Asynchronous, complete and distributed garbage collection in the NGrid project

Joannès Vermorel, Warith Harchaoui

▶ **To cite this version:**

HAL Id: hal-00009571

https://hal.archives-ouvertes.fr/hal-00009571

Submitted on 6 Oct 2005

Asynchronous, complete and distributed garbage collection in the NGrid project

Summer 2005

Computational Biology group
at the Geostatistics Center of the École des Mines de Paris
France

Joannès Vermorel
École normale supérieure de Paris
Computational Biology group at the Geostatistics Center
of the École des Mines de Paris

Warith Harchaoui
École Supérieure d'Électricité
Supélec

Abstract

Luis VEIGA and Paulo FEIRRERA explain that the distributed garbage collection problem can be divided in two :

- acyclic garbage collection solved by the *Reference Listing* method [1] in 1993.

- cyclic garbage collection solved by the *Graph Summarizer* method [2] in 2004 which relies on the *Reference Listing* method.

The aim of this paper is to implement the *Graph Summarizer* algorithm and a few improvements in the NGrid distributed environment.

Introduction

Recent advances in computational biology field need important computational resources. In order to make intensive computations, one can not always afford uni-processor machines such as supercomputers. For example, if one is interested in the determination of the precise role of genes in chemical processes, then one deals with 3.10^9 nucleotides.

So, the problem is twofold because we have to manage:

- the memory space

- the computation

The aim of this paper is the construction of an automatic memory manager called garbage collector in a distributed environment. It first dealt with the deep understanding of two publications [1] [2] about the distributed garbage collector subject and the integration of the algorithms into the NGrid project[1] which is an open source grid computing framework.

1 Presentation of the Veiga & Ferreira's article [2]

1.1 The Garbage Collector

In traditional non-garbage collected programming language such as C or C++, memory is managed by the user with the well-known *malloc* and *free* keywords. Since the 90s, garbage collected languages such as Java, Python and C# have a great impact on the way people are programming because users do not need to think about releasing memory anymore, thanks to the garbage collector.

In local environments, the garbage collector problem was solved almost 20 years ago [3]. But distributed environment is not an easy case because of conflicts between machines while accessing to shared data. This phenomenon is called *Race conditions*. In fact, the distributed problem was partially solved in 1993 with a method called *Reference Listing* [1]. We think that the problem is now completely solved since 2004 thanks to Veiga & Feirrera [2].

1.2 The different kinds of garbage

A simple *singleton* garbage is created when a single temporary object is needed during the call of a method. The object can not be reached by any reference after the method's execution but it still blocks memory space. One says that the object is a *singleton* garbage because it lost its root. In fact there are two types of garbage:

acyclic garbage Now, lets consider that the temporary object is the first element of a temporary linked list. At the end of a method, the whole list becomes garbage because the only root was the first element. The list is an *acyclic* garbage. This kind of garbage is retroactively destroyed by the *Reference Listing* [1] method. One can easily understand that a *singleton* garbage is in fact an *acyclic* garbage.

---

[1]NGrid http://ngrid.sourceforge.net/
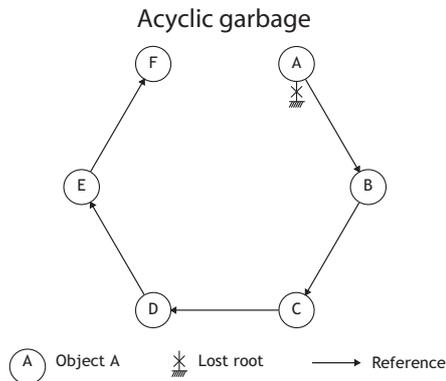
Acyclic garbage

cyclic garbage Finally, the initial object could have been the root of a cyclic graph which can be handled by the Veiga & Feirrera's *Graph Summarizer* algorithm [2].

