The LGC starts the graph tracing from the process's local-root and set of scions. For each outgoing inter-process reference it creates a stub in the new set of stubs. Once this tracing is completed, every object reachable-locally by the mutator has been found (e.g. marked, if a mark-and-sweep algorithm is used); objects not yet found are unreachable-locally; however, they can still be reachable from some other process holding a replica of, at least, one of such objects (as is the case of X_{P1} in Figure 1). To prevent the erroneous deletion of such objects, the collector traces the objects graph (marking the objects found) from the lists `inPropList` and `outPropList`, and performs as follows: i) when a reachable-locally object (previously discovered by the LGC) is found, the tracing along that reference path ends, and ii) when an outgoing inter-process reference is found the corresponding stub is created in the new set of stubs.

3.3 ADGC

From time to time, possibly after a local collection, the ADGC sends a message *NewSetStubs*; this message contains the new set of stubs that resulted from the previous local collection; this message is sent to the processes holding the scions corresponding to the stubs in the previous stub set. In each of the receiving processes, the ADGC matches the just received set of stubs with its set of scions; those scions that no longer have the corresponding stub, are deleted.

As previously stated, the ADGC, to be correct in presence of replicated objects, must ensure the Union Rule.

This rule, fundamental for the safety of the ADGC, is ensured as follows:

- For an object which is reachable only from the `inPropList`, a message *Unreachable* is sent to the site from where that object has been propagated; this sending event is registered by changing a `sentUmess` bit in the corresponding `inPropList` entry from 0 to 1.[5]

  When a *Unreachable* message reaches a process, this delivery event is registered by changing a `recUmess` bit in the corresponding `outPropList` entry from 0 to 1.

- For an object which is reachable only from the `outPropList`, and the enclosing process has already received a *Unreachable* message from all the processes to which that object has been previously propagated, a *Reclaim* message is sent to all those processes and the corresponding entry in the `outPropList` is deleted; otherwise, nothing is done.

  When a process receives a *Reclaim* message it deletes the corresponding entry in the `inPropList`.

In summary, besides the message *NewSetStubs*, two other messages may be sent by the ADGC: *Unreachable* and *Reclaim*. On the receiving process, these messages are handled by the ADGC that performs the following operations: sets the `recUmess` bit in the corresponding `outPropList` entry, and deletes the corresponding entry in the `inPropList`, respectively. Thus, a replicated object is effectively reclaimed (by the LGC) only after the corresponding entry in the `inPropList` is deleted.

3.4 Propagation

In a RM system mutators may create inter-process references very easily and frequently, through a simple reference assignment operation (see Section 2.2). Note that when such an assignment does result in the creation of an inter-process reference, this can only happen because, in the local process, there was already an object replica containing the reference to a remote object. Thus, inter-process references are created as a result of the propagation of replicas. Such propagation leads to the export and import of references, as mentioned in Section 2.3.

Thus, whatever the coherence protocol, there is only one interaction of the mutator with the ADGC algorithm. This interaction is twofold: (i) immediately before a propagate message is sent, the references being exported (contained in the propagated object) must be found in order to create the corresponding scions, and (ii) immediately before a propagate message is delivered, the outgoing inter-process references being imported must be found in order to create the corresponding local stubs, if they do not exist yet. Note that this may result in the creation of chains of stub-scion pairs, as it happens in the SSP Chains algorithm [36]. To summarize, the following rules are enforced by the ADGC:

- **Clean Before Send Propagate**: Before sending a propagate message, enclosing an object Y, from a process $P2$, Y must be scanned for references and the corresponding scions created in $P2$.

- **Clean Before Deliver Propagate:** Before delivering a propagate message, enclosing an object Y, in a process $P1$, Y must be scanned for outgoing inter-process references and the corresponding stubs created in $P1$, if they do not exist yet.

It is worthy to note that the mutator does not have to be blocked while the ADGC specific operations mentioned above are executed (scanning the object being propagated and creating the corresponding scion and stub); such operations can be executed in the background.

From these rules, results the fact that scions are always created before the corresponding stubs; and OutProps are always created before their corresponding InProps. This is due to a causality relationship (their creation is causally ordered) between them.

3.5 Completeness

The ADGC is not complete as it does not reclaim distributed cycles of garbage. The detection and deletion of distributed cycles of garbage is a difficult problem that has been addressed in many ways: global tracing, back-tracing, detection within groups, with centralized or distributed approaches (see Section 8 for a comparison of the most relevant work to ours).

[5] Note that from now on, the replica is not reachable by the local mutator; if another propagate operation occurs bringing a *new* replica of that same object into the process, the *old* replica remains locally unreachable, and a new entry is created in the `inPropList` with the corresponding `sentUmess` set to 0.

We solved this limitation by developing another algorithm, the DCD algorithm, capable of detecting such cycles asynchronously. Once a cycle is detected, the DCD instructs the ADGC algorithm to delete one of its entries so that the cycle is eliminated. Then, the ADGC is capable of reclaiming the remaining garbage objects.

Our algorithm makes use of a centralized approach. The detection of distributed cycles of garbage works on a view of the global distributed graph that is consistent for its purposes. Such a view results from the graph snapshots taken by each process independently (i.e., with no synchronization required at all). As explained later, this view may not correspond to a consistent cut (as defined by Lamport [18]) but it still allows to safely detect distributed cycles of garbage. This view, results from a cut that we call a GC-consistent-cut. GC-consistent-cuts can be obtained without requiring any distributed synchronization among the processes involved.

4 Cyclic Distributed Garbage Collection

In this section, we describe the DCD algorithm; we first provide an overview of the algorithm, the main data structures, and then we present the details of the detection of distributed garbage cycles.

Objects are represented by their name (a letter) and their enclosing process (e.g., A_{P1} in Figure 2). Subgraphs of connected objects may be represented in abbreviation (e.g., $\{\{A,\ W',\ B\}_{P1}, \{F,\ H,\ I\}_{P2}\}$), aggregated by its/their enclosing process. References may be also explicitly described when relevant (e.g., $B_{P1} \rightarrow F_{P2}$).

Remote references are described by their corresponding stubs and scions (e.g., $B_{P1} \rightarrow F_{P2}$). Objects replicated from/to processes (e.g., W'_{P1}) are represented with their associated inProp/outProp entries. Furthermore, the association among replicas of the same object, in different processes, is made explicit by gray dashed lines. This eases visualization of the Union Rule presented earlier.

Throughout this section, for clarity, we extend the notions of reachability, already defined for objects (recall Section 2.1), also to GC structures like scions, stubs and InProp and OutProp entries. In particular, when we say that some stubs are reachable from a scion, we actually mean: *stubs, describing remote references, enclosed in objects, which are reachable from another object, targeted by an incoming remote reference, described by a scion.*

4.1 Cyclic Garbage Comprising Replicated Objects

Following this notation, a simple example of a distributed cycle, comprising replicated objects, can be seen in Figure 2. This cycle can be represented by the following (others possible) chain of objects (starting and finishing in $P2$):

$$\{\{F,\ H,\ I\}_{P2}, \{O,\ W,\ K\}_{P3}, \{D,\ W',\ B\}_{P1}\}$$

Clearly, all objects belong to a distributed garbage cycle, since none of them is reachable from any local root (the one in process $P1$ targeting object A_{P1} has been deleted by the mutator). Therefore, there are no sources of global reachability. However, in this situation, the ADGC algorithm is unable to proceed, because it considers objects to be live, when they are reachable from scions.

Replicas W_{P3} and W'_{P1} must both be found unreachable for any (or both) of them to be reclaimed. However, both of them are targeted by other objects (O_{P3}, D_{P1} respectively) that are reachable remotely (due to I_{P2}, K_{P3} respectively). Thus, the ADGC algorithm presented earlier will never issue *Unreachable* (and consequently, *Reclaim*) messages regarding replicas W_{P3} and W'_{P1}. Conversely, without receiving these messages, processes $P1$ and $P3$ will remain including stubs regarding remote references ($B_{P1} \rightarrow F_{P2}$ and $K_{P3} \rightarrow D_{P1}$, respectively) in their *NewSetStubs* messages. Due to this double interdependency, the ADGC algorithm always perceives the objects included in the example portrayed, as reachable-globally (therefore, as live objects) while, in fact, they are no longer reachable to the mutator.

Cyclic distributed garbage is created at a lower rate than acyclic distributed garbage. Nevertheless, it is still frequent [47, 22, 30]. Arguably, it is even more so with replicated objects. If not detected and reclaimed, it simply accumulates over time, wasting an ever increasing fraction of the memory space.

