A distributed garbage collector for active objects

Isabelle Puaut

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Distributed Garbage Collector for Active Objects

Isabelle Puaut

N° 2134

Décembre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

INRIA
RENNES

A Distributed Garbage Collector for Active Objects

Isabelle Puaut

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet LSP

Rapport de recherche n ° 2134 — Décembre 1993 — 24 pages

Abstract: This paper introduces an algorithm that performs garbage collection in distributed systems of active objects (i.e., objects having their own threads of control). The proposed garbage collector is made of a set of local garbage collectors, one per node, loosely coupled to a global garbage collector. The novelties of the proposed garbage collector come from the fact that local garbage collectors need not be synchronized with each other for detecting garbage objects, and that faulty communication channels are tolerated. The paper describes the proposed garbage collector, together with its implementation and performance for a concurrent object-oriented language running on a local area network of workstations.

Key-words: Garbage collection, concurrent object-oriented languages, distributed systems, active objects.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Un Ramasse-Miettes Distribué d'Objets Actifs

Résumé : Nous proposons dans ce document un algorithme de détection des miettes adapté aux systèmes distribués d'objets actifs (i.e., possédant leurs propres fils de contrôle). Le ramasse-miettes proposé est constitué d'un ensemble de ramasse-miettes locaux, un pour chaque nœud de l'architecture, et d'un ramasse-miettes global au système. La principale originalité de l'algorithme proposé réside dans le fait qu'il n'est pas nécessaire que les ramasse-miettes locaux se synchronisent pour la détection des miettes. En outre, les communications non-fiables sont supportées. Ce document décrit l'algorithme de ramasse-miettes proposé, ainsi que sa mise en œuvre et ses performances pour un langage de programmation concurrent à objets s'exécutant au dessus d'un réseau local de stations de travail.

Mots-clé : Ramasse-miettes, langages de programmation concurrents à objets, systèmes distribués, objets actifs.

1 Introduction

Object-oriented languages are now recognized as powerful tools for the design of large and complex software. In particular, they provide a sound basis to develop applications that are easy to maintain and reuse. Dynamic object creation raises the issue of managing objects in memory. Freeing the memory used by objects may either be done directly by the programmer (*manual* memory management), or by the language run-time environment (*automatic* memory management). In the former case, its the responsibility of the user to decide when an object is no longer needed, and to free the resources is uses, while in the latter case, the language run-time support provides a tool, named *garbage collector* [1], which detects and reclaims unneeded objects without the programmer's intervention. As notably argued in [2], the provision of a garbage collector by the run-time support of an object-oriented language helps the development of robust software for the following reasons. Programmer-controlled memory management is notoriously error-prone: the programmer may forget to free a resource that is no longer used, or free a resource that is still used. Both mistakes are difficult to detect and recover from, especially in systems managing persistent data. Moreover, the provision of a garbage collector by the run-time support of an object-oriented language makes programs shorter, as they are no longer concerned with memory management, and thus easier to maintain.

The increasing use of parallel and distributed architectures has motivated the integration of concurrency in object-oriented languages. There are (at least) two ways for introducing concurrency in an object-oriented framework: first, as notably discussed in [3], the process notion may be integrated within the object notion, thus leading to *active* objects; second, as done for example in Presto [4], parallelism may be introduced through system libraries defining classes to be used for implementing concurrency control (i.e., thread and synchronization classes), thus leading to *passive* objects. As argued in [5], garbage collection in systems of active objects (actors in the mentioned reference) is critical, as active objects not only consume memory but also processor resources. Hence, it is imperative that garbage active objects are identified quickly. As developed later, this adds complexity to the task of garbage collection compared to sequential systems, as both state and activity of objects have to be considered. Furthermore, if the concurrent object-oriented language runs on a parallel or distributed architecture [6],

providing garbage collection becomes crucial for the development of large distributed software, since a distributed resource management scheme is harder to design and implement than a centralized one.