Cyclic garbage
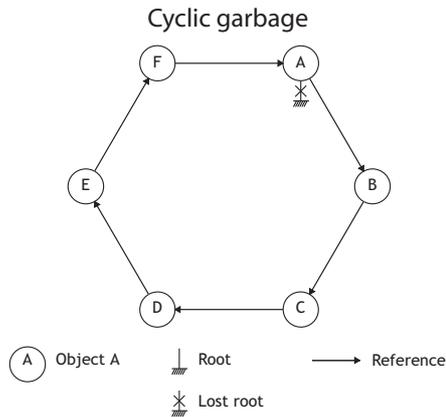
As we wrote before, *Race conditions* make the distributed garbage collection more difficult than the local one.

1.3 A distributed context

An efficient garbage collector must work under the scene of the environment. This means that the user should not have even to think about memory management. It has to be completely transparent *a fortiori* in a distributed environment where heavy computations are made. The NGrid project is a .Net based distributed environment operating system independent where each machine has a local garbage collector inherited from the .Net framework for simple objects which are bounded in the current machine. Moreover in NGrid, distributed objects called GObjects (*G* for Grid) are handled by the distributed garbage collector. For the sake of simplicity, the two kinds of objects shall not be distinguished as this paper only deals with GObjects. The very goal of this paper is to understand the *Graph Summarizer* algorithm and to implement it in the NGrid project.

Up to our knowledge this problem has never been tackled so far in the literature we read without global synchronization or global consensus. Thus we are sorry to be unable to provide any comparison with other algorithms.

1.4 The algorithm

The main strengths of the *Graph Summarizer* are:

the Completeness All kinds of garbage are eventually collected (cyclic and acyclic garbage).

the Fault-tolerance One machine can crash without disturbing the computation of the garbage collection.

the Absence of any global synchronization or consensus which makes the crucial advantage of the algorithm.

The main idea of the algorithm is simple:

Transform cyclic garbage into acyclic garbage.

Indeed, the old *Reference Listing* [1] strategy retroactively handles the acyclic garbage. So, we should only remove few elements of a cycle garbage to transform it into an acyclic one. But still remains the problem of garbage cycle detection. It should be done without errors because deleting non garbage objects is unacceptable.

The first idea one can have about the problem may be simple but wrong. It would be like:

Data: A candidate suspected to be garbage
foreach *objects o linked to the candidate* **do**
  Read o's local references graph.
  if *o is rooted* **then**
  | The candidate is not garbage.
  else
    if *o is met twice* **then**
    | The candidate is garbage
    else
    | Launch the algorithm with o as the new candidate.
    end
  end
end
Result: Is the candidate garbage

Algorithm 1: Naive algorithm

This example shows why this naive algorithm is not correct.

Eight shaped cyclic garbage
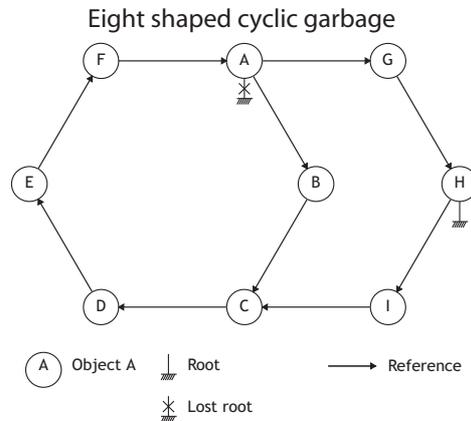
In the alphabetical order starting from A, when leaving F, A is met twice without meeting any root. In fact the H root makes the whole graph reachable and thus the graph is not garbage at all !

We can now easily understand that a more sophisticated algorithm is needed to detect cyclic garbage. The authors dramatically simplify the problem by adding a new hypothesis. The *Reference Listing* [1] strategy destroys all acyclic garbage. Thus:

> The remaining garbage is necessarily cyclic.

This implies that an object with no direct root and no reference towards itself (like an extremity of an acyclic graph) can not exist.

The main notion of the article is the *Graph Summary*. It consists of two sets:

**entering set** made of identifiers of objects entering in a domain with a reference toward itself from another domain.

leaving set made of identifiers of objects leaving a domain with a reference toward itself from the current domain.

It sums up a branch of a locally connex reference graph in a given domain. Here is an example[2]:

---
[2] *Application Domains* and machines are not distinguished for the sake of clarity.

