Asynchronous Complete Distributed Garbage Collection

Luís Veiga Paulo Ferreira

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

Abstract

Most Distributed Garbage Collection (DGC) algorithms are not complete as they fail to reclaim distributed cycles of garbage.

Those that achieve such a level of completeness are very costly as they require either some kind of synchronization or consensus between processes. Others use mechanisms such as backtracking, global counters, a central server, distributed tracing phases, and/or impose additional load and restrictions on local garbage collection. All these approaches hinder scalability and/or performance significantly.

We propose a solution to this problem, i.e., we describe a DGC algorithm capable of reclaiming distributed cycles of garbage asynchronously and efficiently. Our algorithm does not require any particular coordination between processes and it tolerates message loss.

We have implemented the algorithm both on Rotor (a free source version of Microsoft .Net) and on OBIWAN (a platform supporting mobile agents, object replication and remote invocation); we observed that applications are not disrupted.

1. Introduction

Distributed garbage collection is a requirement for distributed object systems. In these systems, cycles are frequent [14]. In addition, when considering persistence, distributed garbage simply accumulates over time degrading performance. This is not simply an issue of disk space. Other aspects like storage management, object loading on primary memory, object marshalling, etc. suffer performance degradations with the extra load imposed by the increase of garbage.

Thus, the design of complete distributed garbage collection (DGC) algorithms is a problem that has been addressed many times before. However, the solutions so far proposed suffer from serious drawbacks, i.e., they require either some kind of synchronization[15] or consensus[7, 8] among processes, or use some other mechanism such as distributed backtracking[6, 11, 16] (possibly optimistic[4]), global counter tresholds[7], a central server processing[9], object migration[2, 10], or impose additional load on the local or acyclic distributed garbage collection[4]. All these approaches hinder scalability and/or performance significantly (more detail in Section 5).

We propose a solution that, as others before, follows a hybrid approach: an acyclic distributed collector based on reference-listing and a cycle detector that complements the first thus providing a complete solution for the problem of DGC.

The algorithm for acyclic DGC is based on reference-listing [17]. This algorithm keeps track of inter-processes references by means of data structures called stubs and scions. 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.

Each process has a local garbage collector (LGC). Root objects include global variables and threads stack. Starting from local roots and scions, the LGC generates a new set of stubs each time it runs. This new set of stubs is then sent to remote processes (this message is called NewSetStubs); these processes, based on the set of stubs received, may conclude which scions are no longer reachable so that they can be safely deleted. Objects that are only reachable through these, just deleted, scions are garbage and can be reclaimed by the next LGC.

The distributed cycles detection algorithm (DCDA) works on object graph snapshots taken by each process independently (i.e., with no synchronization required at all). There is an instance of the DCDA for each one of such processes. Thus, each snapshot is treated by the DCDA independently; messages are then exchanged among DCDAs so that certain reference paths are followed in order to find if they form a distributed cycle of garbage.

Thus, the main contribution of this work is a novel DCDA which is well integrated with traditional acyclic distributed collectors. Our solution does not require global synchronization and does not disrupt applications.

The rest of the paper is organized as follows. The next section provides an intuitive description of the DCDA. Section 3 describes the DCDA in detail using some prototypical examples. Section 4 presents the implementation and performance evaluation. In Sections 5 and 6 we describe relevant related work and final conclusions.

2. Distributed Cycle Detection

Distributed garbage collection identifies *live* and *dead* objects in distributed systems. To do so, it manages reachability information (*local, remote* and *global*) about objects. An object is *locally reachable* when it is transitively reach-

Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)
1530-2075/05 $ 20.00 **IEEE**

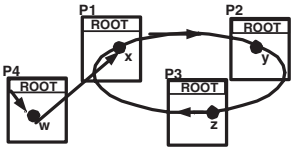

Figure 1. Identifying dependencies in cycles.

able from a local root (e.g., global variables, threads stack) of its enclosing process. An object is *reachable remotely* when it is referenced by other object(s) in different process(es). *Globally reachable* objects are all *live* objects, i.e., all the objects that can be manipulated by applications (in GC terms, *mutators*). Clearly, all objects *locally reachable* are also *globally reachable*. Objects solely *reachable remotely* may be either *globally reachable* or not. The previous are transitively reachable (through a chain of remote references) from a local root of some other process. The latter constitute distributed garbage. Our algorithm is complete. It detects and reclaims cyclic, acyclic and hybrid distributed garbage through cooperation of the of acyclic collector and the cyclic detector.

In this section, we describe the main idea of the DCDA. We follow an intuitive description that does not consider many subtle aspects; these are addressed in the next sections. However, it provides an easily understandable description of the main idea. For simplicity, we first assume that all mutators are suspended; we call this, the stop-the-world DCDA. Afterwards, we do relax this requirement: concurrent DCDA.

2.1. Stop-the-World DCDA

Consider an object x in process X which is kept alive solely because it is reachable from another process, i.e., it is not locally reachable in process X (where x is allocated). If this object is not invoked for a certain amount of time we can make a guess that this object is, in fact, part of a distributed cycle of garbage. However, we are not sure about that. In order to reach a safe conclusion about x's state (live or dead), we conceived an algorithm that, intuitively, works as follows.

