Garbage Collecting the Grid: A Complete DGC for Activities

Denis Caromel, Guillaume Chazarain, Ludovic Henrio

▶ **To cite this version:**

Denis Caromel, Guillaume Chazarain, Ludovic Henrio. Garbage Collecting the Grid: A Complete DGC for Activities. Middleware 2007- ACM/IFIP/USENIX 8th International Middleware Conference,, Nov 2007, Newport Beach, United States. inria-00180150

HAL Id: inria-00180150

https://inria.hal.science/inria-00180150v1

Submitted on 17 Oct 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Garbage Collecting the Grid: A Complete DGC for Activities

Denis Caromel, Guillaume Chazarain, and Ludovic Henrio

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr

Abstract. Grids are becoming more and more dynamic, running parallel applications on large scale and heterogeneous resources. Explicitly stopping a whole distributed application is becoming increasingly difficult. In that context, there is a strong need to free resources a soon as they become useless, leading to automatic termination, using distributed garbage collecting techniques. We propose in this paper a new distributed garbage collector for active objects taking into account cycles but with a complexity similar to the distributed garbage collector of Java/RMI. The algorithm is based on a different approach to collect acyclic and cyclic garbage. On one hand, acyclic garbage is collected by knowing the immediate referencers of an active object and detecting the lack of these referencers. This behavior with respect to acyclic garbage is common to the distributed garbage collector of RMI. On the other hand, cyclic garbage is detected by considering the recursive closure of all the referencers of an active object and finding cycles of active objects waiting for requests. These cycles are found by letting idle active objects make a consensus on a common final activity. The algorithm is fully distributed and has been implemented with no modifications to the local garbage collector. Benchmarks have shown the scalability of the algorithm in a grid context.

Keywords: distributed garbage collection, cycle detection, grid computing.

1 Introduction

Computer grids can be used to deploy complex and long running applications made of distributed activities. To optimize the resource usage of the global application, it is important to free resources held by idle activities as soon as possible.

When these grid applications are written in a high level language such as Java, the underlying platform exposes a local garbage collector to simplify the memory management. The automatic garbage collection mechanism has been adapted to objects accessible over a network in the form of distributed garbage collectors, abbreviated DGC. Currently, the most used DGC implementation seems to be the one of RMI [1,2]. However, this DGC is unable to collect distributed cycles of garbage since it is based on a reference listing [3] approach. Recent cyclic

R. Cerqueira and R.H. Campbell (Eds.): Middleware 2007, LNCS 4834, pp. 164–183, 2007.
© IFIP International Federation for Information Processing 2007

distributed garbage collector [4] typically suffers from a large space complexity, because each process has a view of the whole distributed system.

This paper presents a DGC geared towards the collection of distributed activities which is able to collect both acyclic and cyclic garbage, typically referred to as a complete DGC. Activities are represented by the active object [5] model which provides asynchronous method calls to the object model, thus is well suited to grid applications. More precisely, active objects are remotely accessible objects with their own thread of activity and requests queue.

As they attempt to find cycles of active objects, complete DGC algorithms need a view of the reference graph in order to find cycles in it.

The contributions of this paper consist of:

– a method to build the reference graph between activities without modifying the local garbage collector,
– the identification of cycles of idle activities as cyclic garbage instead of the more common unreachable strongly connected component,
– a DGC algorithm using this characterization of cyclic garbage and
– experimental results of an implementation of this DGC algorithm.

The rest of the paper is organized as follows: section 2 describes the construction of the reference graph which provides the necessary input for the DGC algorithm. How the algorithm identifies garbage (both acyclic and cyclic) based on this reference graph is described in section 3 while section 4 discusses some implementation issues as well as the complexity of the algorithm. Experiments are described in section 5. Finally, section 6 discusses the related work, and some future work ends the paper.

2 The Reference Graph

A crucial aspect of the garbage collection of activities is determining which other activities still hold references to the activity currently being examined and may activate it later on. The references between different activities are in fact transitive references, since there can be a chain of local pointers between the active object and the remote reference. This is identical to the graph summarization technique in [4].

This work is built upon the active object model which permits some assumptions, but these assumptions are not fundamental to the DGC algorithm. One of these assumptions is the *no-sharing property* described below, and we will describe in the discussion in Section 4.1 an approach to relax this requirement.

2.1 The No-Sharing Property

As the distribution of activities on the grid is often unknown beforehand, activities should typically not share (by aliasing) references to the same passive object (standard object). Since references to other activities are represented by passive objects called stubs, such stubs of remote objects are not shared either.

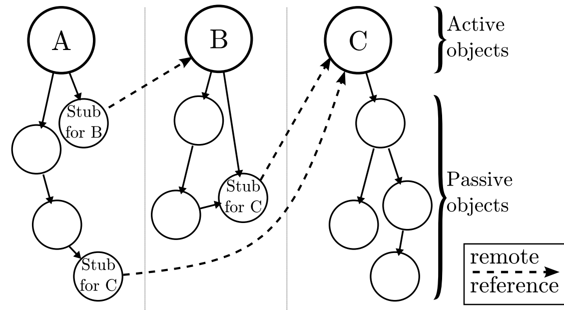


Fig. 1. The no-sharing property: no passive objects are referenced by more than one active object

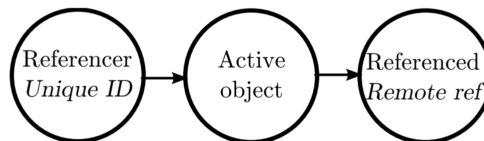


Fig. 2. Remote references are needed for referenced active objects, but only a unique ID is needed to track referencers

This rule[1], illustrated in Figure 1, is called the *no-sharing property*. With this property, once we know that a given active object references a given stub, we can assume that this stub will always be exclusively referenced by this same active object. This property allows, as we will see in the next section, to build the reference graph without modifying the local garbage collector.

