Asynchronous Complete Garbage Collection for Graph Data Stores

Luís Veiga, Rodrigo Bruno, Paulo Ferreira
INESC-ID / Instituto Superior Técnico, University of Lisbon
luis.veiga,rodrigo.bruno,paulo.ferreira@inesc-id.pt

ABSTRACT

Graph data stores are a popular choice for a number of applications: social networks, recommendation systems, authorization and control access, and more. Such data stores typically support both distribution and replication of vertexes across physical nodes.

While distribution provides better load balancing of requests, replication is necessary to achieve improved availability and performance. However, most of these systems still manage replicated memory by hand, resulting in expensive efforts to fix dangling references and memory leaks to ensure referential integrity.

In this paper, we present a novel Garbage Collection (GC) algorithm that safely eliminates both acyclic and cyclic garbage with support for replicated data. Our algorithm provides minimum impact on applications' performance, is completely decentralized and has no coordination requirements. We evaluate our approach against previous solutions and show that our solution is efficient while imposing very little overhead on applications' performance.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: ProcessorsMemory management; C.2.4 [**Computer Systems Organization**]: Distributed SystemsDistributed Applications

General Terms

Algorithms, Performance, Reliability

Keywords

Graph, Memory Management, Garbage Colleciton, Shared Memory

1. INTRODUCTION

Graph data stores or object-oriented based systems (e.g. Java) are used by many companies to solve a variety of problems (e.g. Ehcache (ehcache.org), Hazelcast (hazelcast.com), Terracotta (terracotta.org), etc.). The ability to scale while providing responses to complex semantic queries, the powerful expressiveness, and its simplicity are just some of the many features offered by most graph data stores implementations.

MIDDLEWARE '15, December 07-11, 2015, Vancouver, BC, Canada
©2015 ACM. ISBN 978-1-4503-3618-5/15/12 ... $15.00
DOI: http://dx.doi.org/10.1145/2814576.2814813

To accomplish these features, such systems rely on: i) distribution (to balance requests of multiple clients), and ii) replication (to improve availability and performance).

However, over time, replicated garbage (i.e., objects replicated (or cached) in several nodes that are no longer accessible) is produced by either deleting some outdated client information or updating it with new information. In both cases, objects that could be accessed previously are now inaccessible (and therefore, are considered garbage). As the amount of garbage grows, application's performance decreases. This results from two main factors: extreme memory utilization which could result in swapping objects from memory, and frequent local garbage collections (where the local garbage collector tries to free memory for the application).

To successfully mitigate this problem, i.e., to allow applications to run efficiently, a proper garbage collection (with support for object replication) must be employed. This poses some important requirements: i) all garbage must be reclaimed, meaning that both acyclic and cyclic garbage must be reclaimed; ii) memory contention must be avoided by timely collecting garbage; iii) the overhead of garbage collecting on applications' performance must be minimal.

In order to attain such goals, the garbage collector has to travel the replicated distributed graph and analyze which objects (which could be replicas of other objects) are accessible or not by any application root reference[1]. This operation must be done with minimal interference so as to avoid slowing down applications.

Not considering manual memory management, which is difficult and often results in dangling references (references to objects prematurely deleted) and memory leaks (references to objects that are lost and therefore, the objects are never freed), most current solutions (e.g. [7, 8, 2, 13, 2, 23, 3]) do not address the issue of distributed garbage collection with replication. Thus, if applied to a replicated scenario, such solutions either do not scale (as they consider replicas to be as any other object) or are not safe reclaiming live objects erroneously thus breaking referential integrity. In

[1]A root reference is a special reference held by the runtime system which points to root objects. Examples of root objects are variables stored in registers and global variables.

addition, often they impose a global centralized consensus.

We propose a solution that, is both scalable, complete, and supports object replication. As others before, it follows a hybrid approach: an acyclic GC based on replication-aware reference-listing, and a distributed cycle detector that complements the first, thus providing a complete solution for the problem of collecting garbage.

The algorithm for acyclic GC is based on reference-listing [20] and is adapted to work with replicated objects. Therefore, this algorithm keeps track of inter-processes references and object propagations (replication of objects to remote nodes) and uses a set of safety rules (described in detail in Section 3) to safely reclaim garbage.

The cycles detection algorithm works on object graph snapshots taken by each process independently (i.e., with no synchronization required at all). There is an instance of the cycle detector for each one of such processes. Thus, each snapshot is treated by the detector independently; messages are then exchanged among processes so that certain reference and propagation paths are followed in order to find if they form a cycle of garbage. Thus, there is no central server responsible for detecting and reclaiming garbage cycles. Such detection is done by a set of messages that are sent and then return to the origin, using an algebra, that allows to detect if a garbage cycle has been detected; such a cycle is then broken (by deleting a remote reference) which allows its reclamation by the local GCs of each concerned process.

In sum, the main contribution of this work is a novel distributed garbage collection algorithm which supports distributed and replicated objects, requires no synchronization, and is complete (detects both acyclic and ciclic garbage). Section 5 provides a set of performance results which confirm that our solution has low network and application overhead for detecting garbage.

We envision that this work could be used in real-world systems such as graph databases and distributed shared memory systems (e.g. those previously mentioned) to improve garbage collection safety (not relying on manual memory management), performance, and completeness while ensuring referential integrity. In other words, the algorithms proposed here can be used in a large-scale graph database, for example, to safely and efficiently delete sub-graphs that got disconnected from the main graph, and therefore are no longer accessible from the application logic. This can happen if the application replaces old information or simply deletes it.

In the next section, we describe the underlying graph data model assumed (which is rather general), and the acyclic solution (which does not consider cycles of garbage) for detecting replicated objects garbage. These are then complemented with the asynchronous algorithm for detecting cycles of garbage (for replicated objects) described in Section 3. Section 4 describes the most relevant aspects of the implementation, and Section 5 shows the evaluation results. The paper ends with a section on related work (Section 6), and conclusions (Section 7).

2. BACKGROUND

Before going into the main algorithm, we give some background on two important topics which are base building blocks for the rest of this solution. First, we explain the graph data model, how it operates and most importantly, which kind of systems are equivalent to it. Second, we present an acyclic GC with support for replicated objects. This algorithm is used in our complete GC to collect acyclic garbage.

2.1 Graph Data Model

We now describe a model for graph data, the environment for which our GC is conceived. The model assumes the existence of vertexes and directed edges connecting vertexes. Vertexes can be distributed and/or replicated across multiple physical nodes.

This model is actually used by a number of real-world systems, for example: i) graph databases such as Titan or Neo4J; ii) memory caches/shared memory systems such as Terracotta BigMemory, Infinispan, and Hazelcast.

It is important to note that the graph data model is equivalent to Replicated Memory (RM) model. In such model, edges are mapped to memory objects and vertexes are mapped to object references (which can be local or remote). Hence, the problem of finding garbage in distributed and replicated graphs is equivalent to finding garbage in a RM system. For the rest of this document we refer to our data model as Replicated Memory (RM).

Each participating process in a RM system encloses the following entities: memory, mutator (application), 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.

2.1.1 Memory Organization

Applications can have different views of objects and can see them as language-level class instances, memory pages, data base records, web pages, graph vertexes etc. The unit for replication is the object. Any object can be replicated in any process. Each process can hold a replica of any object for reading or writing according to the coherence protocol being used. This does not preclude the possibility of replicating several objects in a single operation; it simply does not impose it.

The only operation executed by mutators that is relevant for GC purposes is reference assignment; this is the only way for applications to modify the graph of objects. This may result on some object becoming globally 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.

2.1.2 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.