In process X, The DCDA determines which stubs (in process X) are reachable from object x. Those stubs that are locally reachable (directly or indirectly from X's root) are immediately discarded from the point of view of the DCDA; obviously, such stubs do not belong to a distributed cycle of garbage. On the other hand, those stubs that are solely reachable from object x, may be part of such a cycle.

Scions in process X that may also lead (directly or indirectly) to the local graph where x is included, are accounted for as extra dependencies. We define *dependency*, in cycle detection terms, as a scion that leads to the path being traced by the DCDA, i.e., an alternate converging distributed path. Global reachability of the path being traced *depends*, also, on the reachability of such a scion. Therefore, if there is cyclic garbage, such a scion must also belong to it. This dependency is accounted for, and must be eventually resolved by the DCDA. While it is not, no cycle has been safely identified yet. An example is portrayed in Figure 1: the remote reference from w in $P4 \rightarrow x$ in $P1$ is an extra dependency of the cycle, i.e., it is preserving the distributed cycle reachable.

So, the cycles detector sends a probe message along at least one of the above mentioned stubs. These probes (hereafter named as CDM, cycle detection messages) will reach the corresponding scions in remote processes.

In each one of such processes, the DCDA performs as previously described: determines which stubs (inside the process) are reachable from the scion that received the CDM. For clarity, but without loss of generality, assume each process only receives one CDM. Once again, those stubs that are locally reachable are not considered by the cycles detector; thus, the CDM does not follow the corresponding outgoing path. For those stubs that are reachable from the scion that received the CDM, and are locally unreachable, the CDM follows the corresponding outgoing path to remote processes. Once again, all other scions that may lead to any of the afore mentioned stubs, are included as dependencies.

So, the CDM is i) either stopped because, in some process, the DCDA discovers a stub that is locally reachable, or ii) kept on going along the references path so that, eventually it will reach the starting process X. When such event occurs, the CDM carries an algebra that describes the distributed graph that was traversed.

This algebra may indicate that there are dependencies to be resolved, i.e., references pointing to the graph that was traversed; in this case, it is not safe to conclude that we have discovered a distributed cycle; obviously, it is necessary to resolve such dependencies.

However, if the graph is, in fact, a distributed cycle of garbage, then it has no such dependencies yet to be resolved because all those alternate paths were fully and successfully traced. Thus, the DCDA in process X, based on the algebra before mentioned, can safely conclude that it has found a distributed cycle of garbage. So, all it has to do, is to delete the (local) scion from which the CDM has been originated. Then, the distributed acyclic garbage collector will reclaim the remaining objects.

2.2. Concurrent DCDA

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 DCDA may proceed exactly as described previously. However, for obvious reasons, such an assumption is not correct; so, the DCDA 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.

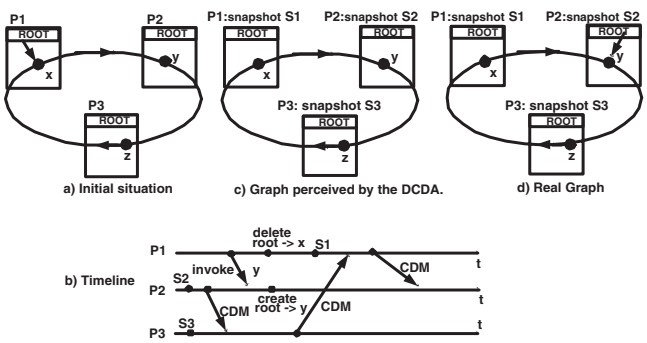

Figure 2. DCDA of independent snapshots.

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) distributed cyclic garbage is always preserved by the acyclic DGC (that is why we need a special detector), i.e., if the DCDA 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 transversed 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.

For DCDA purposes, a CDM-Graph must respect the following invariant:*there can be no invocations along the distributed sub-path to be included in the CDM-Graph*. If we allow this to happen, it means that the mutator is modifying the distributed graph in the back of the DCDA. Consequently, the DCDA may erroneously conclude that it found a garbage cycle. Fig. 2-a illustrates such a case. The initial situation is that of a cycle formed by objects x, y and z in processes $P1$, $P2$ and $P3$, respectively. This cycle is not garbage because x is referenced from the local root in $P1$.

Now consider the sequence of events depicted in the timeline in Fig. 2-b ($S1$, $S2$ and $S3$ are the moments when the corresponding processes make their snapshots). The DCDA starts in $P2$ by sending a CDM to $P3$. Concurrently, the mutator in $P1$, invokes y in $P2$ and deletes the reference from the local root that points to x. As a result of this invocation, a new local reference is created in $P2$'s local root pointing to y. Once this invocation finishes, $P1$ makes a snapshot of its graph (instant $S1$). Given that $S2$ and $S3$ were previously taken, the view of the distributed graph that is perceived by the DCDA instances (i.e., the CDM-Graph) is, in fact, the one represented in Fig. 2-c, instead of the correct one represented in Fig. 2-d. This would lead to the erroneous detection of a distributed cycle of garbage comprising objects x, y and z. This erroneous conclusion would be reached by the DCDA if the invariant mentioned before is not respected. In this case, an invocation took place along the reference path $P1 \rightarrow P2$ 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* without corresponding *Scion* (snapshot of the process holding the scion is not current enough for the CDM-Graph) $\rightarrow$ *Ignore CDM*.
2. *Scion* without corresponding *Stub* (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* with matching *Scion* but there have been remote invocations, 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-DCDA race*.
4. *Stub* with corresponding *Scion* and there were no invocations 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 in, or deny it access to, the path already transversed 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 mutator to run at full-speed at the expense of possibly wasting some detection work (an hypothetical distributed cycle may be partially or completely transversed 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 being performed in the back of the DCDA. In the next sections we describe, in detail, the DCDA using some prototypical examples and showing the algebra carried by CDMs.