Thus, to achieve completeness, we must also provide a detector for cyclic distributed garbage objects. Existing cycle detectors (and otherwise complete DGC algorithms), found in the literature, are not applicable to the RM model. They cannot handle replication safely as stated earlier, given that they do not take replication into account. Naive extensions of these algorithms, in

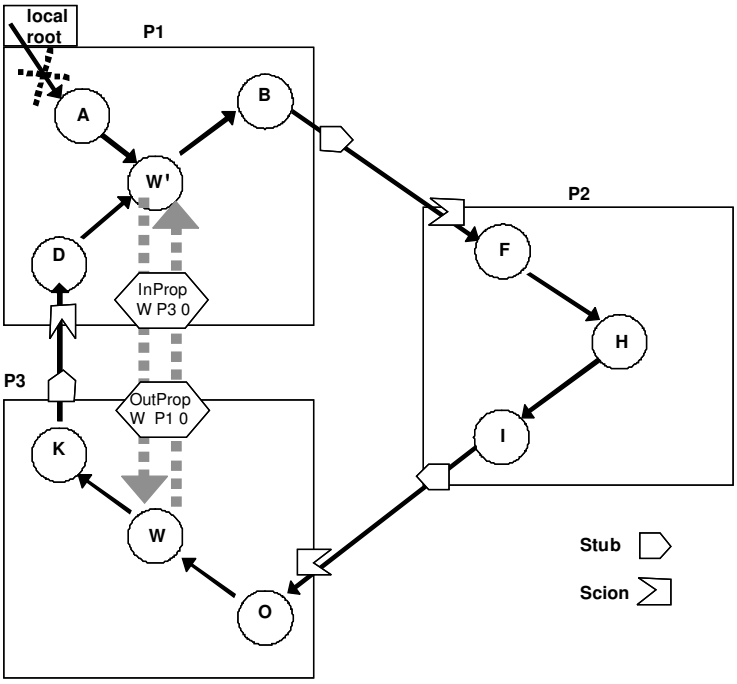


Figure 2: Cyclic distributed garbage comprising replicated objects. Object *W* has been previously replicated from process *P*3 to process *P*1.

which replicas are simply considered as additional objects, are not safe either and lead to premature reclamation of replica content, that could still be of potential use to applications (recall Figure 1). A comprehensive overview of DGC algorithms, with further details, is presented in Section 8.

4.2 Algorithm Overview

Our algorithm makes use of a centralized approach. The first approach of this kind was introduced in [21]. However, our work has several differences that will be become clear afterwards.

In the following paragraphs, we describe the main idea of the DCD algorithm. We follow an intuitive description that does not consider many subtle aspects; these are addressed in the next sections. However, it provides a description of the main idea that is easy to understand.

The process performing the detection of the distributed cycles of garbage (called DCDP for distributed cycles detector process) receives object graph snapshots from each participating process and detects distributed cycles comprised within these processes (existing at the time these snapshots were taken). As a matter of fact, the DCDP receives summarized snapshots of processes (instead of full graphs). These summarized snapshots contain all the information relevant for DCD purposes, and are much smaller than the full object graphs.

Using these snapshots, the DCDP performs a global mark-and-sweep (GMS) on the graphs description received. This is performed in a way so that inter-process references are traced only if the corresponding stub-scion pairs are consistent in the graphs description. Similarly, corresponding OutProp and InProp entries, indicating replication paths are also traced by the GMS, as implicit inter-process references, in order to uphold the Union Rule. Otherwise, the marking on that reference stops.

When a garbage cycle is detected, the DCDP can instruct certain processes to delete one or more of their scions, InProp or OutProp entries. The explicit deletion of such structures is a safe operation due to the property of garbage being stable, i.e., once an object is garbage, it stays so. Explicit deletion of InProp and OutProp can be performed by triggering entries to send *Unreachable* messages. The end result of these explicit deletions is the transformation of a distributed cycle of garbage into a set of acyclic garbage objects; thus, these objects can

be readily reclaimed by the ADGC algorithm described previously.

The DCDP works on a view of the global distributed graph that is consistent for its purposes. As explained in this section, this view may not correspond to a consistent cut (as defined by Lamport [18]) but it still allows to safely detect distributed cycles of garbage. We call such a cut, a GC-consistent-cut.

This a weaker requirement than that of a consistent-cut in a distributed system due to: i) distributed cyclic garbage (as all garbage) is stable, i.e., after it becomes garbage it will not be touched again by the mutator, and ii) distributed cyclic garbage is always preserved by the ADGC (that is why we need a special detector), i.e., if the DCD algorithm does nothing, it still is safe.

GC-consistent-cuts can be obtained without requiring a distributed consensus [15] among the applications processes that send their graph descriptions to the DCDP. This means that the DCDP still performs useful work, i.e. it is capable of detecting cycles, even if its global view of the graph is made of local graph descriptions (sent by the applications processes) at different and uncoordinated moments.

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

4.3 Data Structures

The data structures manipulated by the DCD algorithm are extensions to the ADGC data structures (*Stubs*, *Scions*, *InProp* and *OutProp*). For the DCD algorithm, these structures are grouped, in each process, in two sets: $Source-Set$ and $Target-Set$.

The $Source-Set$, in a process, includes all the entries of the above mentioned data structures that can propagate global reachability *inside* a process, i.e., commonly scions, and OutProp and InProp entries. The need to include InProp entries along with OutProp entries stems from the need to uphold the Union Rule. This way, whenever a replica is reachable-globally, every other replica of the same object must be so as well. Thus, globally reachability must be propagated both ways, thus through OutProp and InProp entries.

The $Target-Set$ in a process includes all the entries of the above mentioned data structures that can propagate global reachability *outside* a process, i.e., commonly stubs, and InProp and OutProp entries. The need to include OutProp entries along with InProp entries, is symmetrical to the previous case. It also stems from the need to uphold the Union Rule.

The notions of $Source-Set$ and $Target-Set$ (calculated via graph summarization described further in Section 4.4) are illustrated generally in Figure 3. Note that to uphold the Union Rule, InProp and OutProp entries belong to both $Source-Set$ and $Target-Set$ of the summarized graph. This is made explicit in the original graph by using double-direction arrows between these entries and the objects they refer to. In the summarized version, this is made clear with different shadings: brighter for $Source-Set$ and darker for $Target-Set$.

The ADGC structures must be extended with timestamps. Thus, DCDP is able to perform GMS along distributed paths consistently, for cycle detection purposes. When needed information is missing, the DCDP acts conservatively to ensure safety. Nonetheless, these timestamps may already be present for other DGC and coherence purposes, like preserving correctness in the presence of lost, duplicated or delayed messages. Thus, this may not be an additional demand to most systems.

Additionally, every entry belonging to the $Source-Set$ must include the set of entries of the $Target-Set$ that are (transitively) reachable from it. The need for this information is justified and detailed in Section 4.4.

Finally, every entry in the $Target-Set$ must bear a special bit indicating reachability-local, i.e., if there is a transitive path from a local root of the enclosing process, that can lead to this entry.

In summary, the extensions to these structures, manipulated by the DCDP, are as follows.

Scion (member of the $Source-Set$):

- $time-stamp$: for GC-consistent-cut purposes.
- *TargetsFrom*: list of stubs, InProps, and OutProps, in the same process, transitively reachable *from* the scion.

Stub (member of the $Target-Set$):

- $time-stamp$: for GC-consistent-cut purposes.
- *Reach.LOCAL*: flag-bit accounting for local reachability (from the local root of the enclosing process) of the stub.

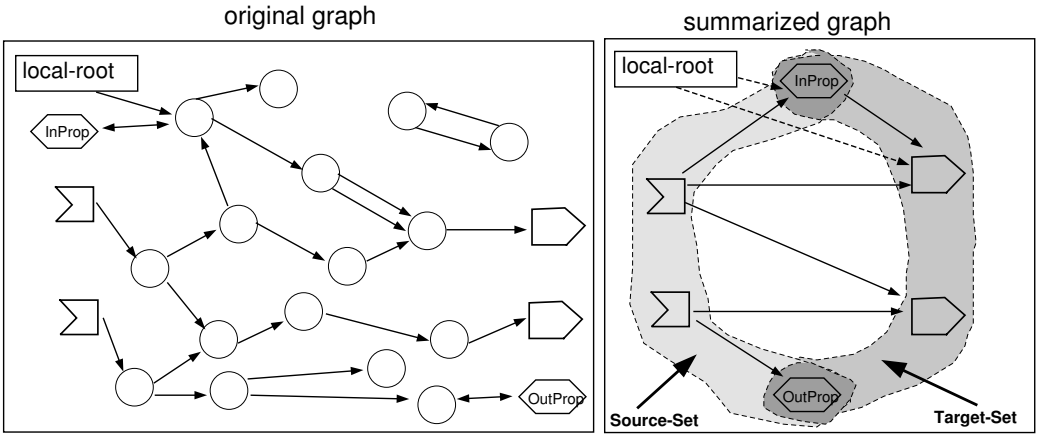


Figure 3: Summarization of an object graph into $Source - Set$ and $Target - Set$.