Graph summary
on domain 2 with candidate B (or A, C, I and N)
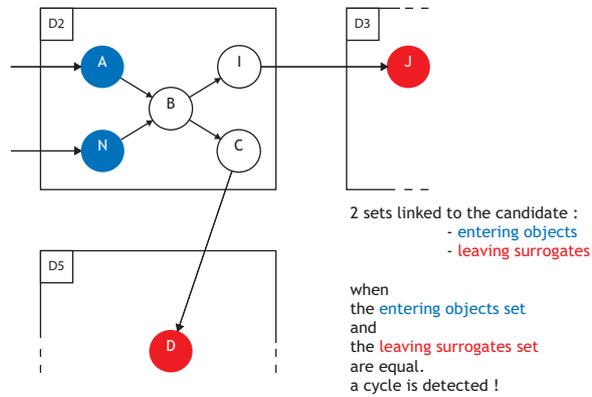
As a matter of fact, from a starting point candidate suspected to be in a garbage cycle, the algorithm gets the summaries of the same connex graph in all domains. There are only two ways to stop the algorithm:

The graph has a root the suspicion was wrong.

The leaving and entering sets are equal the suspicion was right !

Here is the pseudo-code of the *Graph Summarizer* algorithm.

```
Data: candidate c suspected to be garbage on the current domain and
      the global Graph Summary which is initially empty
Compute the local Graph Summary of the current domain with c as
candidate.
if the local graph is rooted then
 │ The graph is not garbage, so give up the detection
else
 │ if the local leaving set is empty then
 │  │ Merge the local and the global Graph Summaries
 │  │ If the global entering and leaving set are empty then the
 │  │ candidate is garbage.
 │ else
 │  │ foreach target t in the leaving set do
 │  │  │ Create a new Graph Summary g that is the union of the local
 │  │  │ (with t as the only element of the leaving set) and the global
 │  │  │ ones.
 │  │  │ if the entering and leaving set of g are equal then
 │  │  │  │ The candidate is garbage.
 │  │  │ else
 │  │  │  │ Continue the algorithm with t as candidate in the domain
 │  │  │  │ referenced by t.
 │  │  │ end
 │  │ end
 │ end
end
Result: Is the candidate garbage
```

Algorithm 2: *Graph Summarizer* algorithm

The article has a proof of the algorithm. But we tested it on many configurations : first with the old pen and paper and then with a computer. As the results were correct in many different cases, we assume that the algorithm is correct.

Of course, the tests were not only on the examples presented before but also on many others such as spiral shaped graphs, distributed and local graphs, graphs that have a reference to a root with garbage elements ... The tests were purely algorithmic. No domains could have been implemented at this stage, we needed to know if the *Graph Summary* algebra actually works or not. We thought it was the best way to deeply understand the algorithm and why it does work. In fact, these tests also give us hints about the figures involved.

The name *Graph Summary* is very well chosen. For example, a graph can have thousands of objects in 10 computers but only 20 leaving and entering objects. So, the compression factor can be very high which saves remote communication between domains.

Unfortunately, in a one-thread and non-distributed environment, the λ-calculus can help to demonstrate algorithms but the concurrent environment does not have such a demonstrating tool yet. Of course, the λ-calculus does not fit the distributed environment because of *Race conditions*. In spite of that, we decided to trust the tests and use the algorithm.

2 Integration of the algorithm into the NGrid project.

The main difficulty was to consider pure algorithmic problems and pure technical problems too. The *leitmotiv* was: How to implement a mathematical algorithm in a real computational environment ?

2.1 Adaptation of the algorithm for NGrid

In order to use the *Graph Summarizer*, we had to do two modifications :

- To translate low-level notions to high-level notions.

- To build the *Inhumation* technique to actually read a references graph.

The algorithm does not work with objects but only with their identifiers because we can not destroy objects that are held by a reference...

Translation of low-level notions to high-level notions

The article [2] has a low-level *clr* approach to present the algorithm. But in fact, NGrid is a high-level environment. That is why low-level notions such as *scions* and *stubs* are irrelevant in a high-level environment. We decided to replace them by the notions of entering and leaving objects (from a current domain's point of view).