3. Algorithm

For clarity, we use simplified language for certain expressions and aspects, when there is no danger of error. In particular, objects are represented by their name (a let-

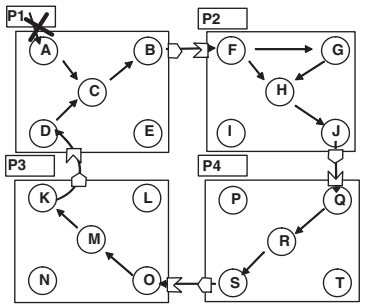

Figure 3. A simple distributed garbage cycle.

ter) and their enclosing process (e.g., A_{P1}, see Fig.3). Subgraphs of connected objects may be represented in abbreviation (e.g., $\{\{A,\ C,\ B\}_{P1}, \{F,\ G,\ H\}_{P2}\}$), aggregated by enclosing process. References may be also explicitly described when relevant (e.g., $B_{P1} \rightarrow F_{P2}$).

For clarity, when we say stubs are reachable from a scion, we actually mean: *stubs accounting for out-going references enclosed in objects that are reachable from a specific object, targeted by an incoming remote reference, this one, represented by a scion.*

Initially, every aspect of the algorithm is explained assuming that all processes are stopped, i.e., without mutator activity. Issues regarding concurrency, safety, and scalability are later addressed.

To describe the cycle detection algorithm, we make use of a simple, yet, general example shown in Fig. 3. There are four processes involved: $P1$ through $P4$. There is a distributed garbage cycle since object A_{P1} has ceased to be reachable from the local root in $P1$. As there are no other reachability roots, the whole cycle is garbage, yet undetectable by acyclic DGC. The cycle can be represented by the following chain of objects (starting and finishing in $P2$):

$\{\{F,H,J\}_{P2}, \{Q,R,S\}_{P4}, \{O,M,K\}_{P3}, \{D,C,B\}_{P1}\}$

In Fig. 3, remote references (e.g., $B_{P1} \rightarrow F_{P2}$) are represented with their associated stubs (e.g., at B_{P1}) and scions (e.g., at F_{P2}).

Data Structures: The structures manipulated by the DCDA are regular acyclic DGC structures, extended with the following information (invocation counters are addressed in Section 3.2):

Scion:

- *invocation counter* (IC): counter for concurrency purposes.
- *StubsFrom*: list of stubs, 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.
- *Local.Reach* : flag-bit accounting for local reachability (i.e., from the local root of the enclosing process) of the stub.

The *StubsFrom* and *ScionsTo* lists, held for each scion and stub, establish reachability associations among scions and stubs in each process. The *StubsFrom* list, for each scion, allows the algorithm to determine, while detecting a cycle, the next set of processes (targeted by out-going references) that should be probed (i.e., sent the CDM) in order to transverse the full cycle.

On the other hand, the *ScionsTo* list, in each stub, allows the algorithm to determine extra dependencies that must be also verified before the cycle is correctly identified.

Finally, the *Local.Reach* flag, in each stub, indicates the local reachability of the stub, i.e., of at least, one of the objects with the corresponding out-going remote reference. This ensures that cycles comprising objects reachable, locally in a process, are never wrongly identified as garbage. When these are found, cycle detection along that path is terminated, with a negative result w.r.t. cycle detection.

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 DCDA, there is no loss of relevant information. This summarization transforms a snapshot of an application graph into a set of scions and stubs, with their corresponding associations. As a matter of fact, references strictly internal to a process are not relevant for the DCDA. In Fig. 3, in process $P2$, references $\{F \rightarrow H,\ F \rightarrow G,\ G \rightarrow H,\ H \rightarrow J\}_{P2}$ fall into this category. This summarization is performed on every snapshot; then it is made available to the DCDA. Thus, while processes can take snapshots by serializing local graphs, the DCDA only uses them in their summarized form, i.e., after graph summarization. In the remainder of the document, *snapshot* and *summarized graph description* are logically equivalent, w.r.t. the DCDA.

In the example shown in Fig. 3, the summarized graph information at process $P2$ would hold the following data (symbol $\Rightarrow$ means *evaluates to or returns*, $\equiv$ relates a field name and its value):

$Scion(F_{P2})_{P2} \Rightarrow \{StubsFrom \equiv \{Q_{P4}\}\}$

$Stub(Q_{P4})_{P2} \Rightarrow$

$\{ScionsTo \equiv \{F_{P2}\}, Local.Reach \equiv false\}$

This means that, in $P2$: i) $Stub(Q_{P4})$ is reachable *from* $Scion(F_{P2})$, ii) $Scion(F_{P2})$ leads *to* $Stub(Q_{P4})$, and iii) $Stub(Q_{P4})$ is not reachable from the local root of $P2$.

Algebra: 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. In the example of Fig. 3, let us assume that a detection is initiated with object F_{P2} as candidate (efficient selection of cycle candidates is an issue

out of the scope of this paper; heuristics found in the literature [10] may be used).