OutProp (member of the $Source - Set$ and of the $Target - Set$):

- $time - stamp$: for GC-consistent-cut purposes.
- *TargetsFrom*: list of stubs, InProps, and OutProps, in the same process, transitively reachable *from* the object the OutProp refers to.
- *Reach.LOCAL*: flag-bit accounting for local reachability (from the local root of the enclosing process) of the object the OutProp entry refers to.

InProp (member of the $Source - Set$ and of the $Target - Set$):

- $time - stamp$: for GC-consistent-cut purposes.
- *TargetsFrom*: list of stubs, InProps, and OutProps, in the same process, transitively reachable *from* the object the InProp refers to.
- *Reach.LOCAL*: flag-bit accounting for local reachability (from the local root of the enclosing process) of the object the InProp entry refers to.

The *TargetsFrom* lists, held for each entry in the $Source - Set$, establish reachability associations, among objects targeted by incoming remote references, and objects holding outgoing remote references, in each process. These lists allows the DCD algorithm to determine, while detecting cycles, the next set of processes (targeted by implicit or explicit outgoing references) that should be marked in order to transverse the full distributed graph available. Finally, the *Reach.LOCAL* flag, in each element of the $Target - Set$, indicates the local reachability of the entries for inclusion in the GMS roots.

These structures effectively summarize, in each process, and solely for distributed cycle detection purposes, all the relevant information of application object graphs. The full calculation of these structures is addressed in the following subsection.

4.4 Graph Summarization

Object graphs in application processes may be very large. Consequently, the size of the corresponding snapshot may contribute to increase bandwidth usage, memory occupation by the DCDP, and a large amount of disk space. In addition, such a large amount of data could turn cycle detection into a CPU-consuming operation requiring access to a large amount of data.

This problem is solved by summarizing the object graph snapshot of each application process in such a way that, from the point of view of the DCDP, there is no loss of relevant information. This summarization transforms a snapshot of an application graph into the two sets ($Source - Set$ and $Target - Set$) presented earlier, with their corresponding associations.

As a matter of fact, references strictly internal to a process are not relevant for the DCDP, as long as the relations between entries in the $Source - Set$ and $Target - Set$ are known. In Fig. 2, in process $P2$, references $\{F \rightarrow H,\ H \rightarrow I\}_{P2}$ fall into this category. In other words, what is relevant for the DCD algorithm, is to know which stubs, InProps and OutProps (i.e.,the

Target − Set), are reachable form scion, InProps and OutProps (i.e., the *Source − Set*).

This summarization is performed on every snapshot; then it is made available to the DCDP. Thus, while processes can take snapshots by serializing local graphs, the DCDP uses them only in their summarized form, i.e., after graph summarization. In the remainder of the document, the terms *snapshot*, *graph description*, and *summarized graph description* are logically equivalent, w.r.t. the DCDP. Once available, summarized object graphs are sent to the DCDP. The summarized graph has obvious advantages both in terms of network and disk usage. In addition, this summarization can be performed lazily, with low priority, with minimal impact on application performance (see Section 7).

In the example shown in Fig. 2, the summarized graph information at process $P1$ would hold the following data:[6]

- $Scion(D_{P1})_{@P1} \Rightarrow \{TargetsFrom \equiv \{F_{P2},\ W'_{P1}\}\}$; this means that, in $P1$, $Scion(D_{P1})$ leads *to* $Stub(F_{P2})$ and $InProp(W'_{P1})$ (describing replication via W_{P3}).

- $Stub(F_{P2})_{@P1} \Rightarrow \{Reach.LOCAL \equiv false\}$; this means that $Stub(F_{P2})$ is not reachable from the local root of process $P1$ (*Reach.LOCAL* is $false$). Note that while the stub refers to an object located in process $P2$, the stub structure is kept *at* process $P1$ where the remote reference actually exists. This is denoted by the symbol @.

- $InProp(W'_{P1})_{@P1} \Rightarrow \{TargetsFrom \equiv \{F_{P2},\ W'_{P1}\}, Reach.LOCAL \equiv false\}$; this means that $InProp(W'_{P1})$ leads *to* $Stub(F_{P2})$ and $InProp(W'_{P1})$ ($InProp(W'_{P1})$ leads to itself, since InProps and OutProps propagate reachability marks in both directions due to the Union Rule; in this case, it will propagate reachability **to** and **from** $OutProp(W_{P3})$). Furthermore, it is not reachable from local root of $P1$.

Therefore, summarized information of process $P1$ is completed with the contents of *Source − Set* and *Target − Set*:

- $Source - Set(P1) \Rightarrow \{Scion(D_{P1}),\ InProp(W'_{P1})\}$
- $Target - Set(P1) \Rightarrow \{Stub(F_{P2}),\ InProp(W'_{P1})\}$

Graph summarization is performed transversing the graph, while propagating and combining reachability information (regarding elements in the *Source − Set*) of objects. This is done breadth-first, in order to minimize the number of times each object is re-scanned.

4.5 Distributed Cycle Detection Process

As already mentioned, to perform the DCD algorithm, the DCDP receives a summarized description of the object graph of applications processes.[7] Note that such object graphs are strictly local to each application process. In addition, as will be made clear afterwards, the DCDP does not require the object graph of all the existing application processes to perform useful work; thus, cycles are detected even with a partial view of the global graph.

It's worthy to note that our algorithm does not require the snapshots to be taken synchronously by every application involved. In other words, there is no need for a distributed consensus [15] which would be clearly a bad solution for performance and scalability reasons. Thus, as explained now, the DCDP analyzes the object graphs with special care for consistency and causality from a DGC point of view.

Each process maintains a private counter global to the process. Time-stamps are created, in each process, by monotonically increasing this counter. Thus, each time a scion, or OutProp entry is created, the counter value is stored in its time-stamp field, and the counter is atomically incremented. Stubs (and InProp entries) receive the same time-stamp stored at their counterpart scions (and OutProp entries, respectively).

Each process sends to the DCDP, included with its summarized graph description, a list containing, for each process, the highest time-stamp value it has received from it. Then, the DCDP performs a global mark-and-sweep (GMS) on the graphs description received. This GMS is done in such a way that inter-process references are traced only if the corresponding stub-scion pair exists in the graphs description. Similarly, corresponding OutProp and InProp entries, indicating replication paths are

[6] Symbol $\Rightarrow$ means *evaluates to* or *returns*, $\equiv$ relates a field name and its value.

[7] A description of an object graph is obtained using a library that, through serialization, writes a file describing the objects, stubs and scions, InProps and OutProps, of the process. This description is then subject to a summarization process.

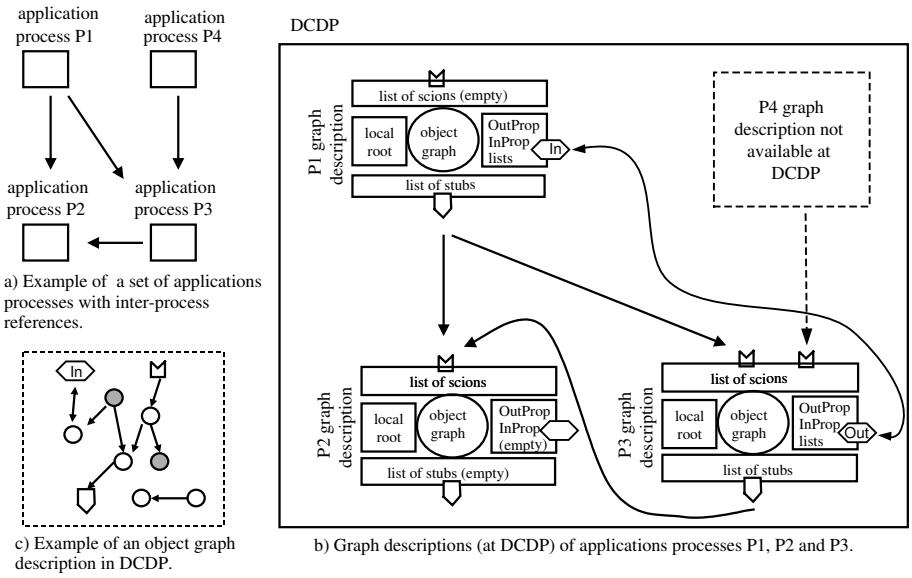


Figure 4: Roots of the GMS at the DCDP.

also traced by the GMS, as implicit inter-process references, in order to uphold the Union Rule. Otherwise, the tracing on that reference stops.