A leaving object is not hosted in the current domain. It simply tells where the real object is that is why leaving object are called surrogate in the *Reference Listing* literature [2]

An entering object is a hosted object that is referenced by an other object in an other domain. In the *Reference Lisiting* literature, it is a hosted object that has *dirty set* with at least two domains.

The *Inhumation* technique

Veiga and Feirrera do not explain how to get the references graph of a domain. That is why we implemented the *Inhumation* method. It is based on serialization basically meant to be used by the .Net framework for remote communication. The idea consists in reading the serialization result because it necessarily contains all the references data we need.

But the serialization process is heavy and moreover it can not be done on active objects. The solution is to ignore the references data of active objects because they *a fortiori* do not concern garbage. Furthermore, we have to avoid disturbing domain's processes by freezing everything to serialize it. We suggest to serialize objects by freezing them one by one while delaying the calls on the object being serialized until the serialization process ends.

The definition of rooted object is a little bit blurred. In a domain, an object is rooted if it is itself a root or if it is referenced by a root or a rooted element. Our idea is to simplify the notion of rooted object by considering all the objects referenced directly or not by roots as real roots.
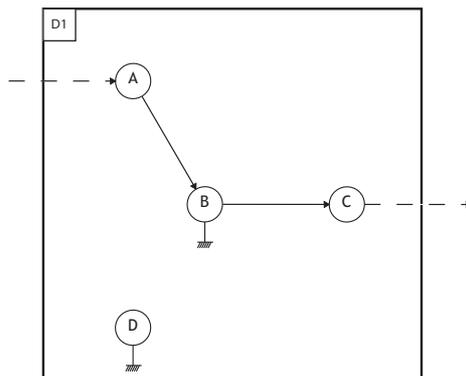
The main idea is to transform and simplify the references graph by keeping its meaning in a garbage detection point of view. Here is an example that explains the *Inhumation* technique:
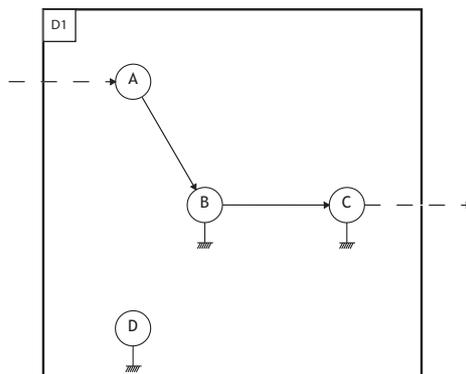
The *Inhumation* technique

Real Domain references graph
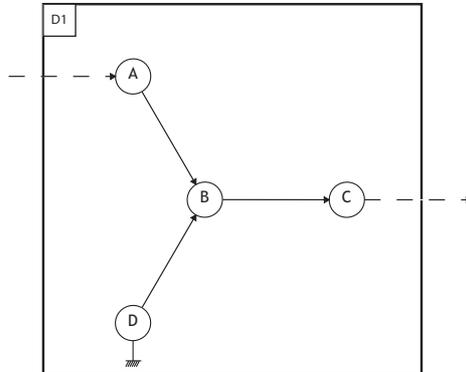
What is seen through the *Inhumation* glasses