The steps performed and relevant state are the following:

1. $P2 : Alg_0 \Rightarrow \{\{F_{P2}\} \rightarrow \{\}\}$, ($F_{P2}$ chosen as candidate for cycle detection; it is the first dependency)
2. $P2 : StubsFrom(F_{P2}) \Rightarrow \{Q_{P4}\}$, (stubs reachable from F_{P2}).
3. $P2 : Alg_1 \Rightarrow \{\{F_{P2}\} \rightarrow \{Q_{P4}\}\}$, (result algebra after including new found stub; scion already included).
4. $P2 : Send\ Alg_1\ to\ P4$, (send CDM to process $P4$).
5. $P4 : Deliver\ Alg1$
6. $P4 : Algebra\ Matching\ for\ Alg_1$

For each CDM delivered to a process, the DCDA performs an algebraic matching, reducing the message (i.e., finding unresolved dependencies) and determining whether a distributed garbage cycle was detected, as described now.

CDM Matching: In the previous example (at step #6), matching of source ($\{F_{P2}\}$) and target ($\{Q_{P4}\}$) sets in the message would produce, as expected, the following result:

6. $P4 : Matching(Alg_1) \Rightarrow \{\{F_{P2}\} \rightarrow \{Q_{P4}\}\}$
7. $P4 : Cycle\ Found \Rightarrow\ false$

This is due to the fact there are no intersecting elements in the two sets, therefore, no matching could be performed.

Following the flow of CDM, consider now that similar steps (1...4) were performed, this time, at $P4$. They would render the following result:

8. $P4 : Alg_1 \Rightarrow \{\{F_{P2}\} \rightarrow \{Q_{P4}\}\}$
9. $P4 : StubsFrom(Q_{P4}) \Rightarrow \{O_{P3}\}$
10. $P4 : Alg_2 \Rightarrow \{\{F_{P2}, Q_{P4}\} \rightarrow \{Q_{P4}, O_{P3}\}\}$
11. $P4 : Send\ Alg_2\ to\ P3$

And, once the CDM carrying Alg_2 arrives at P3:

12. $P3 : Deliver\ Alg2$
13. $P3 : Matching(Alg_2) \Rightarrow \{\{F_{P2}\} \rightarrow \{O_{P3}\}\}$
14. $P3 : Cycle\ Found \Rightarrow\ false$

This result shows, after matching, the relevant information for cycle detection. It illustrates that, until this point, cycle detection relies on establishing a path that eliminates dependency on F_{P2}, and that the wave front of detection is placed on O_{P3}. This agrees with the intuitive result that, starting in $P2$, no cycle can be safely detected in the path $P2 \rightarrow P4 \rightarrow P3$. Continuing detection at $P3$:

15. $P3 : StubsFrom(O_{P3}) \Rightarrow \{D_{P1}\}$
16. $P3 : Alg_3 \Rightarrow$
    $\{\{F_{P2}, Q_{P4}, O_{P3}\} \rightarrow \{Q_{P4}, O_{P3}, D_{P1}\}\}$
17. $P3 : Send\ Alg_3\ to\ P1$

Upon CDM arrival at $P1$:

18. $P1 : Deliver\ Alg3$
19. $P1 : Matching(Alg_3) \Rightarrow \{\{F_{P2}\} \rightarrow \{D_{P1}\}\}$
20. $P1 : Cycle\ Found \Rightarrow\ false$

And, preparing CDM for forwarding:

21. $P1 : StubsFrom(D_{P1}) \Rightarrow \{F_{P2}\}$
22. $P1 : Alg_4 \Rightarrow$
    $\{\{F_{P2}, Q_{P4}, O_{P3}, D_{P1}\} \rightarrow \{Q_{P4}, O_{P3}, D_{P1}, F_{P2}\}\}$
23. $P1 : Send\ Alg_4\ to\ P2$

When the CDM arrives at process $P2$:

24. $P2 : Deliver\ Alg4$
25. $P2 : Matching(Alg_4) \Rightarrow \{\ \{\} \rightarrow \{\}\ \}$
26. $P2 : Cycle\ Found \Rightarrow\ true$

At this moment, for the DCDA, it is safe to assume that a cycle has been found and that object F_{P2} belongs to it. Therefore, it is safe to instruct the acyclic DGC at $P2$ to **delete the scion** accounting for the remote reference to F_{P2}.

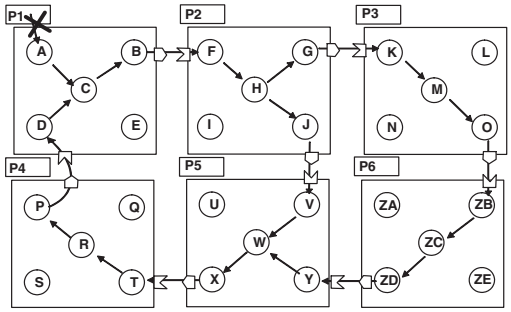

Figure 4. Mutually-linked distributed cycles.

Later, when the LGC runs on process $P2$, the stub accounting for remote reference to Q_{P4} will disappear. This will in sequence, after LGCs in each process, reclaim the distributed cycle. Once again, in the previous steps, were any of the objects reachable locally (in their processes), and that fact would be reflected upon the reachability bit-flag in one stub and terminate the cycle detection.