This paper proposes a garbage collection algorithm that applies to distributed systems of active objects. The proposed garbage collector is composed of a set of local garbage collectors, one per node, loosely coupled with a global garbage collector. The local garbage collectors detect and reclaim garbage objects of the node on which they are running by using only information local to that node. Periodically, each local garbage collector sends information to the global garbage collector through message passing; the global garbage collector merges the informations sent by the local garbage collectors in order to record a global snapshot of the system state relevant to garbage collection. The key novelty of our proposition is that the local garbage collectors need not synchronize with each others for detecting garbage objects, even when sending information to the global garbage collector. Moreover, the proposed garbage collector can be easily extended to cope with an unreliable environment. The remainder of this paper describes with more details our proposition, and is organized as follows. Next Section sketches the specifics of garbage collection in systems of active objects. Section 3 then presents the proposed distributed garbage collector. In particular, its use in a large scale distributed system with faulty communication channels is considered. Its implementation within the distributed run-time system of a concurrent object-oriented language, as well as performance measures, are given in Section 4. The proposed garbage collector is compared with related work in Section 5. Finally, conclusions are given in Section 6.

2 Garbage Collection in Systems for Active Objects

As indicated in [5], the usual definition of garbage in systems managing passive entities, which is based on reachability from *root* cells [1], is not appropriate for systems dealing with active objects. Thus, after a presentation of the considered active object model, the definition of garbage objects for this computation model is given. Then, an existing solution in a non-distributed setting, used by our proposition, is briefly described.

2.1 Object Model

An object is an entity composed of data and of one or several threads of control that operate on that data. An object's data (or object *state*) is a sequence of memory cells, each cell containing either an atomic value (e.g., an integer), or the name of another object (or *reference*). An object is said to be *running* if at least one of its threads of control is running, while it is said to be *inactive* when all its threads of control are blocked. An object O_1 may activate another object O_2, and thus make O_2 running, through either blocking or non-blocking method calls. This arises when O_1 is running and its state embeds a reference on O_2. When activating O_2, O_1 may send to O_2 as parameters a subset of its data. New objects may also be created, the creator obtaining a reference on the newly created object. Notice that the computation model described above is quite general, and is used in many concurrent object-oriented languages (for example [3]). Therefore, the proposed garbage collector may be retained for a wide range of distributed run-time supports for concurrent object-oriented languages based on active objects.

The state of a system with active objects can be depicted as a graph whose nodes represents objects and whose directed edges represent references embedded within objects. Figure 1 shows an example of such graph; in the figure, running objects are drawn as circles, inactive objects are drawn as squares, and active objects interacting with the external world, like for instance I/O devices or external naming objects, are drawn as triangles.

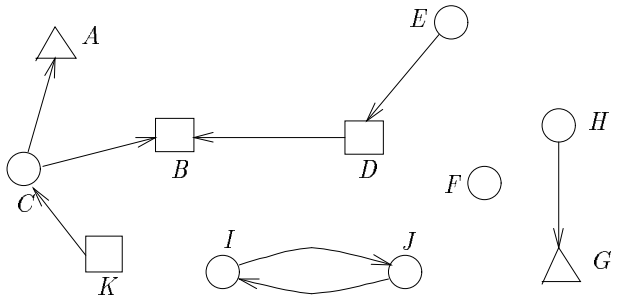

Figure 1: A System of Active Objects

The topology of the graph may change due to interactions between objects. First, an object may become running if it is activated by another running object: for instance in Figure 1, object D may be activated by object E. Second, when activating an object, a running object may send it a subset of

its data, thus adding edges to the graph. For example in Figure 1, object C may send to object B a reference on object A, thus adding an edge from B to A in the graph. New edges may also result from object creation.

2.2 Detecting Garbage in Systems of Active Objects

Detecting garbage in systems of active objects was first addressed in the framework of Actor-based languages [7], and detailed later in [5]. The reader is referred to these two references for a detailed description of the issue of garbage collection in systems of active objects: only an informal definition of garbage is given here. Intuitively, an object is garbage if its absence from the system cannot be detected through external observation, excluding memory and processor resources consumption. Actual detection of garbage relies on the introduction of *root* objects, depicted by triangles in Figure 1; these objects are always needed, as they have the ability to directly interact with the external world. Briefly stated, an object is garbage if it cannot potentially neither call a root object, nor be called by a root object; in other terms, an object is garbage if it cannot potentially interact with a root object.