Ideally, the DGC (distributed garbage collector) should be orthogonal w.r.t. whatever mechanism is used to maintain replicas coherent. Thus, the only coherence operation, which is considered relevant for DGC purposes, is the propagation of an object, i.e. the replication or update (which may carry new references) of an object from one process to another.

The propagation of an object is performed using a propagate message that carries the actual object content. When an object is propagated to a process, 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, the object's enclosed references are imported.

We assume that any process can propagate a replica from another process into itself, when 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.

In a RM system mutators may create inter-process references very easily and frequently, through a simple reference assignment operation. 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 that reference to the 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.

In each process, the coherence engine must hold 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. Usually, this information already exists in the coherence engine in order to manage the replicas. Thus, they are not a special requirement imposed by DGC, though their existence is leveraged by it.

Each entry in these lists indicates the process from which each object has been propagated, and the processes to which each object has been propagated, respectively. 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 propObj has been propagated;
- sentUmess/recUmess - bit indicating if a *Unreachable* message has been sent or received. *Unreachable* messages refer to unreachability from GC local-roots (more details in Section 2.2).

It is worthy to note that in the RM model, the only way a process can create inter-process references is through the execution of two operations: (i) reference assignment, which is performed explicitly by the mutator and (ii) object propagation, which is performed by the coherence engine in order to allow the mutator to access some object.

As an 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. objects.

2.2 Acyclic GC for RM

This section describes a solution for detecting RM acyclic garbage. The overall solution for the problem is constituted by the following algorithms: i) a local tracing-based garbage collection (LGC) algorithm running in each process and ii) a replication-aware reference-listing acyclic distributed garbage collector (ADGC) algorithm (based on previous work [19]). The LGC and ADGC algorithms depend on each other to perform their job, as explained afterwards.

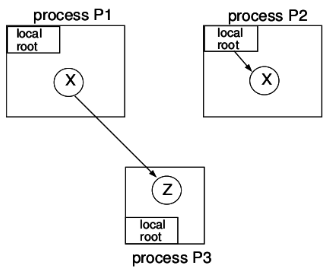

Figure 1: Safety problem of current DGC algorithm which do not handle replicated data: Z is erroneously considered unreachable.

Four important data structures are used by both algorithms (LGC and ADGC):

- Stub - a stub describes an outgoing inter-process reference, from a source process to a target process;
- Scion - a scion describes an incoming inter-process reference, from a source process to a target process;
- inPropList - list of entries specifying a local object and a remote process from which the object was propagated;
- outPropList - list of entries specifying a local object and a remote process to which the object was propagated.

2.2.1 Replication Awareness

To safely collect a single object X (in a replicated distributed object model), one has to guarantee that no application root reference (that can be local or remote) is able to access (directly or indirectly) X. If X or any object found in any path that leads to X is replicated, then all replicas must be also checked for reachablility. This is known as the Union Rule (introduced in Larchant [4]): *a target object Z is considered unreachable only if the union of all the replicas of the source objects do not refer to it.*

Consider Figure 1 in which an object X is replicated in processes P1 and P2. Now, suppose that X_{P1} (this notation is used from here on to refer an object, X in this case, and its corresponding process, P1 in this case) contains a reference to an object Z in another process P3, X_{P1} points to no other object and is not locally reachable. X_{P2} is locally reachable. Then, the question is: should Z be considered garbage? Classical DGC algorithms (designed for non-replicated systems) consider that Z is effectively garbage. However, this is wrong because, in a replicated 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 locally reachable in process P1 does not mean that X is globally unreachable; 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 replicated distributed systems, garbage collection algorithms must enforce the Union Rule.

2.2.2 Local Garbage Collector

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 root references and set of scions. For each outgoing inter-process reference it creates a stub in the new set of stubs (used in next step). Once this tracing is completed, every object locally reachable by the mutator has been found (e.g. marked, if a mark-and-sweep algorithm is used); objects not yet found are locally unreachable; 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 object graph (marking the objects found) from the lists inPropList and outPropList, and performs as follows: i) when a locally reachable 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.

2.2.3 Acyclic DGC

From time to time, possibly after a local collection, the ADGC sends a message *NewSetStubs* to the processes holding the scions corresponding to the stubs in the previous stub set (i.e., the processes to which it has remote references). This message contains the new set of stubs that resulted from the local collection. In each of the receiving processes, the ADGC matches the received new 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:

1. For an object, which is reachable only form the inPropList, a message *Unreachable* is sent to the site from where that object has been propagated. When a *Unreachable* message reaches a process, this event is registered in the corresponding outPropList entry.

2. For an object 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 processes to which that object has been propagated and the outPropList is deleted. When a process receives a *Reclaim* message, it forwards it to other processes that propagated that object from the current process and the corresponding entries from the outPropList and inPropList are deleted.

In summary, besides the *NewSetStubs*, two other messages may be sent by the ADGC: *Unreachable* and *Reclaim*. These messages result in the modification of local data structures that are checked by the LGC (when it performs local collection). Thus, ultimately, the LGC is the one that collects objects.

2.2.4 Object Propagation

In a replicated system, mutators may create inter-process references very easily and frequently, through a simple reference assignment operation (see Section 2.1.1). 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 2.1.2.

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 exit 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 (in parallel with the mutator).

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. ASYNCHRONOUS COMPLETE GC FOR RM

We now describe our cycle collector for RM which, in conjunction with the acyclic presented in Section 2.2, provides a complete DGC for RM.

As explained before (in Section 2.2.1), in order to safely reclaim garbage in RM, one has to ensure that an object can not be accessed neither locally nor remotely from any root. This includes normal object references and object replicas (Union Rule).

The problem gets harder when garbage objects reference each other, creating a garbage cycle. In such scenario, objects are remotely reachable (since the LGC sees an incoming reference the the object) but are not globally reachable, i.e., no local or remote root can access the object.

Figure 2 shows a cycle of garbage in RM (solid lines represent object references while dashed lines represent object propagations). Note that X'$_{P2}$ is a replica of X$_{P1}$ and Y'$_{P3}$ is a replica of Y$_{P4}$. If the GC is not replication-aware, X'$_{P2}$ and Y$_{P4}$ would automatically be considered garbage. However, only if none of the replicas of an object are reachable, it is safe to collect the object. The acyclic DGC described before would not be able to detect garbage because objects have dependencies, either an object reference (scion) or object propagation (inProp or outProp).

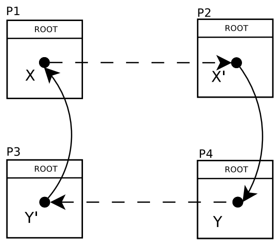

Figure 2: Cycle of garbage in RM. Solid lines represent object references. Dashed lines represent object propagations.

In a complete replication-aware GC, the DGC tracks references and propagations until i) an object is locally reachable (and therefore, there is no garbage cycle) or ii) all references and propagations have been followed and no globally reachable object was found (a cycle is detected). In Figure 2, propagations from X_{P1} and Y_{P4} would be tracked and a cycle would be detected comprising all four objects.

3.1 Overview

We now give an overview of our cycle detector for RM garbage. Consider that object X_{P1} (Figure 2) is suspected to be garbage (efficient selection of cycle candidates is out of the scope of this paper; heuristics found in literature [14] may be used). The cycle detector analyzes four aspects: i) if the object is accessible through another remote object (it will find that X_{P1} is accessible through Y'_{P3}); ii) if the object references other remote objects; iii) if the object is propagated to another process (it will find that X_{P1} is propagated to P2); iv) if the object is propagated from another process.

With this information, the cycle detector concludes that X_{P1} can not be garbage if Y'_{P3} is not garbage and that X_{P1} was propagated to P2. In order to continue the cycle detection, a cycle detection message (from here on called CDM) is sent to process P2 containing the information already discovered in P1.