2.2 Building the Reference Graph

An active object maintains two kinds of relationship with other active objects. There are referencers and referenced active objects. The connectivity requirements for these two kinds of active objects are different. On one hand, the DGC algorithm will try to contact referenced active objects, so an access to the remote references used by the application is needed. On the other hand, referencers only need to be identified by a unique ID, as the DGC algorithm will never try to directly contact them. Concretely, as seen in Figure 2, the DGC algorithm will only try to contact referenced active objects and will just store the ID of the active objects contacting it. This is an important property of the algorithm in that it does not require more connectivity than the deployed application. This

[1] Practically speaking, the only way to defeat this rule is to put a remote reference in some static variable since communications between local or remote active objects always go through a serialization and deserialization step, so no sharing is possible with only these communications.

aspect is particularly useful in grid contexts where the deployment has to deal with connectivity limitations like firewalls and NATs.

The proposed reference graph can be built transparently on top of a local garbage collector. The graph is constructed by hooking into the deserialization of stubs, and by remembering which local active object A (i.e. the recipient of the message) triggered the deserialization, then A can add the stub target B to its list of referenced active objects. When A subsequently sends a DGC message to B, B will add A to the list of its referencers.

Iterating this process of adding edges between active objects builds the reference graph. The reference graph is represented for each active object instance by the list of its remote references. However, a local active object instance may reference several stubs representing the same remote active object. As a consequence, it is crucial to keep track of all of these stubs, as only the disappearing of all of them indicates that an edge should be removed in the reference graph. Instead of independently tracking these stubs, a more efficient way is to add a common tag (a reference to a dummy object) in every stub instance for the same remote object owned by the same local active object, and then the DGC just has to keep a weak reference to this tag in order to detect the local garbage collection of all of these stubs.

3 The Distributed Garbage Collector Algorithm

The distributed garbage collector algorithm relies on the following *Garbage* property to discriminate garbage activities:

$$\forall x, Garbage(x) \Leftrightarrow (\forall y, y \rightarrow^* x \Rightarrow Idle(y)) \quad (1)$$

An active object (x) is said to be garbage if and only if the reflexive transitive closure of its referencers (y) is idle, the active object x is included in the reflexive transitive closure of its referencers. The concept of local idleness ($Idle(x)$) for an active object must be provided by the middleware.

The *Garbage* property is verified using two different approaches for acyclic and cyclic garbage. On one hand, acyclic garbage is found by ensuring that the set of the direct referencers of an active object is empty, this satisfies the *Garbage* property. On the other hand, cyclic garbage is found by letting an active object make a consensus on a "final activity clock" by the reflexive transitive closure of its referencers.

3.1 Detecting Acyclic Garbage Using a Heartbeat

To detect acyclic garbage we request that referencers of an activity periodically send a *DGC message* to the referenced activity in a heartbeat fashion. Its frequency, thereafter referred to as *TTB* for TimeToBeat, is a constant known by all participating active objects in the distributed system. Therefore, increasing TTB lowers the overhead of the DGC but makes it slower to reclaim garbage.

If an active object receives no DGC messages for a certain amount of time, it considers itself as garbage, and is thus destroyed. This amount of time is called TTA for TimeToAlone. This parameter designates the delay after which an active object considers it must have received a DGC message from all of its referencers.

When a local active object deserializes a reference to a remote active object, it makes sure that at least one DGC message is sent to this remote active object at the next broadcast. In other words, even if the reference is quickly garbage collected, the algorithm remembers that one DGC message must be sent anyway. This ensures that a reference to a remote active object that would be quickly exchanged between two other active objects receives DGC messages to keep it alive.

The TTA value should satisfy the formula $TTA > 2 * TTB + MaxComm$ with $MaxComm$ being an upper bound on the communication time between active objects. This formula ensures that referencers always get a chance to send a DGC message before declaring that no messages were received. The worst case is an active object A giving just before its broadcast a reference B to another active object C that has just broadcasted. If the B stub on A is collected just after giving it to C, then C will have to wait $2*TTB+Comm$ without receiving DGC messages from A or C, $Comm$ being the time to send the reference.

3.2 Detecting Cyclic Garbage by Making a Consensus

The very high level view of our algorithm to find distributed cyclic garbage is to traverse the recursive closure of an active object's referencers in order to find cycles of idle activities. This traversal checks at each step that the currently visited active object is idle. An active object may not be able to directly contact its referencers (firewall, NAT), hence the traversal is done in the opposite direction, using the periodic DGC messages described in the previous section. The outcome of the traversal is not affected by this restriction on the direction as a traversal in the correct direction is simulated over the traversal in the opposite direction.

This traversal builds a reverse spanning tree over the reference graph. This means that every active object except the originator (maker of the consensus)

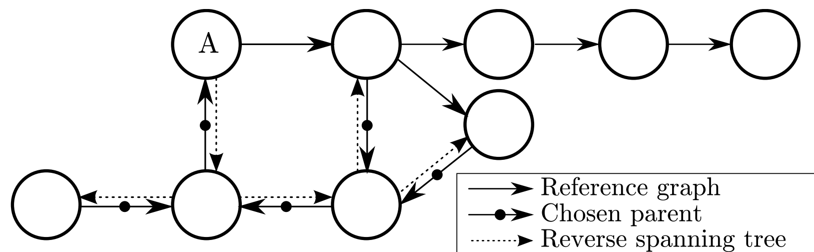


Fig. 3. A reference graph and an associated reverse spanning tree: the reverse spanning tree is used by the originator to make a consensus on its "final activity clock"

promotes a single of its referenced active objects as its parent. The reverse spanning tree is represented by active objects knowing their parent instead of their children because of the connectivity restrictions. Figure 3 shows a reference graph and a possible reverse spanning tree rooted at A over this graph. This tree permits to explore the recursive closure of A's referencers even if it contains cycles.

Activity Clock. The traversal ensures that all visited active objects are idle. As this is obviously a source of races, for each active object the local check is more thorough.