Let us illustrate this definition on Figure 1. Object H embeds a reference on the root object G and is running, and thus it may call G; therefore, H is not garbage. Similarly, C is not garbage, since it may call the root object A. Objects I,J and F are garbage, as they are insulated from the rest of the object graph. K is inactive, and can not be activated in the future, because no object embeds a reference on it; thus, K is garbage. Objects B, D and E are not garbage, because they may be activated and then call a method on a root object. For example, let us consider object B; if C calls a method on B and gives it a reference on A as a parameter, B may then call the root object A, and thus is not garbage.

Note that an object that cannot call a root object at a given time (either because it is inactive or does not embed a reference on a root object) may do so later since it may get a reference on a root object from another object through parameter passing (e.g., see object B in Figure 1). Therefore, there exists a set of transformations that changes the graph of objects from a representation of what can *currently* happen to what can *potentially* happen. Note also that a key property of garbage objects is that they cannot become non-garbage in the future (*stability* property). This is because an object becomes garbage only if there is no possibility of interaction between it and

a root object. Therefore, once an object is garbage, there is no sequence of transformation which could cause it to become non-garbage.

As detailed in [5], it is significantly more difficult to detect garbage objects in systems of active objects as both the state and activity of objects have to be considered. In particular, both the traditional *mark and sweep* and *reference counting* techniques, that are based on reachability from root objects, are not directly suited to systems of active objects. If mark and sweep were used for the system depicted in Figure 1, all the non-root objects would be incorrectly marked as garbage, because they are not reachable from a root object. Similarly, if reference counting were used, objects E and H would be wrongly considered as garbage, as their reference count is zero. Moreover, reference counting may miss garbage objects; for example objects I and J are garbage even though their reference count is not zero.

2.3 An Existing Non-Distributed Solution

Two algorithms are proposed by Kafura in [5] for detecting garbage in non-distributed systems of actors. The algorithms consist in marking the objects using three colors (white, grey, and black) which, on marking termination, have the following meanings:

- **white**: objects colored white can not interact with a root object.
- **grey**: objects colored grey could interact with a root object if they were activated, but cannot become running.
- **black**: objects colored black are non-garbage. They are either root objects or can interact with a root object.

Underlying the marking algorithms is a set of five coloring rules, which are applied until no new marking can be done. Furthermore, the colors of objects can only be darkened. Initially, all objects are marked white, except for root objects, which are marked black. Rule 1 marks black the objects that can directly be activated by a black object. Rule 2 marks black the objects that can directly activate a black object. Rule 3 marks black the objects that can directly activate a grey object. Rule 4 marks grey the inactive objects having a reference on a black object. Finally, rule 5 marks grey the inactive objects that could (if they were activated) call a method on a grey object. When no new marking can be done, all black objects are non-garbage, while

grey and white objects are garbage. Notice that an object is marked at most twice: once grey and once black; the termination of marking follows.

The two marking algorithms proposed in [5] and based on the above marking rules both have a worst case time complexity of $O(p^2)$, and a space complexity of $O(p)$, where p is the total number of objects in the system.

3 A Distributed Garbage Collector of Active Objects

In the following is proposed an extension of the above garbage collection algorithm to a distributed framework. The description focuses on the *detection* of garbage, which is the most critical part of garbage collection; reclaiming the resources used by a garbage object consists in stopping the threads of control running within an object, and then freeing the memory used by the object. The issue of reclaiming the resources occupied by an object is tackled in Section 4. First, the assumptions made on the underlying system are given. The two components of the garbage collector, namely *local* and *global* garbage collectors are then described in turn. The properties of the resulting protocol are sketched. Finally, some extensions of the protocol are proposed.

3.1 Overview of the Proposed Garbage Collector

The system that is considered is made of a collection of machines, or *nodes*, connected by a communication network. In a first approach, communication channels are assumed to be reliable and FIFO (two successive messages sent from a node N to a node M are received by M in the order sent); these restrictions will be removed in Section 3.5. It is also assumed that nodes do never crash.

Each object resides at a particular node, which is the *owner* of the object; the object is said to be *local* to that node. Objects are referred to uniformly regardless of their location by using names that are unique in time and space. References to non-local objects are called *remote references*, and refer to objects via their unique name. Objects can contain both remote references and references on local objects. Each node N keeps track of the names of the following objects it owns: objects embedding remote references, remotely referenced objects, and root objects, by means of the three following respective

tables: *Out_Table*, *In_Table*, and *Root_Table*. Figure 2 depicts the system of active objects given before, where the objects are now distributed on two nodes N and M.