The cycle detector in P2 finds a reference from X'_{P2} to Y_{P4} and a new CDM (containing the information discovered so far) is sent to P4. From P4, a new CDM is sent to P3, where y was propagated to. In P3, a remote reference to X_{P1} is found. A new CDM is sent to P1. Then, the cycle detector realizes that X_{P1} is globally unreachable because: i) all replicas of X_{P1} are only accessible from Y'_{P3}; ii) all replicas of Y_{P4} are only accessible from X'_{P2}; iii) no object is locally accessible or has other incoming remote references.

3.2 Data Structures

The structures manipulated by the cycle detector are regular acyclic GC structures, extended with the following information (invocation counters and update counters are addressed in Section 3.5):

- Scion

  - Invocation Counter (IC): counter for concurrency purposes;

  - StubsFrom: list of stubs, in the same process, transitively reachable from the scion;

  - ReplicasFrom: list of replicated objects, in the same process, transitively reachable from the scion.

- Stub

  - Invocation Counter (IC): counter for concurrency purposes;

  - ScionsTo: list of scions, in the same process, that transitively lead to the stub;

  - ReplicasTo: list of replicated objects, in the same process, that transitively lead to the stub.

  - LocalReach: flag-bit accounting for local reachability (i.e., from the local root of the enclosing process) of the stub;

- inProp and outProp (entries of inPropList and outPropList respectively):

  - Update Counter (UC): counter for concurrency purposes;

  - ScionsTo: list of scions, in the same process, that transitively lead to the replcated object;

  - ReplicasTo: list of replicated objects, in the same process, that lead to the replicated object;

  - StubsFrom: list of stubs, in the same process, transitively reachable from the replicated object;

  - ReplicasFrom: list of replicated objects, in the same process, transitively reachable from the replicated object;

  - LocalReach: flag-bit accounting for local reachability (i.e., from the local root of the enclosing process) of the replicated object.

Each of these data structures is essential for the correct identification of garbage in a RM system as explained in the next section.

3.3 Algorithm

We now describe how our algorithm actually works. For this section, we assume that all mutators are stopped and therefore, while the GC is running, no new references or propagations are created. In Section 3.5 we relax this requirement.

Cycle detections use an algebraic representation encoded in the CDM. The CDM content is comprised of two sets (separated by $\rightarrow$): i) a source-set holding compiled dependencies, and ii) a target-set holding target objects that the message has been forwarded to. The first set (source-set) is subdivided into two dependency sets: propagation and reference dependencies. This algebraic representation is built using the data structures previously presented.

Our algorithm uses CDMs to travel the graph searching for garbage cycles. To guide the search, we take advantage of the fact that replicated objects evolve into a tree of replicas (parent replicas are replicated into child replicas). Similarly to the acyclic algorithm, we try to select references or child replicas available from the current replica before sending CDMs to parent replicas. This way, child replicas are traversed before their parents. Only when a child replica believes that it belongs to a distributed cycle of garbage, it forwards its CDM to its parent replica. After receiving the CDM from all child replicas the parent replica either launches a local cycle detection (if there are reference dependencies left) or instructs the acyclic detector that a cycle has been found.

Consider again the example of Figure 2. Let us assume that a detection is initiated with object X_{P1} as candidate (efficient selection of cycle candidates is an issue out of the scope of this paper; heuristics found in the literature [14] may be used).

The steps performed and relevant state are the following (the notation AlgN $\Rightarrow$ is used to present the contents of the Nth CDM):

1. P1: Alg0 $\Rightarrow$ $\{\{\}, \{X_{P1}\}\} \rightarrow \{\}$, ($X_{P1}$ chosen as candidate for cycle detection; it is the first reference dependency).
2. P1: *outPropList*(X_{P1}) $\Rightarrow$ $\{X'_{P2}\}$, (X'_{P2} found as a propagation of x in P1.
3. P1: Alg1 $\Rightarrow$ $\{ \{ X'_{P2} \}, \{ X_{P1} \} \} \rightarrow \{ \}$, resulting algebra from node P1.
4. P1: Send Alg1 to P2, (send CDM message).
5. P2: Deliver Alg1.
6. P2: *Matching* (Alg1) $\Rightarrow$ $\{\{X'_{P2}\}, \{X_{P1}\}\} \rightarrow \{\}$.
7. P2: *Cycle Found?* $\Rightarrow$ *false*

For each CDM delivered to a process, the cycle detector performs an algebraic matching (determining whether a distributed garbage cycle was detected): a cycle is found if all elements in the source set (including both sub-sets) appear in the target set. According to our example, in step #7, no cycle is found since there are at least two unresolved dependencies.

8. P2: Alg1 $\Rightarrow$ $\{ \{ X'_{P2} \}, \{ X_{P1} \} \} \rightarrow \{ \}$
9. P2: *StubsFrom*(X'_{P2}) $\Rightarrow$ $\{Y_{P4}\}$
10. P2: Alg2 $\Rightarrow$ $\{ \{ X'_{P2} \}, \{ X_{P1} \} \} \rightarrow \{ X'_{P2} \}$
11. P2: Send Alg2 to P4
12. P4: Deliver Alg2
13. *Matching*(Alg2) $\Rightarrow$ $\{\{\}, \{X_{P1}\}\} \rightarrow \{\}$
14. P4: *Cycle Found?* $\Rightarrow$ *false*

Again, no cycle is found since there are still references to account for.

15. P4: *outPropList*(Y_{P4}) $\Rightarrow$ $\{Y'_{P3}\}$
16. P4: Alg3 $\Rightarrow$ $\{ \{ X'_{P2}, Y'_{P3} \}, \{ X_{P1}, X'_{P2} \} \} \rightarrow \{ X'_{P2}, Y_{P4} \}$
17. P4: Send Alg3 to P3
18. P3: Deliver Alg3
19. P3: *Matching*(Alg2) $\Rightarrow$ $\{\{Y'_{P3}\}, \{X_{P1}\}\} \rightarrow \{\}$
20. P3: *Cycle Found?* $\Rightarrow$ *false*
21. P3: *StubsFrom*(Y'_{P3}) $\Rightarrow$ $\{X_{P1}\}$
22. P3: Alg4 $\Rightarrow$ $\{ \{ X'_{P2}, Y'_{P3}, Y_{P4} \}, \{ X_{P1}, X_{P2} \} \} \rightarrow \{ X'_{P2}, Y_{P4}, Y'_{P3} \}$
23. P3: Send Alg4 to P1
24. Deliver Alg4
25. P1: *outPropList*(X_{P1}) $\Rightarrow$ $\{X'_{P2}\}$
26. P1: Alg5 $\Rightarrow$ $\{ \{ X'_{P2}, Y'_{P3}, Y_{P4} \}, \{ X_{P1}, Y_{P4}, Y'_{P3} \} \} \rightarrow \{ X'_{P2}, Y_{P4}, Y'_{P3}, X_{P1} \}$
27. *Matching*(Alg5) $\Rightarrow$ $\{\{\}, \{\}\} \rightarrow \{\}$
28. P4: *Cycle Found?* $\Rightarrow$ *true*

At this moment, it is safe to assume that a cycle has been found and that object X_{P1} belongs to it. Therefore, it is safe to instruct the acyclic GC at P1 to delete the scion accounting for the remote reference to X_{P1}. After deleting the incoming reference, there is no longer a cycle of garbage and therefore the acyclic GC will, with the help of the LGC, target the other objects as garbage.