The roots of the GMS are the following (see Figure 4):

- Those objects that, in each application process, are directly reachable from the local roots (stack, etc.) must be obviously considered roots of the GMS (in Figure 4-c such objects are shaded).

- Scions whose corresponding stubs are included in processes whose graph description is not available at the DCDP (when performing the GMS), are also members of the GMS root (in Figure 4-b such a scion is the one in P3 whose corresponding stub is in P4). These scions are members of the GMS root for safety reasons. As a matter of fact, such scions may not have a corresponding stub (so they could be simply discarded) but the DCDP can't say that for sure. Thus, it uses a conservative approach.

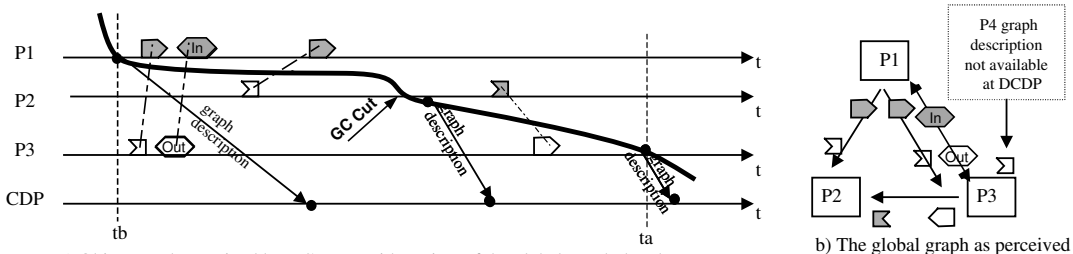
- InProp and OutProp entries, whose corresponding OutProps and InProps are included in processes not available at the DCDP, must also be considered as members of the GMS root. This is a conservative approach, once again, to ensure safety.

- Those scions whose associated time-stamp has a value greater than the greatest value known in the process holding the corresponding stub, are also members of the GMS root.

- OutProp entries, whose associated time-stamp has a value greater than the greatest value known in the processing holding the corresponding InProp entry, are also members of the GMS root.

The last two rules enforce a conservative approach to ensure safety. The scions in this situation are those whose corresponding stubs have not been created yet when their enclosing application process has created its graph description (then sent to the DCDP). Similarly, the OutProp entries are those whose corresponding InProp entries have not been created when their enclosing process generated its graph description (then sent to the DCDP).

Note that the situations, described in the last two items, may occur because the graph descriptions received by the DCDP are snapshots taken at different moments at different processes, with no coordination at all. This is a consequence of the fact that there is no need for global synchronization among participating processes, w.r.t. generating summarized graph descriptions and sending them do the DCDP. Such a situation is illustrated in Figure 5. At moment `ta`, the graphs are in fact those as illustrated in Figure 4-a. However, the view DCDP has, based on graph descriptions received so far, is different because the graph descriptions obtained from P2 is older than P3



a) Object graphs received by DCDP provide a view of the global graph that does not corresponds to a real one; the differences are due to the shaded stubs, scions, and InProp and OutProp entries. LGCs are not represented as they can occur at any time. The line in bold represents an inconsistent cut but safe for cyclic GC purposes. Creation of scion and OutProp entries always preceed creation of stubs and InProps entries. This is because there is a causality relation among them, due to the ADGC safety rules presented.

b) The global graph as perceived by the DCDP based on the graph descriptions received so far: shaded elements do not exist.

Figure 5: View of the graphs as seen by the DCDP.

and the graph description of P1 is even older than P2's. The shaded scions and stubs reflect such differences.

In Figure 5-a we show, in bold, a cut that is not causally consistent; it results from the uncoordinated creation and sending of summarized object graphs, from each application process to the DCDP. It is clear that this cut is such that the creation of stubs and scions, InProps and OutProps, is not consistent for regular DGC purposes. However, based on such graphs, the DCDP builds a GC-consistent cut that allows it to detect distributed cycles of garbage. This cut is consistent w.r.t. the finding of such cycles. Thus, a GC-consistent cut is a set of GC structures, together with their associations, that provide a safe view of the distributed object graph, for the sole purpose of detecting cycles. This is ensured as long as the rules to define the root-set of the GMS (performed by the DCDP) and tracing are respected. In particular, these rules specify which scions, OutProps and InProps must be members of the root-set of the GMS.

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

The DCDP determines which of such scions, OutProps and InProps actually belong to cycles comprising replicated objects. This is done as follows: only scions that are simultaneously garbage and, still, referenced by stubs, can belong to a distributed cycle of garbage. Similarly, only those OutProp entries found to be garbage but with their corresponding InProp entries known to the DCDP, can belong to cyclic garbage. Then, one, any, or all of them, can be selected for deletion and the adequate message(s) sent to their enclosing process(es). The number of messages sent only influences the bandwidth used and the speed of cycle reclamation. Those distributed garbage cycles that already existed when the earliest graph description (being processed by the DCDP) was created, and are totally included in the graph descriptions available at the DCDP, are effectively detected and reclaimed. Thus, considering Figure 5-a, all cycles that existed before `tb`, that are totally enclosed in processes P1, P2, and P3, are detected by the DCDP.

5 Discussion of Algorithm Properties

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

Safety: The ADGC and DCD algorithms are resilient to message loss, delay, re-ordering and replay. Concerning ADGC, loss or delay of *NewSetStubs*, *Unreachable* and *Reclaim* messages does not affect safety because objects deletion is only triggered by reception of these messages. It may, however, delay garbage detection. Message replay is innocuous since all messages are idempotent. This is trivial for *Unreachable* and *Reclaim* messages. *NewSetStubs* messages always carry information about the most recent scion known when they were first sent. This way, a replayed *NewSetStubs* message will not prematurely delete scions created more recently.

W.r.t. DCD, when the DCDP performs GMS, it is safe when considering scions, InProps and OutProps refer-

ring to processes that have yet not sent their graph descriptions to the DCDP. It conservatively considers them as GMS roots. Furthermore, scions, InProps and OutProps, with time-stamps greater than the highest value known in the process holding the stub, InProp and OutProp counterparts, are also conservatively considered as GMS roots. There are no ordering requirements, and therefore no competition, nor racing conditions, among messages sent to DCDP. Message loss will only delay garbage detection. Replay of older messages may, however, prevent detection of newer cycles. This will be solved when an updated graph description is received by the DCDP. Additionally, several independent DCDP may execute without error.

Liveness: Algorithm liveness, w.r.t. acyclic distributed garbage, relies on processes sending *NewSetStubs* messages (containing live stubs) and *Unreachable* and *Reclaim* messages regarding unreachable replicas. This is ensured since every process will eventually send these messages after execution of the LGC.

W.r.t. to DCD, the algorithm liveness is obviously dependent on DCDP receiving messages, carrying graph descriptions, by participating processes. These graph descriptions must be eventually updated regarding each of these processes.

Completeness: The algorithm is complete in the sense that any cyclic distributed garbage is eventually detected and reclaimed. Distributed garbage cycles comprise replicated objects that eventually will be included in the graph descriptions, created independently by the processes comprising them. Once these summarized descriptions are made available to the DCDP, it will detect the distributed garbage cycles contained in them.

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

Floating-garbage consists of just recently created distributed cycles that cannot be detected until summarized graph information, at processes, correctly reflects it. The algorithm is conservative in these situations. Obviously, this is an inevitable phenomenon to GC in general. However, the relevant issue w.r.t. cycle detection is eventually detecting them, since they are stable (therefore, long-lived), and created at a slow rate.

Termination: Regarding termination, cycle detections are trivially granted to terminated due to the centralized approach used for cycle detection. Once the DCDP initiates a cycle detection regarding a set of participating processes, it will terminate promptly whether cycles are found or not. Propagation of reachability marks during GMS is granted to finish, since it is performed locally and needs to visit each element (belonging to $Source-Sets$ and $Target-Sets$) only once.