The cyclic garbage collector algorithm requires every active object to maintain a *named* Lamport logical clock [6], which is used to determine which activity was the last active. The clock is *named* in the sense that the ID of the active object incrementing the clock is embedded in the clock. This active object is called the owner of the activity clock. This additional information provides a total ordering of the named clocks by letting the comparison function of two activity clocks first compare the clock values and then the active object IDs if the clock values are identical.

The essence of using a Lamport logical clock is that if an active object receives a DGC message with a clock which is more recent than its own view of the clock, it updates its clock accordingly. When an active object A increments the activity clock $ID : Value$, it turns it into $A : Value + 1$. A garbage cycle is detected by an active object A when the following conditions are met:

- the recursive closure of A's referencers have a common activity clock called *final activity clock* and
- A is the owner of this final activity clock and is idle.

In order to check that every recursive referencer of an active object has the same activity clock, we construct a reverse spanning tree among the active objects having the same activity clock. If all the recursive referencers agree on the activity clock, then the spanning tree will span all the referencers; which will finally agree on the garbage collection: the consensus will be transmitted over a tree. If some referencers have a different activity clock, then at least one of these referencers references an active object of the spanning tree, and this referenced active object will not agree on the consensus for the final activity clock: no active object is garbage collected.

The activity clock is used to regulate the concurrent execution of the distributed garbage collection and the application which may modify the reference graph through the passing of objects as parameters or return values. The relevant occasions when the activity clock is incremented will be detailed after.

DGC Messages and Responses. For active objects to agree on a final activity clock, all active objects first need to be notified of the new activity clock. To this end, an active object sends DGC messages containing the activity clock to all the active objects it references. The agreement on a consensus is detected by interpreting the DGC responses active objects send upon reception of a DGC message. The precise traversal is discussed in more detail below.

DGC messages flow from the referencers to the referenced active objects in order to advertise their view of the final activity clock while DGC responses flow in the opposite direction (on the same connection) to propagate the activity clock candidate for a consensus. DGC messages also contain the acceptance (or not) of the consensus candidate received in the previous DGC response. Between two active objects DGC messages and responses cannot race with application messages as they are sent over the same FIFO connection.

The reference graph traversal is not a traditional traversal where messages are forwarded as soon as they are received. On the contrary, DGC messages are sent every TTB; DGC responses are sent only in response to DGC messages. The effect is that there are no DGC phases, it is a continuous process.

The content of a DGC message is:

- **sender ID:** used to detect new referencers and to know which DGC response's final activity clock the consensus boolean refers to,
- **final activity clock:** to propagate the final activity clock throughout the reference graph,
- **consensus:** a boolean indicating the acceptance of the final activity clock received in the previous response; this is actually in response to the previous DGC response.

The consensus boolean in a DGC message is set according to these rules:

- **if the destination is the parent:** the conjunction of the consensus values of the sender's direct referencers and the local agreement of the sender,
- **if the sender has a parent which is not the destination:** whether the sender locally agrees with the final activity clock (only the local agreement).

The content of a DGC response is:

- **final activity clock consensus candidate:** the consensus attempt by the traversal,
- **has parent:** boolean indicating if the referenced active object can be a parent, this ensures that the reverse spanning tree is rooted at the originator.

The reverse spanning tree is constructed by having every active object promote one of its referenced active objects as its parent. The reverse spanning tree will conduct the consensus from the active objects to the originator. The *hasparent* boolean in the DGC response checks if the sending active object has chosen a parent or is itself the consensus originator. This ensures that the chosen parent leads to the originator.

An active object updates its view of the final activity clock in the graph using its own activity clock and the DGC messages. It also stores the last DGC message of its referencers in order to compute the consensus boolean value.

The activity clock contained in the DGC response is never used to update an active object's clock, only to try to build a consensus. For example, in Figure 4, if the cycle C1 is busy it will propagate activity clocks in the cycle C2, but the latter will never propagate activity clocks in the cycle C1. As references are oriented, C2 must not prevent C1 from being garbage collected.

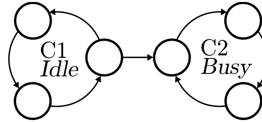


Fig. 4. Activity clocks are not propagated in DGC responses, otherwise C2 would prevent C1 from being garbage collected until C2 is garbage too

Making a Consensus. An idle active object decides that a consensus has been made when all of its referencers sent it a DGC message with the consensus boolean set for its own final activity clock. This is also the current final activity clock of the originator, and the one it sent in the previous DGC responses. The active object detecting the garbage cycle is the root of the reverse spanning tree. Let us see what this means for an active object to be in this state:

- it has propagated its final activity clock in a part (or the whole) of the recursive closure of its referenced active objects,
- the recursive closure of its referencers have all accepted this final activity clock.

An active object needs to propagate its final activity clock only in the part of its referenced active objects that belongs to the same cycle as itself, hence not necessarily all of its referenced active objects.

When is the activity clock incremented. The activity clock is incremented on these three occasions:

Active object becoming idle. This is the primary reason for the existence of the clock. During a traversal, active objects could alternate between being idle and busy (i.e. an active object receives and starts serving requests before the traversal completes) so the outcome of the traversal would be inconsistent. In this case, with the clock, active objects can be idle but still disagree on the proposed final activity clock.

Loss of a referencer. According to the rules for detecting garbage cycles, the active object that breaks the cycle is the owner of the final activity clock, provided that it is idle. As a consequence, we have to enforce that the owner of the final activity clock be in the recursive closure of referencers.

Consequently when an active object detects that one of its referencers has disappeared (i.e. it has not received DGC messages from this referencer in a TTA period), it must increment its activity clock. This is also the reason why active objects track the list of their referencers IDs.

Therefore, when an active object disappears, its referenced active objects increment their activity clocks, so that cycles without external referencers cannot agree upon an unowned activity clock. The latter could result in a cycle where all active objects have a common final activity clock, but as the owner of this final activity clock is not in the cycle, it cannot break the cycle.