3.4 Multiple Detection Paths

In the previous example, only one object (X_{P1}) was leading the cycle detection and therefore, as soon as that object was found, the cycle was detected. However, in many situations, multiple cycle detection directions can be pursued. In this section, we describe one such example, where two different paths can be traversed for cycle detection.

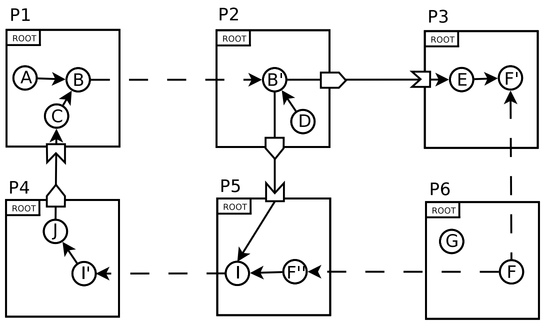

Figure 3: Graph of Garbage with Multiple Possible Detection Paths

Consider Figure 3 which comprises six processes (P1 to P6). No object in the graph is globally reachable meaning that all objects are candidates for collection. Obvious steps, similar to those in the previous example, are omitted for simplicity. Let us assume that detection starts at object C_{P1} . Then we have:

1. P1: Alg0 $\Rightarrow$ $\{ \{ \}, \{ C_{P1} \} \} \rightarrow \{ \}$
2. P1: *ReplicasFrom*(C_{P1}) $\Rightarrow$ $\{B_{P1}\}$
3. P1: *outPropList*(B_{P1}) $\Rightarrow$ $\{B'_{P2}\}$
4. P1: Alg1 $\Rightarrow$ $\{ \{ B'_{P2} \}, \{ C_{P1} \} \} \rightarrow \{ B_{P1} \}$, Send to P2
5. P2: *StubsFrom*(B'_{P2}) $\Rightarrow$ $\{E_{P3}, I_{P5}\}$

Two possible detection paths are discovered in step #5, one leading to E_{P3}, and other to I_{P5}. At this point, there is no way to decide which path is the best. Therefore, two CDMs are sent from P2, one for P3 and another for P5. Each of these messages will either: i) resolve all dependencies and find a garbage cycle; ii) reach nodes that can not resolve the remaining dependencies and abort the detection track.

6. P2: $Alg2_a$ $\Rightarrow$ $\{ \{ B'_{P2}, B_{P1} \}, \{ C_{P1} \} \} \rightarrow \{ B_{P1}, B'_{P2} \}$, Send to P3
7. P2: $Alg2_b$ $\Rightarrow$ $\{ \{ B'_{P2}, B_{P1} \}, \{ C_{P1} \} \} \rightarrow \{ B_{P1}, B'_{P2} \}$, Send to P5

Following the CDM sent to P5 we have:

8. P5: *ReplicasTo*(I_{P5}) $\Rightarrow$ $\{F''_{P5}\}$
9. P5: *outPropList*(I_{P5}) $\Rightarrow$ $\{I'_{P4}\}$
10. P5: $Alg3_b$ $\Rightarrow$ $\{ \{ B'_{P2}, B_{B1}, I'_{P4} \}, \{ C_{P1}, F''_{P5}, B'_{P2} \} \} \rightarrow \{ B_{P1}, B'_{P2}, I_{P5} \}$, Send to P4

Note that, although F''_{P5} is a replicated object, it is not included in the replica dependency set, but instead, it goes to the reference dependency set. This is because we did not traverse this object, we only know that it references an object (I_{P5}) being checked for garbage.

11. P4: *StubsFrom*(I'_{P4}) $\Rightarrow$ $\{C_{P1}\}$
12. P4: $Alg4_b$ $\Rightarrow$ $\{ \{ B'_{P2}, B_{P1}, I'_{P4}, I_{P5} \}, \{ C_{P1}, F''_{P5}, B'_{P2} \} \} \rightarrow \{ B_{P1}, B'_{P2}, I_{P5}, I'_{P4} \}$, Send to P1
13. P1: *ReplicasFrom*(C_{P1}) $\Rightarrow$ $\{B_{P1}\}$
14. P1: $Alg5_b$ $\Rightarrow$ $\{ \{ B'_{P2}, B_{P1}, I'_{P4}, I_{P5} \}, \{ C_{P1}, F''_{P5}, B'_{P2} \} \} \rightarrow \{ B_{P1}, B'_{P2}, I_{P5}, I'_{P4}, C_{P1} \}$
15. P1: *Matching*($Alg5_b$) $\Rightarrow$ $\{\{\}, \{F''_{P5}\}\} \rightarrow \{\}$
16. P1: *Cycle Found?* $\Rightarrow$ false

Since B'_{P2} is already in the target set of $Alg5_b$ and the matching returns unresolved dependencies (F''_{P5}), this cycle detection track is stopped. This mechanism forces messages that cannot resolve more dependencies to stop looping through the graph.

Now, we follow the other cycle detection track from step #6. Note that the step count is individual for each CDM so we now go for step #7:

7. P3: *ReplicasFrom*(E_{P3}) ⇒ {F'_{P3}}

8. P3: *inPropList*(F'_{P3}) ⇒ {F_{P6}}

9. P3: $Alg3_a$ ⇒ { { B'_{P2}, B_{P1}, F_{P6} }, { C_{P1}, E_{P3} } } → { B_{P1}, B'_{P2}, E_{P3}, F'_{P3} }, Send to P6

The CDM reaches a point (in step # 9) where there are no other out-going references to follow. However, there is an unresolved replica dependency (F_{P6}) which could still close the cycle. Therefore, $Alg3_a$ is forwarded to process P6.

10. P6: *outPropList*(F_{P6}) ⇒ {F''_{P5}}

11. P6: $Alg4_a$ ⇒ { { B'_{P2}, F_{P6}, F''_{P5}, F'_{P3} }, { C_{P1}, E_{P3} } } → { B_{P1}, B'_{P2}, E_{P3}, F'_{P3}, F_{P6} }, Send to P5

12. P5: *ReplicasFrom*(F''_{P5}) ⇒ {I_{P5}}

13. P5: *ScionsTo*(I_{P5}) ⇒ {B'_{P2}}

14. P5: *outPropList*(I_{P5}) ⇒ {I'_{P4}}

14. P5: $Alg5_a$ ⇒ { { B'_{P2}, F_{P6}, F''_{P5}, F'_{P3}, I'_{P4} }, { C_{P1}, E_{P3}, B'_{P2} } } → { B_{P1}, B'_{P2}, E_{P3}, F'_{P3}, F_{P6}, F''_{P5}, I_{P5} }, Send to P4

15. P4: *StubsFrom*(I'_{P4}) ⇒ {C_{P1}}

16. $Alg6_a$ ⇒ { { B'_{P2}, F_{P6}, F''_{P5}, F'_{P3}, I'_{P4}, I_{P5} }, { C_{P1}, E_{P3}, B'_{P2} } } → { B_{P1}, B'_{P2}, E_{P3}, F'_{P3}, F_{P6}, F''_{P5}, I_{P5} I'_{P4} }, Send to P1

Note that $Alg6_a$ has the same exact dependencies information as $Alg5_a$. This happens because C_{P1} was already included as a dependence for the cycle detection.

17. P1: *ReplicasFrom*(C_{P1}) ⇒ {B_{P1}}

18. $Alg7_a$ ⇒ { { B'_{P2}, F_{P6}, F''_{P5}, F'_{P3}, I'_{P4}, I_{P5} }, { C_{P1}, E_{P3}, B'_{P2} } } → { B_{P1}, B'_{P2}, E_{P3}, F'_{P3}, F_{P6}, F''_{P5}, I_{P5}, I'_{P4}, C_{P1} }

19. P1: *Matching*($Alg7_a$) ⇒ {{}, {}} → {}

15. P1: *Cycle Found?* ⇒ true

After matching algebra $Alg7_a$, the cycle detector can now conclude that all dependencies are solved and therefore, a cycle has been found. As before, it is enough for the cycle detector to instruct the acyclic GC to delete the scion of C_{P1} which will result in the safe collection of the whole cycle of garbage.