Scalability: W.r.t. scalability, it stems mainly from the loose synchronization requirements (cycle detection is performed asynchronously w.r.t. participating processes), and detections in course do not require storing additional state information at participating processes. Moreover, each participating process is ignorant of the others sending summarized graph descriptions to the DCDP. Hierarchical cooperation of several DCDPs, already presented when we addressed completeness, also contributes to scalability w.r.t. distributed cycle detection. In addition, several distinct DCD can be done in parallel using different DCDP.

6 Implementation

The algorithms were implemented in Rotor[39] (a free version of Microsoft .Net[29]), that we extended to support object replication. We have also implemented them in OBIWAN [40, 41, 13], a middleware that supports object replication on top of .Net and Java.

The algorithms (ADGC and DCD) were implemented combining modules written in C++ and C#. The implementation we have done on Rotor includes virtual machine modification (for LGC and DGC integration), Re-

moting code instrumentation (to detect export and import of references), and distributed cycle detection.

Virtual machine modifications were implemented in C++. This is the language Rotor core is implemented in. Remoting instrumentation code was developed in C#, since high-level code of the Remoting services is written in this language. Graph summarization and the actual DCDP were also written in C#.

In this section we briefly describe the most important implementation aspects concerning the above mentioned code, with greater emphasis on the modifications made on Rotor.

6.1 LGC and ADGC Integration

The ADGC algorithm must cooperate with the LGC, essentially, in two ways:

- the LGC must provide, in some way, the ADGC with information about every remote objects referenced by local objects; this is necessary to ensure that all stubs (representing outgoing remote references) are correctly created/preserved;

- the ADGC algorithm must prevent the LGC from reclaiming objects that are no longer reachable-locally but are target of incoming remote references; this ensures that scions actually prevent objects from being reclaimed. Similarly, InProp and OutProp entries also prevent replicated objects from being prematurely reclaimed.

The implemented solution consists simply on a running thread that monitors existing stubs verifying that they are still valid, i.e., the Rotor transparent proxies associated with them still exist. This is achieved using Rotor weak-references. A similar approach is used to monitor InProp and OutProp entries.

This approach has several advantages: i) it does not impose relevant modifications on the Rotor Kernel (CLR - Common Language Runtime), ii) it can be implemented using a high-level language such as C#, iii) modifications are mainly restricted to the Remoting package, and iv) it does not interfere with the LGC used.

The data structures representing stubs, scions, InPropLists and OutPropLists are all implemented as hashtables so that when needed, they are accessed efficiently.

6.2 Remoting Code Instrumentation

Remoting services code instrumentation intercepts messages sent and received by processes in the context of object propagations (performed resorting to remote invocation), so that scions, stubs, InProps and OutProps are created accordingly.[8]

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

Code implementing ADGC algorithm's explicit messages is grouped in a specific class *DGCManager*; this code runs as a low priority thread in each application process, and is responsible for composing and sending *NewSetStubs*, *Unreachable* and *Reclaim* messages lazily. *NewSetStubs*, *Unreachable* and *Reclaim* messages from other processes, are delivered when well-known remote methods made available by DGCManager are invoked by another process.

6.3 Graph Description Summarization

In order to be consistent, object graph serialization must be performed while the application code is not running. In the current implementation, this is performed immediately after the LGC[9] and before allowing the application to proceed. However, this object graph serialization is needed only for cycle detection. Thus, it only needs to be seldom done. This allows the serialization of the ob-

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

[9] This needs to be performed not necessarily after every LGC. This operation may be only seldom performed, e.g., once out of every 10 (or 100) LGC executions.

ject graph to be done in other more convenient situations, such as when the application stops waiting for input, or is idle. In particular, it does not have to be created every time an LGC occurs. It can be further optimized with operating system support; e.g., using copy-on-write on graph pages, serialization can be performed lazily in the background with minimal delay, extra memory, and processor load.

Graph summarization is coded in C#. It is performed, lazily and incrementally, in each process, after a new object graph has been serialized, by a separate thread (which is almost always blocked). Summarization of object graphs stored persistently in files is performed by an off-line process executing the same function: it transverses the graph, breadth-first, in order to minimize time overhead (i.e., re-tracing of objects). Once summarized, graph information becomes atomically available to the DCDP.

6.4 Cycle Detection

The DCDP performing GMS is implemented in C# both in Rotor[39] and OBIWAN[40, 41, 13]. The choice of the implementation language of the cycles detector had no constraints, provided that inter-operability could be fulfilled between the DCD and the graph serialization/summarization component in each process.

Cycle detection is implemented as a standard mark and sweep algorithm abiding to the marking rules presented earlier (namely, one that enforces the Union Rule). Any mark-and-sweep algorithm could be used (as long as it respects the marking rules). Therefore, we do not provide more details about its implementation.

7 Performance Evaluation

The most relevant performance results of our implementation are those related to phases critical to applications performance: i) stub/scion creation common to any acyclic DGC (corresponding to the implementation of safety rules **Clean Before Send Propagate** and **Clean Before Deliver Propagate** presented in Section 3.4), and ii) snapshot serialization. These phases could delay and potentially disrupt the mutator, therefore applications. Results were obtained using a Pentium 4 Mobile 1600Mhz with 512 Mb RAM.

We measured the creation of stubs and scions when remote references are exported/imported, resulting from the propagation of replicas. For safety, these operations are always performed and cannot be fulfilled lazily. We tested worst case scenarios, discarding potentially long network communication times, that could mask stub/scion creation overhead. Figure 6 and Table 1 show results for increasing series of object propagations carrying 10 references (10 different references being exported/imported), where client and server processes execute in the same machine. This forces the ADGC to create 10 scions and stubs each time a propagation takes place. The overhead associated with the creation of stubs and scions, in this worst case scenario without communication delay, is within 7%-21% which is acceptable for the functionality provided, i.e., a safe DGC (and not one that is lease-based).

# of propagations	Rotor	Rotor w/ DGC	Variation
10	1933	2072	7.19%
100	12417	14731	18.64%
500	58754	70931	20.73%
1000	118890	140191	17.92%

Table 1: Propagation latency (in ms) due to ADGC management of remote references (including relative variation).

The results regarding snapshot creation (by means of serialization of the object graph) were very bad on Rotor (see Figure 7). We must stress, though, that these operations do not have to be performed frequently. As a micro-benchmark, we used graphs with 10000 linked dummy objects (just holding a reference). Rotor serialization takes on average 26037 ms. To serialize the same graph, with every object containing an additional remote reference (additional 10000 stubs), it takes 45125 ms (73% more). This portrays a very conservative scenario, concerning the number of outgoing inter-process references: one in that each object holds a unique remote reference. In normal circumstances, the number of remote references in a process, is several orders of magnitude lower than the number of local references.

Nevertheless, serializing a remote reference is faster than serializing an additional dummy object and, therefore, the impact of serializing GC structures is lower than that of objects. However, these rather un-encouraging results are a direct consequence of the very inefficient serialization code (for any purpose) included in Rotor. We think this is intentional as Microsoft considers several

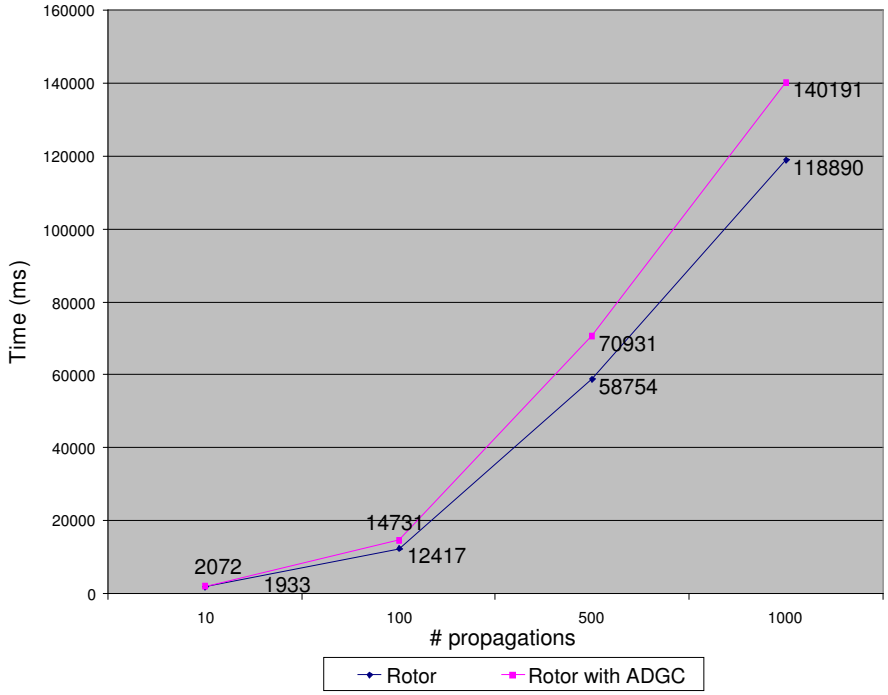


Figure 6: Propagation latency due to ADGC management of remote references.

aspects of the .Net CLR (Common Language Runtime) as commercial product critical code, namely serialization and LGC.