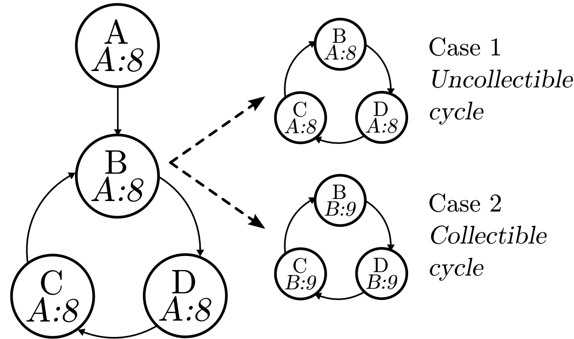


Fig. 5. The loss of a referencer must be detected to avoid uncollectible cycles

In Figure 5, the active object A references a cycle and propagated its final activity clock in this cycle. When A terminates the cycle should not keep its final activity clock that belongs to nobody in the cycle (Case 1). Instead B should notice that it lost a referencer (A) and then should increment its activity clock to obtain B:9 (Case 2).

Loss of a referenced. The reverse spanning tree built over the active object references is built by, for each activity, choosing a single referenced active object and considering it as its parent. An active object tells to its referenced active objects if its activity clock is the same as the referenced's one, but will tell to its parent if its referencers and itself agree on the final activity clock. This simulates a graph traversal with the references other than the parent being the nodes already visited, in order to avoid cyclic dependencies. To justify this traversal, let

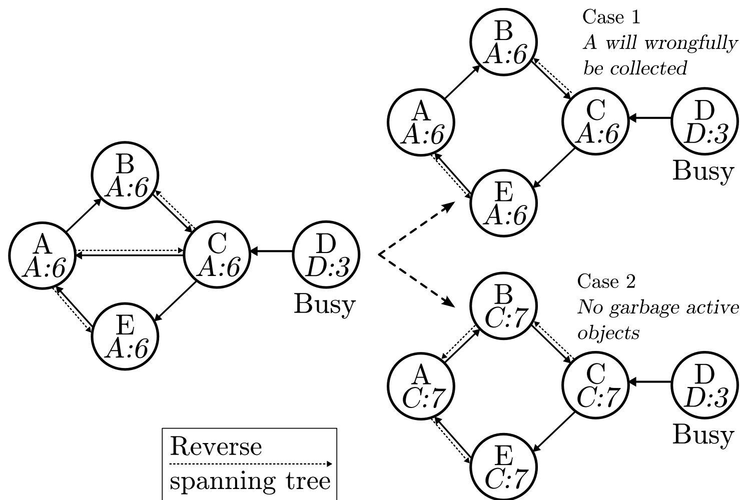


Fig. 6. The loss of a referenced must be detected to avoid collecting live cycles

us suppose instead that an active object would propagate an agreement on a final activity clock only if it received such an agreement from all of its referencers, in this case a consensus would never be reached in cycles.

Therefore, when an active object loses a reference, it may actually lose its parent. And without its parent, an active object will have nobody to tell if its referencers disagree about the final activity clock. In Figure 6, only D is busy, so it prevents the cycle from being garbage collected. C's parent in the reverse spanning tree is A, so C will tell to A that the consensus is rejected, but will not tell it to E as it is not its parent. If the reference edge from C to A disappears, nobody will say to A that the consensus is rejected, and the cycle will wrongfully be garbage collected. The solution is to increment the activity clock when a reference disappears.

The active object A will nevertheless detect the loss of a referencer (C), but detecting the loss of a referenced ensures that C does not have a final activity clock from someone else and no parent, a condition that would break the reverse spanning tree.

3.3 Algorithms

Here we show a pseudo-code version of the four DGC algorithms for: the recursive agreement of the referencers, the broadcasting every TTB, the reception of a DGC message and the reception of a DGC response. For clarity, they have been simplified by omitting the following details: the management of referencers and referenced active objects, the activity clock incrementation in the three cases seen in Section 3.2, and the error handling.

Algorithm 1. $referencers.agree(clock)$: Recursive agreement on *clock*?

for all $referencer$ **in** $referencers$ **do**
  if $referencer.clock \neq clock$ **or** $referencer.consensus =$ **false then**
    return false
return true

Algorithm 2. Every TTB on every active object AO

if $AO.isIdle()$ **then**
  if $now() - AO.lastMessageTimestamp > TTA$ **then**
    $AO.terminate()$ // acyclic garbage
  if $AO.clock.owner = AO$ **and** $AO.referencers.agree(AO.clock)$ **then**
    $AO.terminate()$ // cyclic garbage
for all $dest$ **in** $AO.referenced$ **do**
  $consensus \leftarrow AO.isIdle()$ **and** $dest.lastResponse.clock = AO.clock$ **and**
    ($AO.clock.owner = AO$ **or** $AO.parent \neq$ **nil**) **and**
    ($AO.parent \neq dest$ **or** $AO.referencers.agree(AO.clock)$)
  $dest.sendMessage(AO.id, consensus, AO.clock)$

Algorithm 3. Reception of a DGC *message* by active object AO

```
if message.clock > AO.clock then
    AO.clock ← message.clock
    AO.parent ← nil
AO.referencers[message.sender].clock ← message.clock
AO.referencers[message.sender].consensus ← message.consensus
AO.lastMessageTimestamp ← now()
hasParent ← AO.parent ≠ nil or AO.clock.owner = AO
return response(AO.clock, hasParent) // DGC response
```

Algorithm 4. Reception of a DGC *response* from ref by active object AO

```
ref.lastResponse ← response
if response.clock = AO.clock and response.hasParent and
                AO.parent = nil and AO.clock.owner ≠ AO then
    AO.parent ← ref
```

4 Discussion

4.1 Middleware Integration

The DGC algorithm has been implemented on top of the Java platform which provides a local garbage collector albeit one with a very restricted public interface. In spite of these restrictions we have decided against modifying the Java Virtual Machine or the compiler, because doing so makes applications harder to debug and hinders portability. In the rest of this section, we provide guidelines for the integration of the DGC algorithm in a middleware, taking the ProActive [7] middleware as reference.