What is finally seen to simplify the root notion

2.2 Optimizations and improvements of the Veiga & Feirrera's algorithm due to the NGrid environment

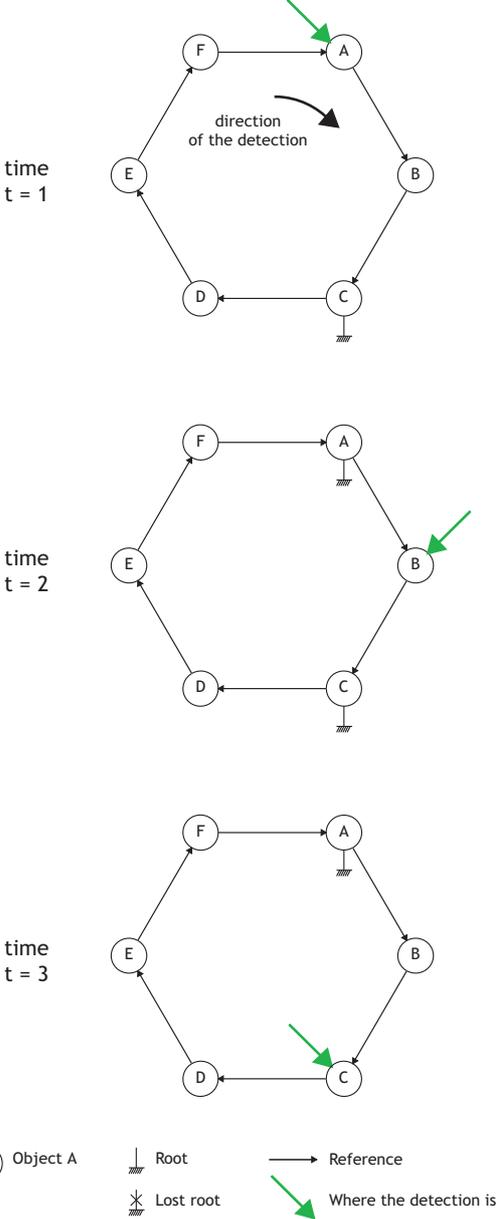These are few optimizations we managed to implement :

- the time stamps strategy to prevent from objects displacements and modifications.

- the *Initiative* grouping strategy to save computations.

- the storage of *Initiative*'s data in each domain to save remote communications.

- the *OneWay* strategy to reduce the number of domains involved at the same time in the cyclic detection process.

The time stamps strategy

We pointed out how difficult it is to get the current domain references graph. Moreover, objects can be changed after the serialization process and before the end of the detection. If it is the case, these objects has been changed through references thus they are directly or indirectly rooted.

To get this important information, we use a time stamp strategy: every access on an object increases a counter of it. During the domain references graph's serialization, a snapshot of time stamps is made. At the end of the detection, if the detector claims that the graph is garbage then we have to check if the time stamps changed. If they did, then the graph is not garbage ! If they did not, then the detector is right. Here is an example that shows how crucial the time stamps strategy is:

Example that shows
how the timestamp strategy is crucial

time t = 1

direction
of the detection

time t = 2

time t = 3

(A) Object A	⊥ Root	→ Reference
	⚹ Lost root	⬎ Where the detection is

In this example, without any time stamps information, the algorithm would destroy a non-garbage graph because the detection did not parse any root. The time stamps strategy relies on a simple fact:

> If an object has been accessed, then it is not garbage.

The *Initiative* grouping strategy

The article [2] explains the *Graph Summarizer* algorithm for one candidate but one can shows a better profit for the expensive serialization process. Our idea is to launch a garbage detection with many candidates of the same domain.

The different detection states are boxed in a bigger process called *Initiative*. If a detection process has ended without being able to conclude, it has to be merged with the other detection process of the same *Initiative* with one common parsed element.

In fact the set parsed by a detection can be huge but the union of the entering an leaving sets is enough ! As a matter of fact, to check if two *Graph Summaries* can be merged does not cost much computation since the sets are small.

To deeply understand what we are talking about, lets consider the eight shaped graph and the crossroad due to the A object.

The storage of heavy *Initiative* data

Unfortunately, to save remote communications between domains, we must store all the heavy data on each visited domain such as the time stamps.

Furthermore, as *Graph Summaries* do not take much time to be calculated nor much memory space compared to the serialization process, we calculate all existing *Graph Summaries* even if they are not needed. In fact, as an *Initiative* can meet a domain twice, we do not know if one *Graph Summary* will be useful or not. But to avoid incoherent graphs due to domain's snapshots at different times, we just pick up one of the old, required and already calculated local *Graph Summary*.

But all these data are linked to an *Initiative*, and when it is over a last remote method can destroy all *Initiative* data in each visited domain.