To address these limitations, we re-implemented the algorithm (the same ADGC, with the same code for DCDP, on OBIWAN[40, 41, 13] at user-level), so that it runs on top of the commercial version of .Net. In this second implementation, with production-level .Net serialization code, serialization times are subject to a speed-up of approximately 100 times. These results are more encouraging (see Figure 8). They range from 250ms to 350ms. This imposes significantly shorter pause times. Moreover, these operations needs to be performed only sporadically.

8 Related Work

Given that the DGC algorithms presented in this article provide a complete approach to the problem of DGC for replicated object systems, the solutions can be related to a large number of work performed in the area of garbage collection. As a matter of fact, DGC has been a mature field of study for many years and there is extensive literature [28, 1, 37] about it. However, given that our main contribution addresses the issue of detecting and reclaiming cycles of garbage for replicated object systems, we focus on previous work dealing either with DGC algorithms in replicated environments (which are not complete), or with algorithms for collecting distributed cycles of garbage, (i.e., algorithms that are complete) in non-replicated environments. Note that, as there is no previous work addressing the detection and reclamation of garbage cycles in a replicated system, this approach covers the all spectrum of existing solutions that may have some aspect that can be compared to our DCD algorithm.

8.1 ADGC for Replicated Environments

DGC for replicated objects was first addressed by Ferreira [7, 8] in the framework of the Larchant project [9, 10, 11]. This work proposed a new DGC algorithm based on a set of safety rules that take into account the existence of replicated objects, thus solving the problem illustrated in Figure1. In particular, the Union Rule then introduced, ensures the safety of the DGC algorithm. However, the solution then proposed is not scalable because it requires the underlying communication layer to support causal delivery. In addition, it is not complete as it fails to detect

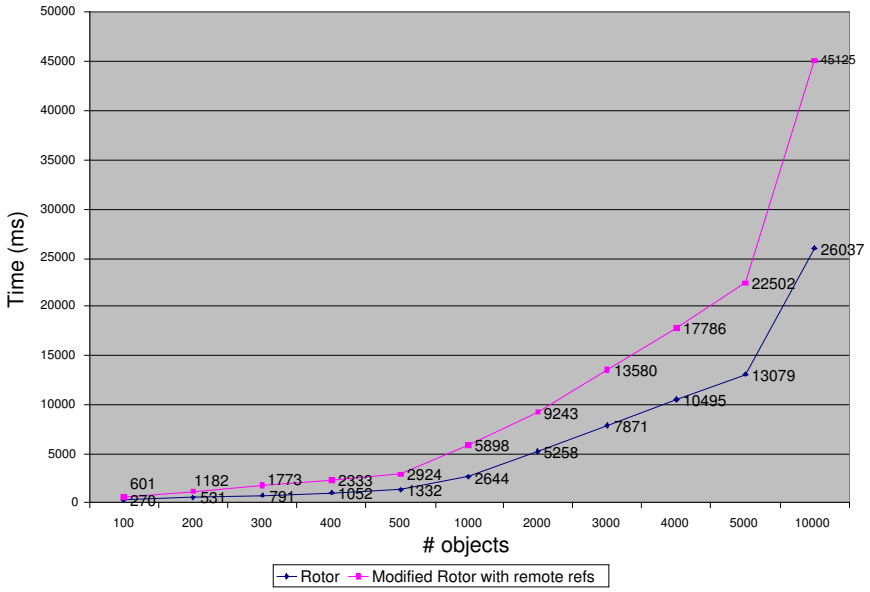


Figure 7: Test-graph serialization times in Rotor: local references only, and with custom-serialization of proxy objects for remote references.

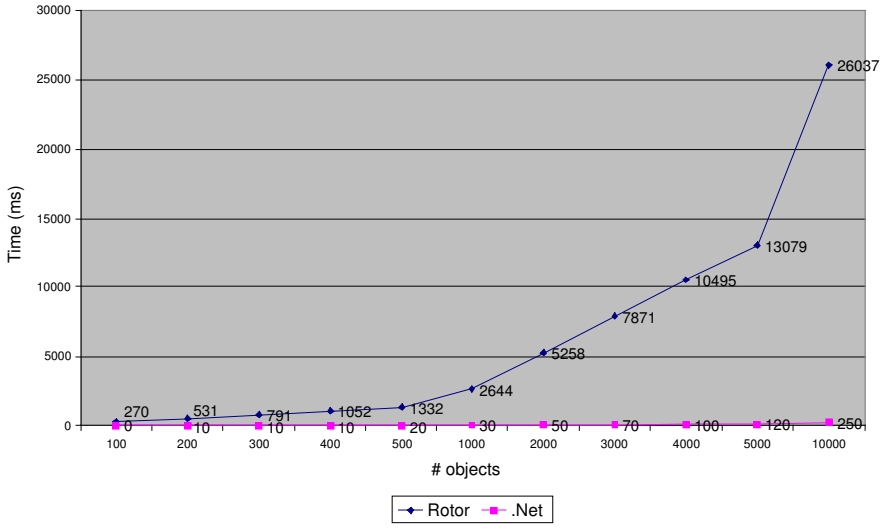


Figure 8: Test-graph serialization times in Rotor and in .Net commercial version.

and reclaim cycles of garbage.

The PerDiS project [5, 34, 12] improved on previous work so that it no longer required the underlying communication layer to support causal delivery. This was achieved by means of a new set of rules that conservatively creates the scion-stub pair of an inter-process reference immediately before being exported/imported. However, this algorithm is also not complete, i.e. it does not reclaim cycles of garbage. This algorithm has also been used to enforce referential integrity and minimize storage waste in web content replication systems [43].

Our current work, concerning the ADGC described in this article, improves on the mentioned above algorithms in the sense that it is integrated with the DCD, thus making a complete solution for the GC in object replicated systems.

The work described in [48] is related to ours given that it addresses the issue of DGC for a distributed shared memory system (TreadMarks [17]). They propose a conservative collector that uses partitioned garbage collection on a process basis; this approach is the same used by many others. All messages exchanged between processes are scanned for possible contained pointers; this is similar to the safety rules mentioned above w.r.t. the DGC in Larchant. However, their solution is specific to TreadMarks, i.e. it is not widely applicable. In addition, it does not reclaim cycles of garbage.

8.2 Cyclic DGC for non-Replicated Environments

Global propagation of time-stamps until a global minimum can be computed was first proposed in [16] to detect distributed cycles of garbage. Distributed garbage collection based in cycles detection within groups of processes was first introduced in [19]. These algorithms are not scalable since they require a distributed consensus by the participating processes on the termination of the global trace. This is also impossible in the presence of faults [15].

Migrating objects to a single process in order to convert a distributed cycle into a local one, that is traceable by a basic LGC, was suggested by several others in [4, 23]. Object migration, for the sole purpose of GC, is a heavy requirement for a system, needs extra and possible lengthy messages (bearing the actual objects) among participating processes. It is very difficult to accurately select the appropriate process that will contain the entire cycle. Cycles that span many objects, copied into a single process in charge of tracing may cause overload.

In another centralized approach, distributed garbage collection is performed by a logically centralized server [21] that receives graph information from every process. The centralized server performs complete distributed garbage collection and informs processes of objects to delete. Requirements on clock synchronization and message latency are strict making this solution unscalable.

The work presented in [46] proposes trial deletion to detect distributed cyclic garbage. It uses a separate set of reference count fields for trial deletion in each object. These count fields are used to propagate the effect of trial (simulated) deletions. Trial deletion starts on an object suspect of belonging to a distributed cycle. The algorithm simulates the recursive deletion of the candidate object and all its referents. When, and if the trial counts of every object of the sub-graph drop to zero, a distributed cycle has been successfully found. It imposes the use of reference counting for LGC (which is seldom chosen); this is an important limitation. The recursive freeing process is unbounded. Furthermore, it has problems with mutually referencing distributed cycles of garbage.