The ProActive middleware is a Java implementation of active objects. Method calls on active objects are transparently asynchronous as they return a *future*. A future is an object that serves as a placeholder for the actual result. After an asynchronous call, execution continues on the caller side and the caller will transparently wait for the return value of the call when first using the future.

Idle Active Objects. The DGC algorithm requires the ability to decide if an active object is busy or idle. An active object is typically implemented as a server listening for requests and serving them. Hence, in the case of single threaded active objects, it is easy to decide whether an activity is currently busy serving a request or not: an active object waiting for requests is said to be idle.

Some kinds of active objects are never idle, they would be the roots in a local garbage collector. In this DGC, the roots are:

- registered active objects as anyone can look them up at any time, this is identical to marking the registry as a root when the implementation permits (registry implemented as an active object),

– dummy active objects used as referencers when non active code references an active object.

The second item needs more explanation. It was stated that the referencers of an active object are active objects too. This is not always the case for example when some `main()` method acquires a reference to a remote active object, it is not part of an active object. To this end, the middleware creates a dummy active object that has no activity but provides the non functional properties needed by the middleware and the DGC. This allows the assumption that all referencers are active objects.

Reference Orientation. Reference edges are oriented, therefore a busy referenced active object will not prevent an idle referencer from being garbage collected, as seen in Figure 4. This implies that a referenced active object may not be able to update a future with its value for its referencer (caller) if the latter was garbage collected. This property is not necessary for garbage collection but fits with the middleware as the receipt of an updated future cannot wake up an idle activity by itself. An active object waiting for a future is busy as waiting for a future can only be done during the service of a request. Hence, this property is accepted as it is more aggressive towards garbage. Nevertheless, we could get rid of this property by dropping the orientation of references edges in a middleware where the reception of a future can wake up an idle activity, with a callback mechanism for example.

The Process Graph. If the no-sharing property is not desired or available we cannot reliably build a local reference graph without stopping all the threads or modifying the local garbage collector. Therefore, only a coarser graph would be available in this case: the graph of address spaces (processes). The bounds of address spaces are clearly identifiable as a serialization and a deserialization step are always needed to cross them. The graph of processes contains the same vertices as the reference graph, to wit, all objects in the distributed system. The edges (x, y) of this graph (P) can be determined from the reference graph (R) with the following formula:

$$\forall (x, y) \in R, \forall x' \in Proc(x), \forall y' \in Proc(y), (x', y') \in P \quad (2)$$

x, x', y, y' are active objects and $Proc(x)$ is the process hosting the active object x. Using the process graph instead of the reference graph makes the DGC algorithm implementable on a broader range of middlewares, but limits its ability to find cyclic garbage to whole processes. For instance, a garbage cycle spanning some processes where some active objects are still live will not be collected if only the process graph is available.

The DGC of RMI uses another graph: each remote object maintains the list of stubs targeting this remote object, as dictated by the reference listing approach. Edges in this graph thus connect vertices of different types: stubs to remote objects, hence this graph is not suitable for cycle detection as it does not contain edges from remote objects to stubs.

4.2 Asynchrony - Real-Time Needs

This DGC algorithm is not fully asynchronous because, as seen in Section 3.1, it requires an upper bound on the communication time. Fully asynchronous distributed garbage collectors, while resistant to transient failures, have the limitation that undetected failures can prevent garbage collection as these are indistinguishable from transient failures. Consequently, and like the DGC of RMI, our algorithm is hard real-time as a missed deadline can cause a malfunction in the application if an active object is wrongfully garbage collected. However, the synchronization between active objects is very loose as it is represented by $TTA - 2 * TTB$. As this difference can be made as large as needed, deadlines can therefore be pushed arbitrarily far away, obviously slowing down the DGC.

For deadlines in the range of minutes (the common case) care must be taken to avoid letting TCP timeouts be the cause of missed deadlines. To this end, a basic TCP streaming socket should be avoided for the broadcasting because of its synchronous nature as blocking on a socket will delay for no reason the remaining of the broadcast. The alternative is to broadcast in parallel using either non-blocking I/O, asynchronous I/O or threads.

Another cause of missed deadlines can be the pauses caused by the local garbage collector. This was significant enough to justify the increase of the default lease time of the RMI DGC from one minute to one hour [8] in Sun Java 1.6. In our experiments, we have not yet found the need for these long deadlines.

4.3 Complexity Analysis

The presented distributed garbage collector works by building the reference graph and then a reverse spanning tree. Nevertheless, at no point in time is the whole graph known by a single active object. Active objects keep information only about their immediate neighbors (referencers and referenced active objects).

To sum up, for each active object, the added size in data structures is proportional to the numbers of referencers and referenced active objects.

DGC messages and responses between active objects are of fixed size and are exchanged every TTB between every couple of referencer/referenced active objects.

Figure 7 shows an abridged example with the main steps both in the presence of garbage, and when a single object prevents the formation of garbage. The time complexity of finding garbage can be determined by reviewing the steps (as shown in Figure 7) needed to detect a garbage cycle:

1. propagating the final activity clock through the reference graph (DGC messages),
2. propagating the consensus candidate through a reverse spanning tree (DGC responses),
3. propagating the consensus decision through the reference graph (DGC messages).

These steps proceed in an unsynchronized parallel fashion as the construction of the reference graph in the different active objects may be at different steps.