While many other graphs can be created, the fundamental operations to detect any other cycle using our algorithm were expressed in this example.

3.5 Dealing With Concurrency

We now detail our approach to allow mutators concurrently modify the distributed graph. Our approach is based on previous work [23] which is only safe for non-Replicated Memory systems. Therefore, we now provide a full solution that supports our RM data model.

Obviously, assuming that all mutators are suspended is not reasonable. So, periodically, each process stores a snapshot of its internal object graph on disk. This snapshot is performed by each process with no coordination w.r.t. other processes; thus, each process is completely independent.

If we assume that a set of such snapshots, taken independently by each process, provides a consistent view of the global distributed object graph, the cycle detector may proceed exactly as described previously. However, for obvious reasons, such an assumption is not correct; so, the cycle detector has to ensure that the set of snapshots visited by CDMs is, in fact, a consistent view for the purpose of finding distributed cycles of garbage.

Therefore, it is only required that the sub-graph being independently traced, to determine if it is a distributed garbage cycle, is observed consistently. 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) replicated cyclic garbage is always preserved by the acyclic GC (that is why we need a special detector), i.e., if the cycle detector does nothing, it still is safe.

Thus, we define CDM-Graph(x) as a consistent view restricted to the distributed sub-graph, headed by object x, enclosed in the correct combination of N process snapshots with N-1 CDMs. Cycle detection proceeds as the CDM-Graph is being constructed, i.e., with each CDM sent to a process, combined with its snapshot and, after update, sent to another process. Thus, a CDM carries a consistent view of the fraction of the CDM-Graph already traversed by it, i.e., the processes the CDM has been sent from. When a CDM-Graph is safely and completely constructed, with all dependencies resolved, a distributed garbage cycle has been detected.

3.5.1 Graph Summarization

Object graphs in application processes may be very large. Consequently, the size of the corresponding snapshot may contribute to increase detector complexity and occupy 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 (a snapshot) of each application process in such a way that, from the point of view of the cycle detector, there is no loss of relevant information. This summarization transforms a snapshot of an application graph into a set of scions, stubs, inProps, and outProps, with their corresponding associations.

This summarization is performed on every snapshot; then it is made available to the cycle detector. Thus, while processes can take snapshots by serializing local graphs, the cycle detector only uses them in their summarized form, i.e., after graph summarization.

3.5.2 Race Condition Invariants

For cycle detection purposes, a CDM-Graph must respect the following invariant: there can be no invocations or replica updates/propagations along the distributed sub-path to be included in the CDM-Graph. If we allow this to happen, it means that the mutator or the coherence engine is modifying the distributed graph in the back of the cycle detector. Consequently, the cycle detector may erroneously conclude that it found a garbage cycle. Figure 4-a illustrates such a case. The initial situation is that of a cycle formed by objects X_{P1}, X'_{P2}, Y_{P3}, and Y'_{P4}. This cycle is not garbage because X_{P1} is referenced from the local root in P1.

Now consider the sequence of events depicted in the timeline in Figure 5 (S1, S2, S3, and S4 are the moments when the corresponding processes make their snapshots). The cycle detector starts in P2 by sending a CDM to P3. Concurrently, the coherence engine (which manages replica updates) issues an update from P3 to P1. This update results in a new remote reference from X_{P1} to Y_{P4}. Right after the update operation, there is a remote invocation (from X_{P1} to

Y_{P4}), which creates a local root pointing to Y_{P4}. When the invoke returns, the local root at P1 (X_{P1}) is deleted. After deleting the local root, P1 makes a snapshot of its graph (S1).

Given that S2, S3, and S4 were previously taken, the view of the distributed graph that is perceived by the cycle detector instances (i.e., the CDM-Graph) is, in fact, the one represented in Figure 4-b, instead of the correct one represented in Figure 4-c. This would lead to the erroneous detection of a distributed cycle of garbage comprising objects X_{P1}, X'_{P2}, Y_{P3} and Y'_{P4}. This erroneous conclusion would be reached by the cycle detector if the invariant mentioned before is not respected.

In this case, a replica update and an invocation took place along the reference path P1 $\rightarrow$ P2 $\rightarrow$ P3 that had been previously stored in the snapshot and will be included in the CDM-Graph when the CDM arrives to P1.

The invariant dictating the construction of a CDM-Graph is implemented using the following conservative safety rules (Situation $\rightarrow$ Action), when process snapshots are pairwise-combined through CDMs:

1. Stub/outProp without corresponding Scion/inProp (snapshot of the process holding the Scion/inProp is not current enough for the CDM-Graph) $\rightarrow$ Ignore CDM;
2. Scion/inProp without corresponding Stub/outProp (reference creation message in transit, acyclic garbage, or snapshot of the process holding the stub not current enough) $\rightarrow$ The CDM is never sent since there is no stub in CDM-Graph;
3. Stub/outProp with matching Scion/inProp but there have been remote invocations/replica updates, and possibly reference copying, along the CDM-Graph after one of the snapshots was taken; it is not consistently accounted for in the snapshot and the CDM) $\rightarrow$ Terminate CDM-Graph construction, i.e., terminate detection avoiding mutator-cycle detector race;
4. Stub/outProp with corresponding Scion/inProp and there were no invocations or replica updates after snapshot (safe to continue CDM-Graph creation and detection) $\rightarrow$ Proceed CDM-Graph construction, combine CDM with process snapshot and continue detection.

There are two straightforward ways to uphold CDM-Graph invariant w.r.t. the last two rules: i) pessimistic: to freeze the mutator and the coherence engine in, or deny it access to, the path already traversed while detection is in course, or ii) optimistic: to detect, at a later stage, that this invocation has indeed occurred. The first option is clearly undesirable as it disrupts applications with no justification (if the mutator wants to access objects, they are clearly not garbage). The second option allows the application to run at full-speed at the expense of possibly wasting some detection work (an hypothetical distributed cycle may be partially or completely traversed by the detector, only to find out that meanwhile, a distributed invocation on that cycle has taken place). The algorithm needs only to ensure safety in these cases (and it does) since they must be infrequent when efficient heuristics are used to select cycle candidates. Thus, the solution conceived consists on a barrier that detects invocations and replica updates being performed in the back of the cycle detector.

3.5.3 Mutator-Cycle Detector Race

Following the example in Figure 4 and the timeline in

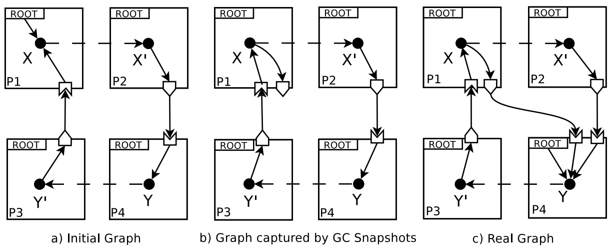

Figure 4: Possible Graph States in a Race Condition

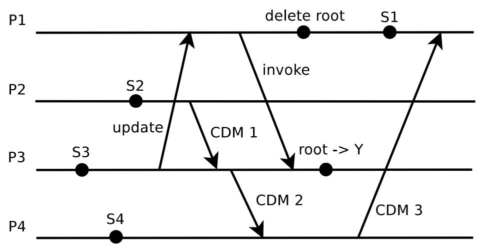

Figure 5: Possible Cycle Detection Timeline in a Race Condition

Figure 5, we now describe how the cycle detector is able to detect mutator activity and stop the cycle detection preserving the safety of the algorithm.