In [24], distributed backtracing starts from suspected objects (of belonging to a distributed cycle of garbage), and stops until it finds local roots or when all objects leading to the suspect have been backtraced. There are two mutually recursive procedures: one to perform local backtracing and another is in charge of remote backtracing. Distributed backtracing results in a direct acyclic chaining of recursive remote procedure calls, which is clearly unscalable. To ensure termination and avoid looping during backtracing, each *ioref* (representing remote references) must be marked with a list of trace-id's to remember which backtraces have already visited it. This requires processes to keep state about detections on course which raises questions of fault-tolerance. Local back-tracking is performed with resort to optimized structures similar to our graph summarization mechanism. To ensure safety, reference copies (local and remote) must be subject to a transfer-barrier that updates *iorefs*. The distributed transfer barrier may need to send extra messages that are guarded against delayed delivery.

Distributed backtracking is also used in [33] for cycle detection in CORBA. As in our work, it addresses detailed issues about implementation of this concept in a real environment/system with off-the-shelf software.

In [31], groups of processes are created to be scanned as a whole and detect cycles exclusively comprised within them. Groups of processes can also be merged and synchronized so that ongoing detections can be reused and combined. It has fewer synchronization requirements w.r.t. [19, 32]. When a candidate is selected, two strictly ordered distributed phases must be performed to trace objects. Mark-red phase paints the distributed transitive closure of the suspect objects with the color red. This must be performed for every cycle candidate. Termination of this phase creates a group. Afterwards, the scan-phase is started independently in each of the participating processes. The scan-phase ensures un-reachability of suspected objects. Objects also reachable from other clients (outside the group) are marked green. This consists of alternating local and remote steps. The cycle detector must inspect objects individually. This demands strong integration and cross-dependency with the execution environment and the local garbage collector. Mutator requests on objects are asynchronous w.r.t. GC; when this happens during scan-phase, to ensure safety, all of an object descendants may need to atomically be marked green, which blocks application when it is actually mutating objects. As in [24], GC structures need to store state about all ongoing detections passing through them.

In [14], marks associated both with stubs and scions are propagated between sites until cycles are detected. Marks are complex holding three fields (distance, range and generator identifier) and an additional color field. Local roots first, and then scions, are sorted according to these marks. Stubs require two marks. Objects are traced twice every time the LGC runs (with important performance penalty to applications) starting from local roots and scions: first in decreasing, and then in increasing order of marks, towards stubs. Mark propagation through objects to the stubs is decided by min-max marking (this is heavier than simply reach-bit propagation). One message propagates marks from stubs to scions.

Cycle detection is started by generators that propagate marks, initiating in local roots and scions recently created or touched by the mutator. When a remote invocation takes place, a new generator is created and its associated mark must be propagated along the downstream distributed sub-graph. Generator records include creation time, a range field and a locator of the mark generator. White marks represent pure marks while gray marks indicate mixing of marks from different generators during a local trace. When a generator receives back its own mark, colored white, a cycle has been detected. If the mark is gray, it means other paths lead to the scion and sub-generations must be initiated. Stub messages need to include, besides marks, additional information about every single sub-generator reaching each stub. Sub-generators are created in the back-trace of the generator that receives the gray mark. This lazy back-tracking mechanism can be very slow. An optimistic variation leverages knowledge about sub-generators triggering several back-traces in different processes. Possible errors are prevented resorting to a special black color associated with marks in scions whose sub-generator status is later revised.

The resulting global approach to cycle detection is achieved at the expense of additional complexity and performance penalties. It imposes a specific, longer, heavier LGC that must collaborate with the cycle detector. There is a tight connection and dependency among LGC, acyclic DGC and cycles detection. This is inflexible since each of these aspects is subject to optimization in very different ways, and should not be limited by decisions about the others. The mark propagation consists of a global task being continuously performed; it has a permanent cost. Instead it should be deferred in time, and executed less frequently.

Relevant mutator events (i.e., edge-creation and edge-destruction), and causal dependencies among them, are monitored in [22] to perform DGC. Only edge-creation and destruction involving remote references are relevant. Objects targeted by remote references are designated global roots. Analysis of mutator events is used as an alternative to tracing the distributed object graph. Each relevant event equates to lazily sending a control message (either create or destroy) with respective direct dependency vector (DDV) that reflects causal dependencies on operations performed by other global roots. DDV are logged, merged with its causal predecessors, and propagated, until full vector-time is obtained for each global root. This enables calculation of the complete transitive closure of the graph. This approach is complete. It is resilient to message loss and duplication, and lazy message exchange avoids races and synchronization bottlenecks. However, it has unbounded latency for all garbage detection (not just for cyclic garbage) and increased space overhead. The use of vector clock logs, for each global root, makes this algorithm unscalable.

Our management of unsynchronized summarized graph descriptions can be related to GC-consistent-cuts in databases as proposed in [38]. In this work, a GC-

consistent-cut has one or more copies of every page in the database. These copies, possibly inconsistent from a transactional point of view, can be created at different instants. However, all these pages, when combined with knowledge from database locks, may be consistently and safely used for LGC purposes. This may require at least as much memory as the database data itself. This work can be applied only to a centralized database system, it is not distributed, and is strongly dependent of the specific information provided by the database synchronization mechanisms.

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

GC-consistent-cuts were used before, in the context of distributed systems, in [42]. However, the solution presented is not replication-aware, i.e., it is not safe in the presence of replicated objects. A de-centralized version of this algorithm is presented in [44, 45]. It introduces algebra-based cycle detection by forwarding probe messages (piggy-backed on acyclic DGC messages) that carry an algebraic representation of a distributed path being transversed. This algebra comprises all the objects that the message has been sent to (called *targets*), and unresolved dependencies (*sources*). The latter constitute objects still un-visited by the probe message, but that may still maintain the whole cycle reachable.

Upon message reception, each process applies a matching predicate to the received algebra. If it succeeds, a distributed garbage cycle has been found. If not, it may either forward an updated version of the algebra to other processes (containing target objects) or terminate detection (e.g., when local roots are reached, a mutator-cycle detector race is detected, or an identical probe message has already been forwarded through the same path). State concerning multiple cycle detections in progress is kept exclusively in each of algebra-carrying probing messages. The algorithm does not need to store information in processes regarding each specific on-going detection. This helps scalability and robustness.

8.3 Additional Remarks

Design and implementation of complete GC on single-site partitioned object stores is thoroughly addressed in [25]. GC collects one partition at-a-time. Inter-partition references must be managed, and inter-partition cycles can occur, as inter-site in DGC. However, there are no issues of distribution to address here.

Recently, in [27] a complete proof of safety and liveness is provided for *Birrel's* reference listing algorithm [2, 3] on which Java DGC is based. Although this algorithm is not complete, as it cannot detect distributed cycles of garbage objects, it is arguably the most widely deployed DGC algorithm.

Another example of usage of snapshots in distributed object stores, while completely unrelated to GC, appears in [26]. It enables efficient system archiving and allows safe computation over earlier system states.

In summary, our approach is the first to address memory management for replicated object systems, in a comprehensive manner. It presents the first DGC algorithm for these systems, that is complete, i.e., that can detect and reclaim distributed cycles of garbage comprised of replicated objects spanning several processes. It has few requirements on synchronization avoiding disruption to mutator and intrusion to LGC. Furthermore, it has been implemented on realistic off-the-shelf systems.

9 Conclusions

We presented a comprehensive solution to address memory management for replicated object systems. To the best of our knowledge, we presented the first distributed garbage collection algorithm that is both safe in the presence of replication, and complete w.r.t. distributed cycles of garbage comprising replicated objects.

The main contributions of our work are: i) a novel distributed cycles detector algorithm that is able to reclaim distributed cycles comprising replicated objects. It does not require global synchronization, it is scalable, and it is not intrusive w.r.t. mutator and LGC. It is able to make progress without requiring all processes to participate, ii) the notion of DGC-consistent cut applied to wide area replicated objects, and iii) an implementation on Rotor and on .Net with minimum impact on the source code of Rotor runtime.

In comparison with previous work, our approach, while being complete and scalable, is more flexible. In fact, it imposes fewer and lighter restrictions w.r.t. synchronization among processes, state management at each process about detections in course, and intrusion with the

mutator and with the LGC. Thus, it is specially adequate for realistic systems with off-the-shelf software. This fact is confirmed by our implementation on Rotor and using .Net.