Multiple Dependencies: With the previous example we have portrayed a situation where the fundamental dependency (the one that holds after algebra matching) lies solely on one object: F_{P2}. This is demonstrated in processes $P4$, $P3$, $P1$, after applying algebra matching, in steps 6, 13, 19. This is not the general case, since there can be several dependencies to account, like in mutually-linked cycles.

In the example in Fig. 3, if there was one other process able to access the distributed cycle (e.g. $P5$), it could only do so, by means of a remote-reference targeting one of the comprised objects. Therefore, this remote reference would be accounted as an extra dependency and would prevent, at this stage, the erroneous detection of the cycle.

3.1. Mutually Referenced Cycles

An important prototypical example of complex cyclic garbage is that of mutually-referenced cycles (see Fig. 4). We describe how the DCDA detects such cycles. Obvious steps, similar to those in the previous example, are omitted for simplicity.

Let us assume that detection starts, once again, at object F_{P2}. Then we have:

1. $P2 : StubsFrom(F_{P2}) \Rightarrow\ \{V_{P5},\ K_{P3}\}$
2. $P2 : Alg_{1a} \Rightarrow\ \{\{F_{P2}\} \rightarrow \{V_{P5}\}\}$ *and send to* $P5$
3. $P2 : Alg_{1b} \Rightarrow\ \{\{F_{P2}\} \rightarrow \{K_{P3}\}\}$ *and send to* $P3$

Since $StubsFrom(F_{P2})$ has more than one element, several different CDM derivations are created with different out-going paths (in this case: one regarding V_{P5} and other to K_{P3}).

Upon arrival of CDM carrying Alg_{1a} at $P5$:

4. $P5 : StubsFrom(V_{P5}) \Rightarrow\ \{T_{P4}\}$
5. $P5 : ScionsTo(\{T_{P4}\}) \Rightarrow\ \{Y_{P5}\}$
6. $P5 : Alg_{2a} \Rightarrow$
   $\{\{F_{P2},\ V_{P5},\ Y_{P5}\} \rightarrow \{V_{P5},\ T_{P4}\}\}$, *send* $P4$

Note that on step #5, an additional pass is performed. For every stub reachable from the given scion, all other scions that may lead to the same stub (in this case $Y_{P5} \rightarrow T_{P4}$), are also accounted for as dependencies to be resolved. This information is readily available in the summarized graph description in $P5$. Similar steps to step #5, in Section 3, were omitted; they were redundant since there was an one-to-one correspondence between stubs and scions (e.g. $ScionsTo(StubsFrom((F_{P4})_{P2}) \Rightarrow \{F_{P4}\})$ and, therefore, no extra dependencies were added.

Upon arrival at $P4$ and applying the algorithm, the outcome includes the following results at each process:

7. $P4: Alg_{3a} \Rightarrow \{\{F_{P2}, V_{P5}, Y_{P5}, T_{P4}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}\}\}$, *send* $P1$
8. $P1: Alg_{4a} \Rightarrow \{\{F_{P2}, V_{P5}, Y_{P5}, T_{P4}, D_{P1}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}, F_{P2}\}\}$, *send* $P2$

When the CDM arrives at $P2$, one of the cycles has been transversed. Detection continuing at $P2$ performs the following steps:

9. $P2: Deliver\ Alg_{4a}$
10. $P2: Matching(Alg_{4a}) \Rightarrow \{\{Y_{P5}\} \rightarrow \{\}\}\}$
11. $P2: Cycle\ Found \Rightarrow false$

Naturally, the algorithm is not able to infer that a cycle has been found. Moreover, matching of Alg_{4a} states that there still is an unresolved dependency on Y_{P5}. This agrees with Fig. 4 where the reference $ZD_{P6} \rightarrow Y_{P5}$ represents a branch of the rightmost cycle connecting with the leftmost cycle. Cycle detection will proceed as presented next:

12. $P2: StubsFrom(F_{P2}) \Rightarrow \{K_{P3}, V_{P5}\}$
13. $P2: Alg_{5a,a} \Rightarrow \{\{F_{P2}, V_{P5}, Y_{P5}, T_{P4}, D_{P1}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}, F_{P2}, K_{P3}\}\}$, *send* $P3$
14. $P2: Alg_{5a,b} \Rightarrow \{\{F_{P2}, V_{P5}, Y_{P5}, T_{P4}, D_{P1}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}, F_{P2}\}\}$, *send* $P5$
15. $P2: (Alg_{4a} = Alg_{5a,b}) \Rightarrow true$, *stop CDM forwarding for* $Alg_{5a,a}$, *terminate branch*

In the previous steps, in process $P2$, two different derivations of Alg_{4a} were created. The first one, $Alg_{5a,a}$, created due to stub $(K_{P3})_{P2}$ should be forwarded to $P3$. Regarding $Alg_{5a,b}$, no forwarding should occur and this branch of detection should be terminated. This stems from the fact this CDM derivation holds information about a cycle, the leftmost, that has already been traced. Thus, no new information was obtained and there is no point in continuing. If not, it would loop forever with the same outcome, i.e., denouncing a dependency of the leftmost cycle on Y_{P5}. This ensures algorithm termination w.r.t cyclic garbage whose reachability is dependent of upstream acyclic garbage not yet reclaimed by the acyclic DGC.