Table 1 contains the invocation and update counters for all GC data structures involved in the process. Three phases are presented: i) initial, which corresponds to the beginning of the timeline in Figure 5; ii) snapshot, corresponding, in each process, to the moment when the snapshot is taken; iii) real, the final graph state, which corresponds, to the end of the timeline.

For simplicity, assume that, in the initial state, all counters have the same value α. Note that both α and β can take any value. We only assure that α causally precedes $\alpha + 1$. Cells filled with a dash represent a data structure counter that did not exist in the corresponding graph phase.

According to the timeline in Figure 5, the cycle detection starts in process P2. We now follow the algebra in each of the messages:

1. P2: *StubsFrom*(X'_{P2}) $\Rightarrow$ $\{Y_{P4}\}$
2. P2: CDM 1 $\Rightarrow$ $\{\{X'_{P2}\},\{\}\} \rightarrow \{\}$, Send to P4
3. P4: *outPropList*(Y_{P4}) $\Rightarrow$ $\{Y'_{P3}\}$
4. P4: CDM 2 $\Rightarrow$ $\{\{X'_{P2}, Y'_{P3}\}, \{X'_{P2}\}\} \rightarrow \{Y_{P4}\}$, Send to P3
5. P3: *StubsFrom*(Y'_{P3}) $\Rightarrow$ $\{X_{P1}\}$
6. P3: CDM 3 $\Rightarrow$ $\{\{X'_{P2}, Y'_{P3}, Y_{P4}\}, \{X'_{P2}\}\} \rightarrow \{Y_{P4}, Y'_{P3}\}$, Send to P1
7. P1: *outPropList*(X_{P1}) $\Rightarrow$ $\{X'_{P2}\}$
8. P1: *outProp*(X'_{P2}) $\succ$ *inProp*(X_{P1})
9. P1: *Cycle Found?* $\Rightarrow$ *false, abort detection*

As it is possible to confirm from Table 1, *outProp*(X'_{P2}), from snapshot S1, succeeds *inProp*($_{P1}$), from snapshot S2. This invalidates the invariant presented before and therefore, the cycle detection is aborted.

4. IMPLEMENTATION

The algorithms (replication-aware reference-listing and cycle detector) were implemented combining C++ and C#.

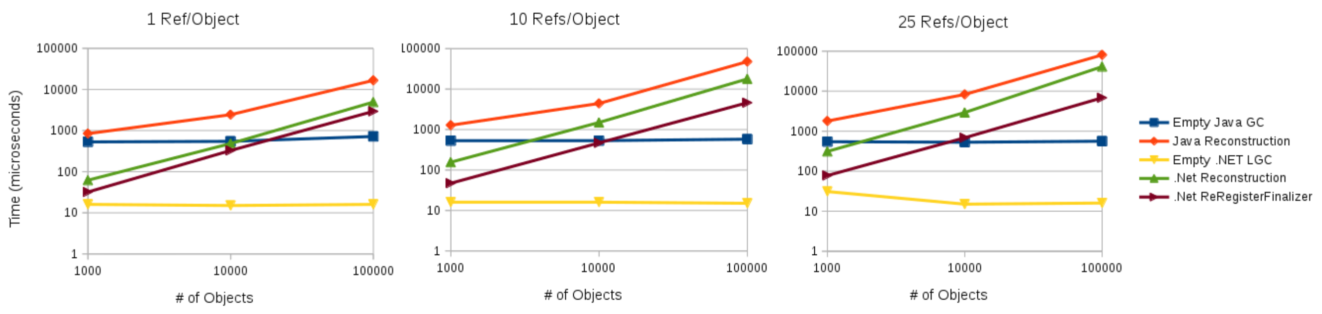

Figure 6: Tota LGC Overhead due to Enforcement of the Union Rule

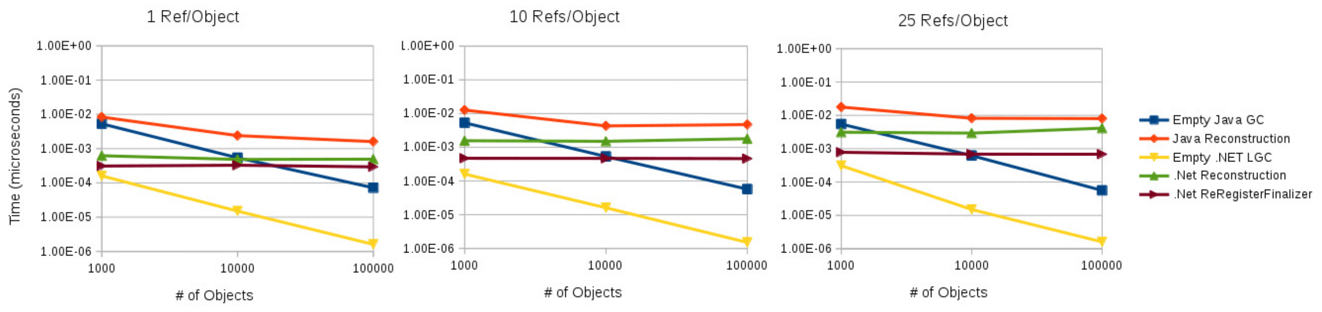

Figure 7: LGC Object Unitary Cost to Enforcement of the Union Rule

GC Structure	Initial	Snapshot	Real
P2:inProp(X_{P1})	α	α	$\alpha + 1$
P2:stub(Y_{P4})	α	α	α
P4:scion(X'_{P2})	α	α	α
P4:outProp(Y'_{P3})	α	α	α
P4:scion(X_{P1})	-	-	β
P3:inProp(Y_{P3})	α	α	α
P3:stub(X_{P1})	α	α	α
P1:scion(X_{P1})	α	α	α
P1:outProp(X'_{P2})	α	$\alpha + 1$	$\alpha + 1$
P1:stub(Y_{P4})	-	β	β

Table 1: GC Data Structures Counters

The implementation includes SSCLI[2] virtual machine modification (for LGC and DGC integration), remoting code instrumentation (to detect export and import of references and replicas), and distributed cycle detection.

The reference-listing algorithm must cooperate with the LGC, essentially, in two ways: i) the LGC must provide, in some way, the reference-listing algorithm with information about every remote object referenced or propagated to/from local objects; ii) the reference-listing algorithm must prevent the LGC from reclaiming objects that are no longer locally reachable but are target of incoming remote references or are propagated to/from other processes.

The approach consists simply on a running thread that monitors existing stubs, scions, inProps, and outProps, verifying that they are still valid. This is achieved using weak-references. This approach has several advantages: i) it does not impose relevant modifications on the CLR (Common Language Runtime) implementation, ii) it can be implemented using a high-level language such as C#, iii) modifications are mainly restricted to the Remoting package, and iv) it does not interfere with the LGC used.

Remoting services code instrumentation intercepts messages sent and received by processes in the context of remote invocation and object propagation so that all GC data structures are managed properly.

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 or, alternatively, by an off-line process. It transverses the graph, breadth-first, in order to minimize overhead (i.e., re-tracing of objects). Once summarized, graph information becomes atomically available to the cycle detector. CDM algebra matching is implemented in C# both in SSCLI.

We also implemented a RM simulator to run scalability tests of our algorithm. The simulator is implemented in CLOS and it simulates a real network of nodes. Each node contains objects which can reference and be referenced by local and remote objects. The simulator also supports object propagation, i.e., objects can be replicated in remote nodes. Asynchronous messages are used to simulate remote invocations between objects and object propagations.