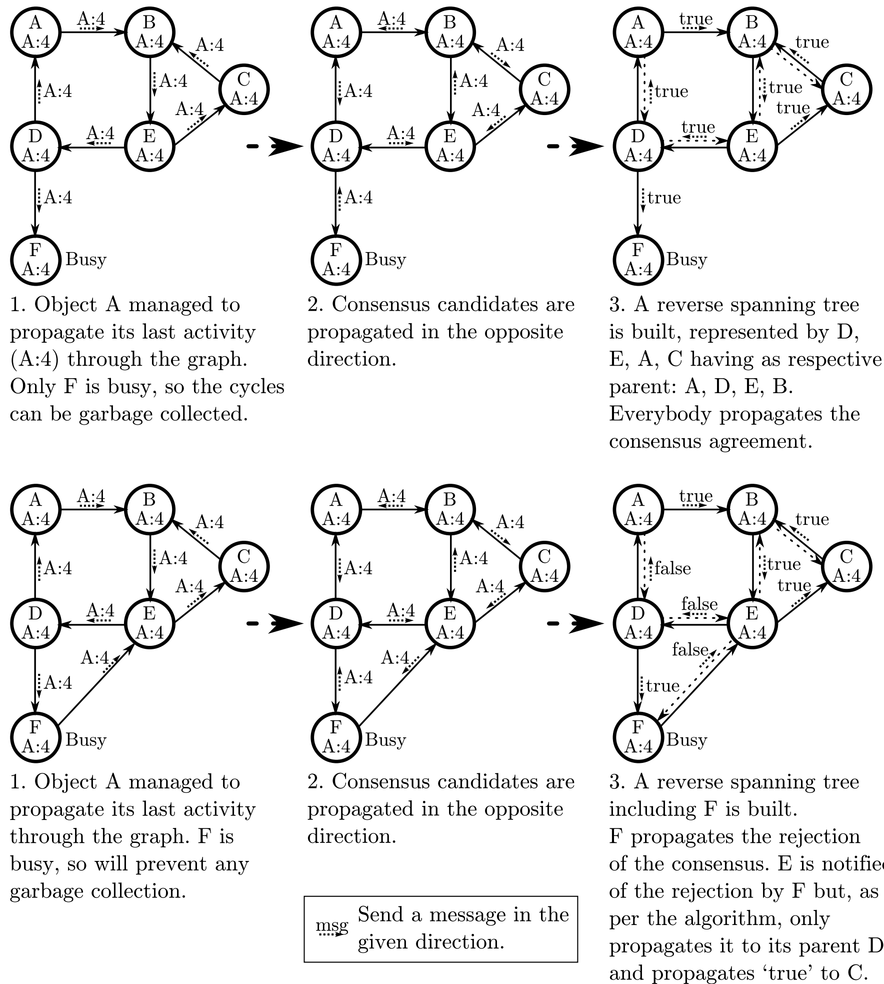


Fig. 7. Two cycle detection examples: a garbage compound cycle, and a single live object preventing the compound cycle from being collected. The three steps are separated but their execution is actually unsynchronized.

To evaluate the time complexity, we introduce h as the maximum height of all spanning trees and reverse spanning trees for the distributed system to consider, which can be bigger than the diameter of the reference graph. The maximum height spanning tree represents the time needed to propagate DGC messages while the maximum height reverse spanning tree represents the time needed to propagate the DGC responses; they may not be equal as the reference graph may not be exclusively made of cycles. Then the order of the time to detect a garbage cycle is in $O(h * TTB)$.

After detecting a garbage cycle, to evaluate the time needed to fully collect it we need to take into account an optimization in the algorithm. For simplicity

reason, this optimization has not been introduced in Section 3.2 as it is not required by the algorithm. When a consensus is made, the active object finding itself in a dead cycle waits during TTA before terminating. During this time it stops sending DGC messages as it does not need anymore to keep its referenced active objects alive, and it gives DGC responses indicating that a consensus has been reached in order to propagate the information through the referencers. This behavior is not required by the algorithm as it could simply terminate a single active object and expect the acyclic or cyclic garbage collector to go on with the remaining active objects. Nevertheless, cyclic data structures can contain sub-cycles so that the acyclic garbage collector cannot take care immediately of the remaining active objects. For this reason, and as will be shown by the benchmarks, we argue that propagating the result of the consensus is an important optimization, otherwise the acquired knowledge is partially dropped and the consensus process must start again for the sub-cycles. Therefore, a fourth step is added to go from the detection of a garbage cycle to its full collection:

4. propagating the consensus acceptation through a reverse spanning tree (DGC responses).

With this optimization in mind, a precise order of the time to garbage collect a cycle is in $O(h * TTB) + TTA$. The added TTA recalls that all active objects in cyclic garbage wait during TTA before terminating, so the last one will wait needlessly during TTA.

5 Experiments

For the experiments, the DGC algorithm has been implemented in the ProActive middleware.

In all of the following benchmarks, we measured the total network traffic by using an instrumented local SOCKS server [9] on every machine. All JVMs are instructed to forward all of their connections to the local SOCKS server which then simply forwards the connection and prints its transferred size at the end. Thus, our communication numbers only include the TCP payload but are not impacted by unrelated network traffic. Accounting only TCP is enough as all communications are over RMI, and the DNS usage is extremely low. Also, DGC messages and responses transmitted inside a single JVM are not accounted as they are directly passed by reference.

5.1 Hardware, Network and Software Environment

The following experiments have been realized on clusters in three sites of the French Grid'5000 [10] platform: Bordeaux, Sophia and Rennes. We used 49 nodes from Bordeaux, 39 from Sophia and 40 from Rennes, totalizing 128 nodes. Hardware and software details follow:

– **Bordeaux**: AMD Opteron 248 or Intel Xeon EM64T 3GHz, Dual CPU, 2G RAM, Gigabit Ethernet, RTT latency: 0.2ms. Debian 4.0 x86_64, Linux-2.6.18, Sun Java 1.5.0_10.