Finally, we have to prevent from infinite loops due to incoherent snapshots because of *Race conditions* where a leaving surrogate points to a leaving surrogate of the same object in an other domain. The solution is stop a detection process if it asks twice the same local *Graph summary* which is easy when local *Graph summaries* are stored in domains.

the *OneWay* strategy

Finally, here is the last improvement of our work based on the *Graph Summarizer*: to transform the algorithm into a non-recursive one. Along the references graph of a candidate object, the algorithm can meet crossroads between domains. The first natural idea is to parse domains in a *for cycle*. But it would cost too much remote communications while a domain would wait each other until the next visited domain returns an answer which creates eventually a tree of waiting domains. Waiting for every method to complete incurs a performance penalty if the calls themselves are independent. Our idea is to do the *for cycle* with the *OneWay* .Net feature. This feature allows asynchronous calls without waiting for the result. The only problem is to make these calls independent.

In fact, we can show that the branches of a connex graph can be parsed in every orders. In a crossroad, two detection processes might *want* to parse different domains especially when the graphs are not easily connex. In this case a vote is made between the *Graph Summaries* of the same *Initiative* to

determine what is the next domain to be visited. An *Initiative* ends when there is no more domain to be visited.

When the vote does not agree with one detection process, it just travels without doing anything but *Graph Summaries* are small which saves remote communications.

Using this powerful *OneWay* feature means that there is a maximum of two domains working on the same time in a *Initiative* detection process. To sum up all these features, here is the algorithm :

```
Data: Set of (candidate, Graph Summary) pairs which is a Cycle
      Detection Message of an Initiative
foreach (c, g) pair of the set do
    if the pair needs to parse the current domain then
        Compute the local Graph Summary of the current domain with c
        as candidate. if the local Graph Summary has already been asked
        then
        |   Create the (null, g) pair.
        else
            if the local graph is rooted then
            |   The graph is not garbage, so forget the pair.
            else
                if the local leaving set is empty then
                |   Create a new global Graph Summary that is the union
                |   of g and the local Graph Summary.
                else
                    foreach target t in the leaving set do
                    |   Create a new global Graph Summary that is the
                    |   union of g and the local Graph Summary (with t as
                    |   the only element of the leaving set) and the global
                    |   ones.
                    end
                end
            end
        end
    else
    |   Keep the pair.
    end
end
Merge each (null, g) pair among the kept and created pairs with all pairs
that have a common local Graph Summary
foreach (null, g) pair among the kept and created pairs do
    if the entering and leaving set are equal then
        The candidate is garbage but check it with the time stamps
        strategy.
        Forget the pair.
    else
    |   Keep the pair.
    end
end
if there are only (null, g) pairs then
|   The Initiative is over.
else
    Launch a vote between (not null, g) pairs to determine the next
    domain to visit.
end
```

Algorithm 3: Improved *Graph Summarizer* algorithm in the NGrid context

The integration of these improvements have been successfully implemented in the NGrid project. The codes are available at : http://sourceforge.net/projects/ngrid

Conclusion

The *Graph Summarizer* algorithm is a solution that follows a hybrid approach: an acyclic distributed collector based on the *Reference Listing* [1] algorithm and a cycle garbage detector that complements the first thus providing a complete solution for the problem of distributed garbage collection.

We managed to implement the algorithms with few improvements due to the high-level environment : NGrid. Now published heuristics in objects' lifetime prediction should solve the problem of picking the right candidate before launching the big algorithm presented or maybe the right domain...

The main contributions of our work to the garbage collection problem are:

- an implementation of the *Graph Summarizer* algorithm to detect cyclic garbage.

- the *Inhumation* technique to get the references graph of a domain.

- a *OneWay* version of the *Graph Summarizer* algorithm.

Today, our implementation of a cyclic distributed garbage collection is meant to be launched by the user with all the current domain's objects as candidates. A lifetime object predictor could be integrated to launch automatically our algorithm.

References

[1] Greg NELSON Susan OWICKI Andrew BIRELL, David EVERS and Edward WOBBER. Distributed garbage collection for network objects. Technical Report 116, digital - Systems Research Center, Palo Alto, California, United States of America, December 1993.

[2] Paulo FEIRRERA Luis VEIGA. Asynchronous, complete distributed garbage collection. Technical Report RT/11/2004, INESC-ID/IST, Lisboa, Portugal, June 2004.

[3] Paul R. WILSON. *Uniprocessor garbage collection techniques.* Springer-Verlag.s, Saint-Malo, France, 1992.