Two algorithms are implemented on top of the simulator: i) a previous algorithm [23], which does not support RM; ii) the algorithm presented in Section 3. We modified the previous algorithm to support replicas in a trivial way: object

[2]Shared Source Common Language Infrastructure (SSCLI), previously codenamed Rotor, is an open-source implementation of Microsoft's Common Language Infrastructure, part of .NET.

propagations are transformed into two remote references, one from the original object to the new object and other from the new object to the original object. In other words, inPros are transformed into scions and outProps are transformed into stubs. This modified version of the algorithm supports RM and is both complete and safe.

It is interesting to note that both algorithms take the same amount of time to identify the cycle. This happens because both algorithms need to traverse at least once each node inside the cycle. The main difference between both algorithms is in how they conduct their graph traversal. Since our solution is replication-aware, we take advantage of such knowledge to perform smarter decisions regarding which paths to take. As shown in the next section, this results in substantial less network overhead in our approach.

5. EVALUATION

We evaluate our algorithms using both our prototype and a simulator and tackle two issues: i) local GC overhead, and ii) network overhead. These are the most relevant performance aspects given their relation to the (lack of) intrusiveness w.r.t. applications, and scalability, respectively.

5.1 Local GC Overhead

To assess the cost of enforcing the Union Rule in our system, we injected user code object finalizer methods (methods that are called when the object is no longer reachable from a local root). We did a series of experiments, with varying parameters. We used an Intel(R) Core(TM) i5 CPU 760 @ 2.80GHz with 8GB RAM, equipped with .Net Framework 4.5 and OpenJDK 1.7.

The experiments portray a worst-case scenario that provides an upper-bound to the penalties imposed. It is assumed that all objects in memory have been replicated from another process, and that they are continuously: i) being detected as unreachable locally (thus, their finalizer is executed), and ii) immediately made reachable to the mutator again (by means of a reference being imported).

When this happens, the object must be handled in order that, in the future, local un-reachability will be detectable again. This is achieved using the techniques : i) object reconstruction, and ii) re-registering objects for finalization. Thus, the experiments depict a worst-case scenario. In normal operation, this overhead is not imposed to all objects, nor for every execution of the LGC or object invocation. An object that is reachable locally imposes no additional overhead to LGC, attributable to the Union Rule. Similarly, an object already detected as unreachable locally, imposes no additional LGC overhead. The overhead occurs during the transitions between local reachability and unreachability. When an object replica becomes unreachable locally, its finalizer must be executed to resurrect the object and preserve it. When an object replica, that is unreachable locally, becomes reachable again (due to reference import), one of the two techniques described must be used. In real scenarios, these transitions happen to each object, only in a very small fraction of LGC executions. Moreover, this penalty would be completely masked if the content of the object replica is also being refreshed from another process across the network.

The results of the experiments are presented in Figures 6 and 7. Each value is expressed in milliseconds and is averaged over three separate runs. In each run, the LGC of the virtual machine was explicitly executed, in loop, 100 times. This is achieved by invoking System.gc(); System.runFinalization() in Java, and in C#, System.GC.Collect(); System.GC.WaitForPendingFinalizers(). The graphs depict total execution times and unitary cost per object, for increasing number of objects (1000,10000, and 100000), and references contained in each of them (1,10, and 25). All graphs are in logarithmic scale and contain five data series:

Empty Java LGC: Times regarding plain execution of Java LGC. In the first execution, the totality of objects are reclaimed. The remaining executions are thus empty. It provides a lower-bound on LGC execution-time in Java;

Java Reconstruction: Times regarding Java LGC and, after each LGC execution, finalizer code that performs reconstruction of all objects, and replacement of all internal references, with proxies;

Empty .Net LGC: Times regarding plain execution of LGC in .Net. It serves a purpose analogous to Empty Java LGC;

.Net Reconstruction: Times regarding .Net LGC combined with object reconstruction in finalizer code;

.Net ReRegisterFinalize: Times regarding .Net LGC and, after each LGC execution, finalizer code that simply re-registers the object for finalization, thus allowing the finalizer to be continuously executed, avoid additional object and proxy creation.

The results presented show that, although there is no support in existing LGC, in Java and .Net, to provide differentiated information regarding object reachability, it is feasible to enforce the Union Rule resorting to user-level code. The experiments evaluate the combined cost of the two operations required: i) detecting local un-reachability to preserve objects, and ii) ensure that when an object becomes reachable to the local mutator, again, it will be possible to detect local un-reachability again in the future (using object reconstruction and re-registration for finalization). Even in the worst-case scenario portrayed (both in frequency of the

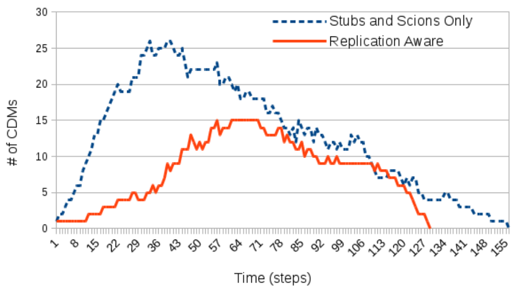

Figure 8: Number of CDMs per Simulation Step

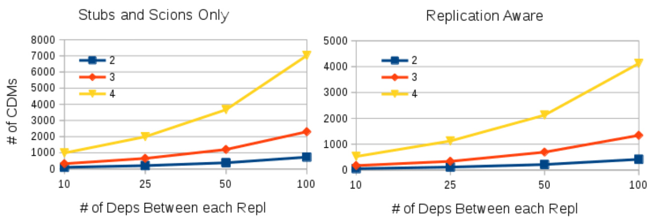

Figure 9: Number of issued CDMs

operation and absence of network communication), unitary costs are in the order of microseconds. Maximum values are of 25.4 in Java, and 14.5 for .Net, while the minimum are 6.32 for Java and 0.67 for .Net, which is actually lower than the unitary cost of Java in cases identified already.

5.2 Network Overhead

We now explore the network overhead of our RM cycle detector by simulating a real deployment of our algorithm in a distributed graph store. Hence, we use synthetic graphs replicated across multiple nodes (using our simulator). Each synthetic graph consists on a triangle mesh in which each triangle forms a cycle. This is, in fact, a worst case scenario, where the number of interconnected cycles is very high. Note that, however, for cycle detection purposes, only the number of remote references and object propagations will impact the number of CDMs needed to detect a cycle. Local references will be hidden by graph summarization as described before.

Therefore, we consider two variables: i) the number of replicated nodes and ii) the number of dependencies (remote references and object propagations) between each node which compose the distributed cycle. For example, with four replicated nodes and 100 dependencies, we have 4 physical nodes with 100 links to any of the other three physical nodes. All these links are connected in a large cycle of garbage which spans all 4 nodes. We compare our solution with a modified version of a previous algorithm [23] (described in Section 4).

Figure 8 presents the number of CDMs issued through the execution of a cycle detection on a graph with replication factor of 4 and 10 dependencies between each replica node. The solid line represents our solution while the dashed line represents the modified algorithm. Each simulation step represents a virtual time interval when processes can read incoming messages and compute outgoing messages.

Both algorithms identify the cycle after 51 simulation steps. However, our approach uses less CDMs through the cycle detection process. This happens because we distinguish between reference and propagation dependencies, allowing us to limit the amount of CDM flooding. Moreover, we forward CDMs when we are left with only propagation references. When forwarding CMDs, nodes do not have to compute a new CDM.