- **Sophia**: AMD Opteron 2218, Dual CPU, 4G RAM, Gigabit Ethernet, RTT latency: 0.1ms. Rocks 3.3.0 x86_64, Linux-2.4.21, Sun Java 1.5.0_10.
- **Rennes**: Intel Xeon 5148 LV, Dual CPU Dual Core, 2G RAM, Gigabit Ethernet, RTT latency: 0.1ms. Ubuntu 6.10 x86_64, Linux-2.6.19.1, Sun Java 1.5.0_10.

The RTT network latencies between sites are as follows: 8ms between Rennes and Bordeaux, 10ms between Bordeaux and Sophia, 20ms between Rennes and Sophia.

5.2 NAS Benchmarks

We used a ProActive/Java implementation of some of the NAS Parallel Benchmarks [11], this implementation uses explicit termination of active objects, therefore we know the earliest time at which objects could be garbage collected. Hence, we could measure both the overhead in computation time and the time required by the DGC to collect all the active objects at the end.

This NAS Benchmarks implementation is the worst case in terms of communication overhead for the DGC algorithm as every active object has a reference to every other active object because of global barriers. By this account, every TTB there is a communication between every couple of active objects. Nevertheless, the DGC algorithm itself is not exercised much as the reference graph is static: references are created at initialization time and are not changed thereafter.

This benchmark shows the impact of the DGC on a real workload. The DGC parameters are set in a slightly aggressive manner as the TTB is set to 30 seconds and the TTA to 61 seconds as per the formula in Section 3.1.

The tested kernels are:

- **CG:** A conjugate gradient method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix.
- **EP:** An embarrassingly parallel kernel.
- **FT:** A 3-D partial differential equation solution using FFTs.

The benchmarks are class C on 256 active objects with a round-robin distribution. We show the average and standard deviation of the total bandwidth consumed over 3 runs. The overhead is evaluated as $\frac{T_{dgc}-T_{nodgc}}{T_{nodgc}}$. The DGC algorithm is independent of the communication pattern, so a heavily communicating kernel like CG or FT will experience a lower overhead than a lightly communicating kernel like EP.

As expected, the bandwidth overhead of a lightly communicating benchmark like EP is very high as most of the communication is caused by the DGC algorithm. The DGC time is the time between when the benchmark has its result and when the DGC collects all the active objects. As the TTB was set to 30 seconds, the benchmark shows that the 256 active objects are collected in 15 or 17 DGC messages/responses broadcasting iterations. This speed is caused by two factors. The first one is the optimization that consists in propagating a notification that a garbage cycle has been found to all the members of the cycle. Without this

Kernel	No DGC		DGC		Overhead
	Average	Std. dev.	Average	Std. dev.	
CG	194351.81 MB	3965.60 MB	223639.83 MB	1532.94 MB	15.07 %
EP	69.75 MB	0.56 MB	717.92 MB	47.00 MB	929.28 %
FT	41999.48 MB	3383.64 MB	48187.78 MB	873.03 MB	14.73 %

Fig. 8. Bandwidth overhead

Kernel	No DGC		DGC		Overhead	DGC time	
	Average	Std. dev.	Average	Std. dev.		Average	Std. dev.
CG	3529.45 s	27.11 s	3190.00 s	5.41 s	-9.62 %	534.31 s	26.77 s
EP	8.36 s	0.54 s	8.37 s	0.44 s	0.12 %	530.41 s	42.40 s
FT	424.40 s	7.31 s	427.66 s	2.46 s	0.77 %	457.41 s	3.61 s

Fig. 9. Time overhead

optimization, after each consensus, a single active object is collected and the consensus must start again. The other reason for the speed is the fact that the reference graph in this benchmark is a complete graph, so consensus attempts are quickly propagated throughout the graph.

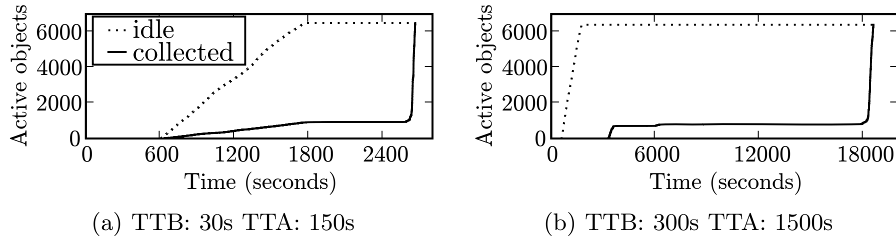
The negative time overhead for CG can be explained by the differences in the network usage caused by the DGC algorithm. The ProActive middleware is built on top of RMI and the DGC algorithm does its broadcast in a separate thread. By default, RMI closes the sockets it opens after 15 seconds [12] of inactivity. Consequently, the DGC broadcasting implicitly opens the sockets in its own thread and the benchmark code will not be slowed down by the latency of opening a TCP connection. Experimenting with a very high `connectionTimeout` value, hence preventing the closing of RMI sockets, gives a positive overhead. The running time rises from 2488.2 seconds without the DGC to 2499.23 seconds with the DGC, giving an overhead of 0.44%. However, the drawback of this kind of tuning is the increase in resource usage as sockets are accumulated and never closed.

These NAS benchmarks have shown that the time overhead of the DGC algorithm is insignificant, the bandwidth overhead in slightly communicating applications is important but does not result in a slowdown of the application.

5.3 DGC Torture Test

A special purpose test to stress the DGC algorithm was made. This is a simple master/slave application where slaves continuously exchange references between themselves and the master during at least ten minutes, then become idle. Thus a very complex reference graph is created and the DGC has to destroy it after the ten minutes of intense activity.

The only data exchanged by active objects consists in the remote references, so the communication overhead of the DGC is predominant. We measured the impact of changing the TTB and TTA values on the total communication size and total time.



(a) TTB: 30s TTA: 150s

(b) TTB: 300s TTA: 1500s

Fig. 10. Evolution of the number of idle and garbage collected active objects

Experiments have been realized in the same hardware and software environment as the NAS with 128 machines, each of them hosting 50 slave active objects. The total number of active objects is therefore 6401, including the master active object.