Upon arrival of $Alg_{5a,a}$ at $P3$ we have:

16. $P3: Deliver\ Alg_{5a,a}$
17. $P3: Matching(Alg_{5a,a}) \Rightarrow \{\{Y_{P5}\} \rightarrow \{K_{P3}\}\}$
18. $P3: Cycle\ Found \Rightarrow false$

Preparing the next CDM to forward:

19. $P3: StubsFrom(K_{P3}) \Rightarrow \{ZB_{P6}\}$
20. $P3: Alg_{6a,a} \Rightarrow \{\{F_{P2}, V_{P5}, Y_{P5}, T_{P4}, D_{P1}, K_{P3}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}, F_{P2}, K_{P3}, ZB_{P6}\}\}$, *send* $P6$

Upon arrival of $Alg_{6a,a}$ at $P6$ we have, this time abbreviated:

21. $P6: Matching(Alg_{6a,a}) \Rightarrow \{\{Y_{P5}\} \rightarrow \{ZB_{P6}\}\}$
22. $P6: Cycle\ Found \Rightarrow false$
23. $P6: StubsFrom(ZB_{P6}) \Rightarrow \{Y_{P5}\}$
24. $P6: Alg_{7a,a} \Rightarrow \{\{F_{P2}, V_{P5}, Y_{P5}, T_{P4}, D_{P1}, K_{P3}, ZB_{P6}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}, F_{P2}, K_{P3}, ZB_{P6}, Y_{P5}\}\}$, *send* $P5$

And, upon arrival of $Alg_{7a,a}$ at $P5$ we have, finally:

25. $P5: Matching(Alg_{7a,a}) \Rightarrow \{\{\} \rightarrow \{\}\}$
26. $P5: Cycle\ Found \Rightarrow true$

The distributed mutually referring cycles could have also been detected if derivation Alg_{1b} (see step 3) had been continued.

Several detections can be performed in parallel, at any rate of progress, and comprising any number of processes, without conflict. The previous example is a general one. Other situations mixing acyclic garbage (either upstream or downstream) with cyclic garbage are also solved with the cooperation of the acyclic DGC (previous to cycle detection and after, respectively).

3.2. Dealing With Concurrency

Interaction between the mutator and the DCDA is very limited. As described in Section 2, cycle detection is performed resorting to off-line, summarized descriptions of the memory graph in each process. Thus, there is no contention between the mutator and the DCDA. Mutator actions are not delayed due to synchronization with cycle detection activity.

As it was mentioned in Section 2.2, just combining independently taken graph snapshots, at every process, does not produce a consistent view of the distributed graph. However, these snapshots of different processes are pair-wise combined (with arriving CDMs) just for the purpose of detecting distributed cycles. Therefore, we do not require a global consistent view (e.g., one that is produced by a causal cut) of the distributed graph.

For each detection in course, we need to ensure that every mutator event, performed on the CDM-Graph, is completely represented in the set of snapshots (one for every process comprising the cycle). This stems from the fundamental property that: if there have been mutator events on the CDM-Graph *after* creating snapshots on any of the processes, it means that the cycle is not garbage, at least not yet (as far as we can tell from the information provided in the snapshots). If it is indeed a cycle, to be safe, we may need to update the snapshots of one or more of the processes. In summary, the CDM-Graph cannot be touched while detection is in course.

The word *after*, in the previous sentence, does not imply any notion of global timing; just causality between each pair of processes (determined by mutator events and restricted to the CDM-Graph), solely for the purpose of each cycle detection. A particular case of this situation happens when, for instance, a CDM is delivered to a scion that is not yet inscribed in the summarized graph (it was created after the last graph summarization). In this case, these CDM are simply discarded and those detections terminated. In the following of this section, we present the techniques used to ensure safety, when dealing with the mutator in the case of distributed cycle detection.

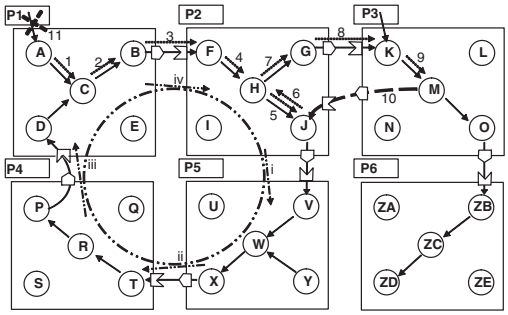

Figure 5. A mutator-cycle detection race.

A cycle detection starts assuming that the objects traced are, indeed, a cycle. If during the flow of a CDM across a series of processes, those objects are invoked/accessed, there should be a way to revise this assumption in the event of this new information. Thus, a distributed race between the mutator and the DCDA may occur with the following sequence of events:

1. There is a local root in one process Px holding the cycle reachable (so no actual garbage cycle exists).
2. A detection is already in course and has not yet reached process Px.
3. The mutator performs a remote invocation (possibly chained through various processes) that switches the root to one of the processes already visited by the detection.
4. Snapshot information becomes available at Px now stating that the object is no longer reachable locally.
5. The detection will be able to trace, lazily, the whole cycle without finding any local root, thus, wrongly detecting a non-existing garbage cycle.

Therefore, algorithm correctness in the presence of mutator activity lies in the ability of determining (until the last moment) if there has been mutator activity in the cycle itself. This is a natural reason to abort cycle detection. However, safety must be preserved without incurring the mutator in significant delays. We present a prototypical example of mutator-detector race and then, the solution to detect those situations and abort detection.