Figure 8 also shows that our solution stops traversing the network sooner (in a smaller number of steps). This has two important advantages: i) less CDMs means less overhead network traffic, and ii) less local application overhead (since each CDM involves some computation for integrating snapshots).

The performance of both algorithms if further assessed in Figure 9 which presents the number of CDMs using in a cycle detection when the amount of dependencies and the number of replicated servers increases from 2 to 4 and from 10 to 100, respectively. The number of simulation steps needed to identify the garbage cycle, in each graph, is presented in Table 2.

From Figure 9 we conclude: i) as the amount of dependencies increases, the network overhead also increases; ii) as the number of replicated nodes increases, the difference in network overhead between both solutions increases, i.e., the benefits from using our solution are more significant when we increase the number of replication nodes. This results from the fact that adding more replication to the graph increases the number of possible paths to search the graph. Since our solution limits the way the cycle detector selects new paths, our solution imposes much less network overhead.

Repl Factor / # Deps	10	25	50	100
2	25	55	105	205
3	38	83	158	308
4	51	111	221	411

Table 2: Number of Steps until Cycle Detection

6. RELATED WORK

The DGC algorithms presented in this article provide a complete approach to the problem of DGC for replicated object systems; thus, 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 [1, 15, 21] about it.

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 with the collection of distributed cycles of garbage. Note that most previous work addresses the detection and reclamation of garbage cycles in non-replicated system. Thus, we address existing decentralized solutions that may have some aspect that can be compared to our cycles reclamation algorithm.

Global propagation of time-stamps until a global minimum can be computed was first proposed in [7] to detect distributed cycles of garbage. Distributed garbage collection based in cycles detection within groups of processes was first introduced in [8]. These algorithms do not scale 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 [5].

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 [2, 12]. 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.

The work presented by Vestal [25] 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 Maheshwari [13], 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 [18] 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 Rodrigues [16], 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 re-used and combined. It has fewer synchronization requirements w.r.t. others [8, 17]. 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 [13], GC structures need to store state about all ongoing detections passing through them.

In Fessant [9], 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. 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.

The work by Veiga [23], although providing a complete distributed garbage collection, is also centralized, thus relying on a central server that periodically receives (summarized) graph snapshots from each site and performs a global mark-and-sweep. This solution raises scalability and reliability issues.

The work by Caromel [3] addresses the Grid and collects active objects (i.e. activities) with a complete DGC. Once again, replication is not considered. Cycles are collected by: i) considering the recursive closure of all the referencers of an active object, and ii) finding cycles of active objects waiting for requests. As in other solutions, this requires active objects to reach a consensus.

The pseudo root approach [26] addresses garbage collection for an actor system on the grid and requires a consistent global view of the system to collect distributed cycles. This solution is based on a centralized global garbage collector. The authors say that this is a concurrent, asynchronous, and non-FIFO solution. Such cycle collection is triggered periodically, and include some computing nodes from which it obtains a local snapshot; these are then merged to identify the garbage cycles, and the concerned nodes are informed.

In summary, detection of distributed cycles has been addressed with several solutions: (i) object migration, explicit [2] and via indirection [6] (train algorithm), (ii) trial deletion [25], (iii) propagation of marks or time-stamps, global [7, 11, 9], within groups [8, 16], (iv) distributed back-tracing [13, 18], (v) centralised detection, loosely-synchronised [10], asynchronous [24, 22], requiring a consensus [26], and (vi) cycle detection algebra [23].

Thus, in short, none of the above solutions addresses the issue of DGC with replication. If applied to a replicated scenario, such solutions either do not scale (as they consider replicas to be as any other object) or are not safe reclaiming live objects erroneously thus breaking referential integrity. As a matter of fact, our current approach described in this paper is the first to address memory management for replicated object systems, in a comprehensive and decentralized manner. It presents the first decentralized 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, and is decentralized, thus not relying on any central service.

7. CONCLUSIONS

In this paper we present a comprehensive solution to collect garbage in a generic RM system. Our solution is distibuted, does not require synchronization, and is complete. The evaluation results confirm that our algorithm does not hinder application performance (i.e., the LGC do not suffer significant overhead) and the amount of network messages is reduced compared to other approaches.

We envision that this work can improve current systems such as distributed caches, graph databases, or distributed shared memory systems (e.g. Ehcache, Hazelcast, Terracotta, etc.) by improving their support for automatically reclaim garbage.

Acknowledgments. This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference UID/CEC/50021/2013.

8. REFERENCES

[1] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373, Sept. 1998.

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

[3] D. Caromel, G. Chazarain, and L. Henrio. Garbage collecting the grid: A complete dgc for activities. In R. Cerqueira and R. Campbell, editors, *Middleware 2007*, volume 4834 of *Lecture Notes in Computer Science*, pages 164–183. Springer Berlin Heidelberg, 2007.

[4] P. Ferreira and M. Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 394–401. IEEE, 1996.

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

[6] R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage collecting the world: One car at a time. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 162–175, New York, NY, USA, 1997. ACM.

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

[8] B. Lang, C. Queinnec, and J. Piquer. Garbage collecting the world. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 39–50, New York, NY, USA, 1992. ACM.

[9] F. Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 200–209, New York, NY, USA, 2001. ACM.

[10] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '86, pages 29–39, New York, NY, USA, 1986. ACM.

[11] S. Louboutin and V. Cahill. Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 516–525, May 1997.

[12] U. Maheshwari and B. 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.

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

[14] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. *Distributed Computing*, 10(2):79–86, 1997.

[15] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 211–249, London, UK, UK, 1995. Springer-Verlag.

[16] H. Rodrigues and R. Jones. Cyclic distributed garbage collection with group merger. *Lecture Notes in Computer Science*, 1445, 1998.

[17] H. C. C. D. Rodrigues and R. E. Jones. A cyclic dist. garbage collector for Network Objects. In O. Babaoglu and K. Marzullo, editors, *Tenth Int'l W'shop on Dist. Algorithms WDAG'96*, volume 1151 of *LNCS*, Bologna, Oct. 1996. SV.

[18] G. Rodriguez-Riviera and V. Russo. Cyclic dist. garbage collection without global synchronization in CORBA. In P. Dickman and P. R. Wilson, editors, *OOPSLA '97 W'shop on Garbage Collection and Memory Management*, Oct. 1997.

[19] A. Sánchez, L. Veiga, and P. Ferreira. Distributed garbage collection for wide area replicated memory. In *Proceedings of the 6th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 6*, COOTS'01, pages 61–76, Berkeley, CA, USA, 2001. USENIX Association.

[20] M. Shapiro, P. Dickman, and D. Plainfossé. Ssp chains: Robust, distributed references supporting acyclic garbage collection. 1992.

[21] M. Shapiro, F. L. Fessant, and P. Ferreira. Recent advances in distributed garbage collection. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, pages 104–126, London, UK, UK, 1999. Springer-Verlag.

[22] L. Veiga and P. Ferreira. Complete distributed garbage collection: an experience with rotor. *Software, IEE Proceedings -*, 150(5):283–290, Oct 2003.

[23] L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 24a–24a. IEEE, 2005.

[24] L. Veiga, P. Pereira, and P. Ferreira. Complete distributed garbage collection using DGC-Consistent cuts and .NET AOP-support. *IET Software*, 1(6):263–279, Dec. 2007.

[25] S. C. Vestal. *Garbage Collection: An Exercise in Dist., Fault-Tolerant Programming.* PhD thesis, University of Washington, Seattle, WA, 1987.

[26] W.-J. Wang and C. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In Y.-C. Chung and J. Moreira, editors, *Advances in Grid and Pervasive Computing*, volume 3947 of *Lecture Notes in Computer Science*, pages 360–372. Springer Berlin Heidelberg, 2006.