In Figure 10 we can see that active objects start to become idle after their last running iteration 600 seconds after being started. During this phase, some acyclic garbage is quickly reclaimed, then the consensus is being made. Finally, thanks to the optimization that consists in notifying the referencers when a garbage cycle is found the whole graph is rapidly collected. The total bandwidth consumed for the TTB values 30s and 300s are respectively 1699MB and 2063MB. For reference, without the DGC algorithm, the total bandwidth consumed is 228MB and the last active object finishes after 1718 seconds.

This benchmark has shown that the DGC algorithm scales to a large number of active objects with a very dynamic reference graph.

6 Related Work

One can find lots of distributed garbage collector algorithms in the literature, we will concentrate here on a representative sample. We exclude special purposes algorithms having requirements like a central server or migration support.

In the DGC algorithm [4] by Veiga and Ferreira, cycle detection messages traverse the reference graph and grow information about it. Referencers are called dependencies and represent the still unknown part of the graph. At each step of the traversal, the cycle detection message may add some unresolved dependencies or may resolve some of them depending on the traversed object. A garbage cycle is identified as such when it has no more unresolved dependencies; one of its elements is then terminated. A drawback of this approach is that the growth of the message is limited only by the total size of the distributed system, so the communication overhead can become large.

The DGC algorithm presented by Le Fessant [13] is based on the propagation of marks from referencers to their referenced objects. Marks have a color: black for activities, white for idleness, and gray if both colors were encountered. These marks are generated by local roots and remote objects, a cycle is detected when a remote object receives only its own mark with the white color. This algorithm requires a tight cooperation with the local garbage collector. Unfortunately, no information is provided about the time complexity for the collection of cycles.

Lang, et al. [14] describe a DGC algorithm based on the construction of hierarchical groups over the reference graph. Each group performs a mark and sweep to destroy cycles it fully contains. The determination of groups leaves room for optimization by the application. But, as cycles are collected depending on the formed groups, no indication is given of the time needed to detect a cycle.

7 Future Work

7.1 Dynamic Parameters

The presented algorithm is configured by only two parameters: TTB and TTA. They are supposedly constant and known to every active object in the distributed systems. Two improvements are considered:

- allowing each active object to specify its own TTB and TTA value,
- dynamically adjusting the TTB and TTA in respect to the presence of suspected garbage and according to the communication rate.

The first envisioned improvement addresses the cases of applications with disparate garbage collection needs. A distributed application can be composed of a static part and a more dynamic one. The more dynamic part would benefit from smaller TTB and TTA, resulting in faster garbage collection, while the static part would lower its DGC overhead by increasing TTB and TTA.

Then, dynamically adjusting them becomes attractive in order to augment the broadcasting frequency when some garbage is suspected, i.e. when an active object gets a parent and some of its referencers agree with the consensus, or lower it when the distributed system is highly loaded.

7.2 Breadth First Spanning Tree

The DGC algorithm makes a consensus by traversing a reverse spanning tree, therefore the height of the reverse spanning tree influences the speed of finding garbage cycles. Shallow reverse spanning tree are thus preferred as their traversal is faster. Currently, the reverse spanning tree is constructed by choosing the first referenced active object with the right response as the parent. This produces shallow trees by relying on the time to reply, that is, one edge should be traversed faster than two unless the broadcasts are fortuitously synchronized. Nevertheless, a proper algorithm to ensure that the height is minimal is being considered.

8 Conclusion

We have shown a practical algorithm and its implementation for complete distributed garbage collection including cycles of garbage. Though this algorithm can be adapted to any distributed environment, it is particularly precise in a middleware featuring the no-sharing property, as in the active objects model. Cycle detection is based on a consensus reached by exploring a reverse spanning

tree, but the algorithm does not require more connectivity than the original application. To summarize, the algorithm only relies on the knowledge of idle processes, and of their remote references. Also, benchmarks have shown the scalability of the algorithm in grid contexts. All these properties make our algorithm particularly adapted to any middleware having only a limited control over the local garbage collector, and in particular for grid computing.

Acknowledgements. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners.

The authors would like to thank Christian Delbé and Stijn Mostinckx for their extensive reviews and helpful comments.

References

1. Wollrath, A., Riggs, R., Waldo, J.: A Distributed Object Model for the Java System. Computing Systems 9(4), 265–290 (1996)
2. Birrell, A., et al.: Distributed Garbage Collection for Network Objects. Digital, Systems Research Center (1993)
3. Plainfosse, D., Shapiro, M.: A survey of distributed garbage collection techniques. In: Proceedings of the International Workshop on Memory Management (1995)
4. Veiga, L., Ferreira, P.: Asynchronous Complete Distributed Garbage Collection. In: IPDPS 2005. Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, vol. 01, IEEE Computer Society Press, Los Alamitos (2005)
5. Caromel, D., Henrio, L.: A Theory of Distributed Object. Springer, Heidelberg (2005)
6. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
7. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Deploying, Composing, for the Grid. In: Grid Computing: Software Environments and Tools, Springer, Heidelberg (2006)
8. RMI gcInterval is too short by default. Sun Bug Database (2004), http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6200091
9. Nylon. SOCKS 4 and 5 server, http://monkey.org/~marius/pages/?page=nylon
10. Cappello, F., Caron, E., Dayde, M., Desprez, F., Jeannot, E., Jegou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Richard, O.: Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In: Grid'2005 Workshop, Seattle, USA, November 13-14,2005, IEEE/ACM (2005)
11. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. The International Journal of Supercomputer Applications (1995)
12. ConnectionTimeout property, http://java.sun.com/javase/6/docs/technotes/guides/rmi/sunrmiproperties.html#connectionTimeout
13. Le Fessant, F.: Detecting distributed cycles of garbage in large-scale systems. Principles of Distributed Computing (PODC), Rhodes Island (August 2001)
14. Lang, B., Queinnec, C., Piquer, J.: Garbage collecting the world. In: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 39–50 (1992)