3.2.1. Example of Mutator-DCDA Race In Fig. 5, there are six processes ($P1$...$P6$). There are two independent sequences of events: i) mutator-caused events (numbered 1...11), and ii) cycle detection events (numbered *i...iv*). Algorithm safety consists in showing correct behavior in spite of any interleave of the two sequences. We assume, for simplicity, that there is updated graph summarized information, in every process, and available before event 1 and event *i*.

If no more graph summarizations occur, present information is sufficient in order to handle the situation safely: when CDM arrives at P1 (instant *iii*, regardless of mutator activity), cycle detector will be informed that $Local.Reach(B_{P2}) \Rightarrow \ true$ and will abort detection.

However, if local graph summarized information at $P1$, is updated after event 11 (root erasure) and before event *iii*, with $11 \prec iii$, the combination of graph info at $P1$ and cycle information forwarded from $P2$...$P5$...$P4$ in *iii* will produce an inconsistent view of the distributed graph. Upon arrival of cycle message at $P1$, in this case, we have $Local.Reach(B_{P2}) \Rightarrow \ false$ and, after algebra matching, will forward the information to $P2$ where the cycle will, eventually, be erroneously detected.

In order to prevent this inconsistent behavior, the DCDA must be informed of mutator activity in any part of the path it will trace. We explain, now, how this race condition is avoided.

Recalling structures definition, there is an additional invocation counter associated with every stub and scion. The extra field *IC*, included in every stub and scion, is incremented and piggy-backed, each time a remote invocation (or reply) is performed through the remote reference.

In the previous example, let us assume that we have, before events 1 and *i* as DGC info:

$Stub(F_{P2})_{P1} \Rightarrow \{IC \equiv x\}$

$Scion(F_{P2})_{P2} \Rightarrow \{IC \equiv x\}$

And, off-line, graph summarized information for cycle detection:

$Stub(F_{P2})_{P1} \Rightarrow$
$\{ScionsTo \equiv \{D_{P1}\}, Local.Reach \equiv true, IC \equiv x\}$

$Scion(F_{P2})_{P2} \Rightarrow \{StubsFrom \equiv \{Q_{P4}\}, IC \equiv x\}$

Consider, again, the sequence of events presented where race conditions can occur. CDM holds invocation counters, only when they are relevant to this race condition, others are omitted:

1. $t = i@P2 : Alg_{1a} \Rightarrow$
   $\{\{\{F_{P2}, x\}\} \rightarrow \{V_{P5}\}\}$, *and send to* $P5$
2. $t = ii@P5 : Alg_{2a} \Rightarrow$
   $\{\{\{F_{P2}, x\}, V_{P5}\} \rightarrow \{V_{P5}, T_{P4}\}\}$, *and send to* $P4$
3. $t \in \{1..11\}@\ \{P1..P3\}$:
   (*series of remote invocations initiated in* $P1$ *that result in reference to* J_{P2} *being exported to* $P3$;
   A_{P1} *becomes unreachable locally in* $P1$;
   M_{P3} *now holds the entire cycle globally reachable*).
4. $11 \prec t \prec iii@P1$:
   Take snapshot, update summarized description (now includes $Stub(F_{P2})_{P1} \Rightarrow$
   $\{ScionsTo \equiv \{D_{P1}\}, Local.Reach \equiv \ false,$
   $IC \equiv \ x+1\}$).
5. $t = \ iii@P4 : Alg_{3a} \Rightarrow$
   $\{\{\{F_{P2}, x\}, V_{P5}, T_{P4}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}\}\}$,
   and send to $P1$
6. $t = \ iv@P1 : Alg_{4a} \Rightarrow \ \{\{\{F_{P2}, x\}, V_{P5}, T_{P4}, D_{P1}\}$
   $\rightarrow \{V_{P5}, T_{P4}, D_{P1}, \{F_{P2}, x+1\}\}\}$, *and send to* $P2$
7. $t \succ \ iv@P2 : \ Matching(Alg_{4a}) \Rightarrow$
   $\{\{\{F_{P2}, x\}\} \rightarrow \ \{\{F_{P2}, x+1\}\}\}$
8. *Cycle Found* $\Rightarrow \ false$, *different IC values* $(x$ *and* $x+1)$ *for* F_{P2}, *detection abort*

This use of counters also holds the following advantage: detections already in course for **real** cycles are never aborted due to updates in summarized graph information or, in other words, a detection in course, regardless of when it was initiated can only be aborted if one of its subgraphs was actually touched by the mutator, after it has begun. Thus, there are very loose synchronization requirements for cycle detection and it can be performed lazily without disruption

to applications. Race condition detection in the previous example can be optimized if $P1$ analyzes unmatched counters in the algebra it is about to send to $P2$. However, that is not required to maintain safety.

In summary, the safety rules enforced are: i) CDM sent to non-existent scions are discarded and detection terminated and, ii) different invocation counter values, in source and target sets of CDM, for the same object, cause detection abort.

For lack of space, we address the relevant properties (safety, liveness, completeness, termination, and scalability) of any complete distributed garbage collector, with further detail in [21], discussing them against the algorithm proposed. We also present algorithm pseudo-code.