Finally, although we have implemented the ADGC and DCD algorithms in Rotor and OBIWAN, our solutions are rather general. It is possible to apply the same ideas and, in particular the notion of the CG-consistent-cut and the DCDP, to other platforms supporting object replication.

In the future, we plan to address the following subjects:

- investigate how the implementation can be further optimized, namely w.r.t. graph summarization (possibly integrating it with incremental LGC provided with the VM)
- as an alternative (yet, complemental) direction, w.r.t. the previous item, we also wish to maximize portability, possibly at the cost of hurting performance. This way, we want to investigate placement of the all GC-related code in custom proxies, static classes, and custom attributes (with associated code) applied to application classes. This would allow that applications with and without replication and/or cyclic GC can co-exist without changing applications neither the VM.
- addressing the formal correctness proof of the ADGC and DCD algorithms .
- development of a de-centralized version of the algorithm presented here.

References

[1] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: a survey of distributed garbage collection. *ACM Computing Surveys (CSUR)*, 30(3):330–373, 1998.

[2] Greg Nelson Susan Owicki Edward Wobber Andrew Birrel, David Evers. Distributed garbage collection for network objects. Technical Report 116, Digital Systems Research Center, 1993.

[3] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 217–230, New York, NY, USA, 1993. ACM Press.

[4] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR-178.

[5] Xavier Blondel, Paulo Ferreira, and Marc Shapiro. Implementing garbage collection in the perdis system. In *Proc. of the Eigth Int'l W'shop on Persistent Object Systems: Design, Implementation and Use (POS-8)*, 1998.

[6] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Comm. of the ACM*, 21(11):966–975, November 1978.

[7] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, California (USA), November 1994. ACM.

[8] Paulo Ferreira and Marc Shapiro. Garbage collection of persistent objects in dist. shared memory. In *Proc. of the 6th Int'l W'shop on Persistent Object Systems*, Tarascon (France), September 1994. Springer-Verlag.

[9] Paulo Ferreira and Marc Shapiro. Garbage collection in the larchant persistent dist. shared store. In *Proc. of the 5th W'shop on Future Trends in Dist. Computing Systems*, Cheju Island (Republic of Koreia), August 1995. IEEE.

[10] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in dist. shared memory through garbage collection. In *Int'l Conf. on Dist. Computing Systems (ICDCS'96)*, Hong Kong, May 1996. IEEE.

[11] Paulo Ferreira and Marc Shapiro. Modelling a dist. cached store for garbage collection: the algorithm and its correctness proof. In *ECOOP'98, Proc. of the Eight European Conf. on Object-Oriented Programming*, Brussels (Belgium), July 1998.

[12] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, Jo ao Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Robert, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. PerDiS: design, implementation, and use of a PERsistent DIstributed Store. *Recent Advances in Dist. Systems, Springer Verlag LNCS, Eds. S. Krakowiak and S.K. Shrivastava*, 1752, February 2000.

[13] Paulo Ferreira, Luís Veiga, and Carlos Ribeiro. Obiwan - design and implementation of a middleware platform. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1086–1099, November 2003.

[14] Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Conference on Principles of Distributed Computing(PODC)*, 2001.

[15] M. Fisher, N. Lynch, and M. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):274–382, April 1985.

[16] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985. Springer-Verlag.

[17] P. Keleher, A. Cox, and W. Zwaenepoel. TreadMarks: Dist. shared memory on standard workstations and operating systems. *Proc. of the 1994 Winter USENIX Conf.*, January 1994.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] Bernard Lang, Christian Quenniac, and José Piquer. Garbage collecting the world. In *Conf. Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 39–50. ACM Press, January 1992.

[20] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[21] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 29–39, Vancouver (Canada), August 1986. ACM.

[22] Sylvain R.Y. Louboutin and Vinny Cahill. Comprehensive dist. garbage collection by tracking causal dependencies of relevant mutator events. In *Proc. of ICDCS'97 Int'l Conf. on Dist. Computing Systems*. IEEE Press, 1997.

[23] Umesh Maheshwari and Barbara Liskov. Collecting cyclic dist. garbage by controlled migration. In *Proc. of PODC'95 Principles of Dist. Computing*, 1995. Later appeared in Dist. Computing, Springer Verlag, 1996.

[24] Umesh Maheshwari and Barbara Liskov. Collecting cyclic dist. garbage by back tracing. In *Proc. of PODC'97 Principles of Dist. Computing*, 1997.

[25] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In *Proc. of SIGMOD'97*, 1997.

[26] Chuang-Hue Moh and Barbara Liskov. Timeline: A high performance archive for a distributed object store. In *Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[27] Luc Moreau, Peter Dickman, and Richard Jones. Birrell's distributed reference listing revisited. Technical Report 8–03, University of Kent, Canterbury, July 2003. (later accepted for publication in ACM Transactions On Programming Languages And Systems in 2004).

[28] David Plainfossé and Marc Shapiro. A survey of dist. garbage collection techniques. In *Proc. Int. W'shop on Memory Management*, Kinross Scotland (UK), September 1995. http://www-sor.inria.fr/SOR/docs/SDGC_-iwmm95.html.

[29] David S. Platt. *Introducing the Microsoft.NET Platform*. Microsoft Press, 2001.

[30] Nicolas Richer and Marc Shapiro. The memory behavior of the WWW, or the WWW considered as a persistent store. In *POS 2000*, pages 161–176, 2000.

[31] Helena Rodrigues and Richard Jones. Cyclic distributed garbage collection with group merger. *Lecture Notes in Computer Science*, 1445, 1998.

[32] Helena C. C. D. Rodrigues and Richard E. Jones. A cyclic dist. garbage collector for Network Objects. In Ozalp Babaoglu and Keith Marzullo, editors, *Tenth Int'l W'shop on Dist. Algorithms WDAG'96*, volume 1151 of *Lecture Notes in Computer Science*, Bologna, October 1996. Springer-Verlag.

[33] Gustavo Rodriguez-Rivera and Vince Russo. Cyclic distributed garbage collection without global synchronization in corba. In *OOPSLA'97 GC & MM Workshop*, 1997.

[34] Alfonso Sanchez, Luís Veiga, and Paulo Ferreira. Dist. garbage collection for wide area replicated memory. In *Proc. of the Sixth USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio (USA), January 2001.

[35] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, dist. references and acyclic garbage collection. In *Symp. on Principles of Dist. Computing*, pages 135–146, Vancouver (Canada), August 1992. ACM.

[36] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, dist. references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de Recherche en Informatique et Automatique, Rocquencourt (France), November 1992. http://www-sor.-inria.fr/SOR/docs/SSPC_rr1799.html.

[37] Marc Shapiro, Fabrice Le Fessant, and Paulo Ferreira. Recent advances in distributed garbage collection. *Lecture Notes in Computer Science*, 1752:104, 2000.

[38] M. Skubiszewski and P. Valduriez. Concurrent garbage collection in O2. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proc. of 23rd Int'l Conf. on Very Large Databases*, pages 356–365, Athens, 1997. Morgan Kaufman.

[39] David Stutz. The microsoft shared source cli implementation. MSDN Library Article, Microsoft Corporation, march 2002.

[40] Luís Veiga and Paulo Ferreira. Incremental replication for mobility support in OBIWAN. In *The 22nd International Conference on Distributed Computer Systems*, pages 249–256, Viena (Austria), July 2002.

[41] Luís Veiga and Paulo Ferreira. Mobility support in OBIWAN. In *2nd Microsoft Research Summer Workshop*, Cambridge (UK), Sep 2002.

[42] Luís Veiga and Paulo Ferreira. Complete distributed garbage collection, an experience with rotor. *IEE Research Journals - Software*, 150(5), oct 2003.

[43] Luís Veiga and Paulo Ferreira. Repweb: Replicated web with referential integrity. In *18th ACM Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA, Mar 2003.

[44] Luís Veiga and Paulo Ferreira. Asyncronous, complete distributed garbage collection. Technical report rt/11/2004, INESC-ID Lisboa, june 2004.

[45] Luís Veiga and Paulo Ferreira. Asynchronous complete distributed garbage collection. In *19th IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, USA, april 2005.

[46] S. C. Vestal. *Garbage collection: an exercise in distributed, fault-tolerant programming*. PhD thesis, Seattle, WA, USA, 1987.

[47] Paul Wilson. Distributed garbage collection general discussion for faq. GCList Mailing List (gclist@iecc.com), march 1996.

[48] Weimin Yu and Alan Cox. Conservative garbage collection on dist. shared memory systems. In *The 16th Int'l Conf. on Dist. Computer Systems*, pages 402–410, Hong-Kong, May 1996.