4. Implementation and Evaluation

The algorithms (reference-listing and cycle detector) were implemented combining C++ and C#. The implementation includes Rotor[20] (a free version of Microsoft .Net[13]) virtual machine modification (for LGC and DGC integration), remoting code instrumentation (to detect export and import of references), and distributed cycle detection.

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

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

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

The approach consists simply on a running thread that monitors existing stubs verifying that they are still valid, i.e., the transparent proxies associated with them still exist. This is achieved using weak-references. 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 so that scions and stubs are created accordingly.

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) 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 DCDA.

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

Table 1. RMI in original Rotor and DGC-extended.

CDM algebra matching is implemented in C# both in Rotor and OBIWAN[3]. The algebra representation, in C#, is optimized to minimize redundancy and ease matching. Thus, each scion/stub representation holds two bits, indicating whether they are present in the CDM source and/or target set.

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, 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 in remote invocations; 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. Table 1 shows results for increasing series of remote invocations of a remote method, with 10 arguments (10 different references being exported/imported), where client and server processes execute in the same machine. This forces the DGC to create 10 scions and stubs each time the remote method is invoked. The overhead associated with the creation of 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 (not a lease-based one) running on Rotor.

The results regarding snapshot serialization (that does not have to be performed frequently) were very bad on Rotor. On average, for graphs with 10000 linked dummy objects (just holding a reference), Rotor serialization takes 26037 ms. To serialize the same graph, with every object containing an additional remote reference (additional 10000 stubs), takes 45125 ms (73% more). Nevertheless, serializing a remote reference is faster than serializing an additional dummy object and, therefore, the impact of serializing stubs 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 (intentionally as Microsoft regards serialization and local GC as product critical code in .Net).

To fully address this issue, we re-implemented the algorithm (the same acyclic DGC, with the same code for DCDA, on OBIWAN[3] 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, roughly, 100 times faster, thus encouraging. They range from 250ms to 350ms which imposes significantly shorter pause times. Furthermore, this needs to be performed only sporadically. This results should also be regarded as upper bound values, since in normal circumstances, application graphs have much higher density of local than of remote references. Stub and scion creation times are comparable with those presented before. CDM matching in cycle detections is inexpensive and performed infrequently and asynchronously.

5. Related Work

Distributed garbage collection has been a mature field of study for many years and there is extensive literature [1, 18] about it. Therefore, we focus this section mainly on other proposals for collecting distributed cycles of garbage, (i.e., algorithms that are complete).

Global propagation of time-stamps until a global minimum can be computed was first proposed in [7] to detect distributed cycles. Distributed garbage collection based in cycles detection within groups of processes was first introduced in [8]. 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 [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, is used in [2, 10]. 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 [11], distributed backtracing starts from suspect 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 backtracking, 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 [16] 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 [15], groups of processes are created to scan 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 [8]. 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 [11], GC structures need to store state about all ongoing detections passing through them.

In [4], 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 subgraph. 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 mech-

anism 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. Moreover, since it consists of a global task being continuously performed, it has a permanent cost. Instead it should be deferred in time and localized just for candidates which are a fraction of the objects.

Our management of unsynchronized summarized graph descriptions can be related to GC-consistent-cuts in databases as proposed in [19]. 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 local GC purposes only.

Another example of usage of snapshots in distributed object stores, while completely unrelated to GC, appears in[12]. It enables efficient system archiving and allows safe computation over earlier system states.

In summary, our approach is more flexible. It has fewer requirements on synchronization, cycle detection state in processes, disruption to mutator and intrusion to LGC. Furthermore, it has been implemented on realistic off-the-shelf systems.

6. Conclusions

We presented a comprehensive solution to the problem of distributed garbage collection. The main contributions of our work are: i) a novel distributed cycles detector algorithm that requires no global synchronization, is scalable, not intrusive w.r.t mutator and LGC, and makes progress without requiring all processes to participate, ii) an implementation on Rotor and on .Net with minimum impact on the source code of Rotor runtime, iii) the notion of an algebraic matching process for distributed cycle detection.

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 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.

Finally, although we have implemented the DCDA in Rotor and in OBIWAN, our solutions are rather general. It is possible to apply the same ideas and, in particular the notion of the CDM algebra and the DCDA, to other platforms. In the future, we plan to address the formal correctness proof of the DCDA.

References

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

[2] P. 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.

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

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

[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] M. Fuchs. Garbage collection on an open network. In H. Baker, editor, *Proc. of Int'l W'shop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Concurrent Engineering Research Center, West Virginia University, Morgantown, WV, Sept. 1995. Springer-Verlag.

[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. Quenniac, and J. 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, Jan. 1992.

[9] B. Liskov and R. 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), Aug. 1986. ACM.

[10] 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.

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

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

[13] D. S. Platt. *Introducing the Microsoft.NET Platform*. Microsoft Press, 2001.

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

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

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

[17] M. Shapiro, P. Dickman, and D. Plainfoss. Robust, dist. references and acyclic garbage collection. In *Symp. on Principles of Dist. Computing*, pages 135–146, Vancouver (Canada), Aug. 1992. ACM.

[18] M. Shapiro, F. L. Fessant, and P. Ferreira. Recent advances in distributed garbage collection. *Lecture Notes in Computer Science*, 1752:104, 2000.

[19] 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.

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

[21] L. Veiga and P. Ferreira. Asyncronous, complete distributed garbage collection. Technical report rt/11/2004, INESC-ID Lisboa, june 2004.