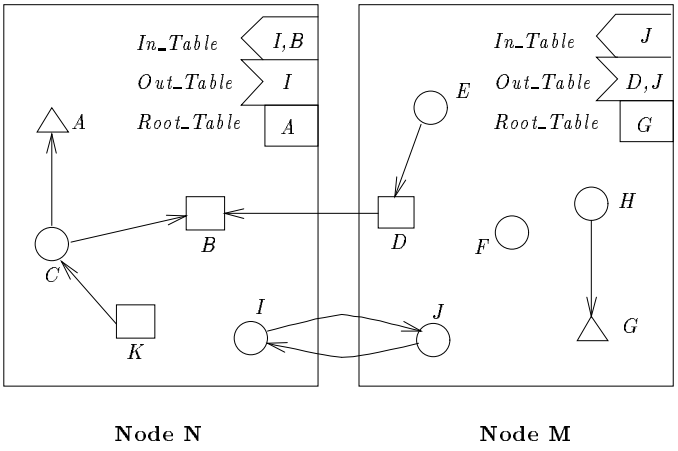

Figure 2: A Distributed System of Active Objects

Objects interact through message passing on the communication network. Due to parameter passing on object activations, messages may embed object unique names. When a message containing references is sent from one node to another, *In_Table* and *Out_Table* are updated accordingly. An object name remains stored in *In_Table* or *Out_Table* until the garbage collector finds it to be no longer needed, as detailed later.

Our strategy for garbage collection consists in having nodes responsible for doing local garbage collection and managing the resources for the objects it owns. A *local garbage collector* is associated to each node N and detects garbage by using only local information (garbage objects detected by the local garbage collector are called *local* garbage objects). A centralized garbage collection service, called *global garbage collector*, gathers information sent by the local garbage collectors in order to detect the remaining garbage objects (called *global* garbage objects). Both the local and global garbage collectors use the marking algorithm presented in Section 2. Marking is done without halting the computation. For space considerations, we do not consider this aspect, which is tackled in [8].

3.2 Local Garbage Collector

A local garbage collector is running on each node N, and is both responsible for detecting local garbage and for sending information to the global garbage collector. The following paragraphs focus on these two points.

3.2.1 Detection of local garbage

In order to detect local garbage at node N, objects that may interact with remote nodes must always be retained, since it is not known if they are able to interact with root objects owned by other nodes. Let us consider an object x owned by a node N that is (potentially) referenced by an object y owned by another node (x's name is stored in *In_Table* at node N). Object x must be retained even if it is inactive because it may be activated by y. A running object whose name is stored in *Out_Table* must be retained because it may activate a remote object. An inactive object containing a remote reference must be retained only if it can potentially be activated.

Consequently, the marking rules presented in the previous section are used in the following way. Initially, root objects of N, as well as objects whose names are stored in *In_Table*, and running objects whose names belong to *Out_Table* are marked black. Inactive objects whose names are stored in *Out_Table* are colored grey. All the other objects of node N are colored white. When marking is complete, all white and grey objects are garbage and can be reclaimed. If an inactive object whose name is stored in *Out_Table* is found to be garbage, its name is removed from *Out_Table*.

		Node N	**Node M**
Initialization:	black	{A,B,I}	{J,G}
	grey	{}	{D}
	white	{C,K}	{H,E,F}
End of Local Collection:	black	{A,B,C,I}	{D,E,G,H,J}
	grey	{K}	{}
	white	{}	{F}
Local Garbage		{K}	{F}

Progress of local garbage collection for the two nodes of Figure 2 is shown above. At the end of marking, objects K and F are identified as garbage and can be reclaimed.

3.2.2 Collaboration to the detection of global garbage

The objects needed for detecting global garbage are the objects that can potentially communicate with remote objects, either directly or indirectly. Periodically, a local garbage collector running on a node N identifies these objects, and then sends the objects references, as well as references contained within them, to the global garbage collector, that processes them asynchronously.

The objects needed for the detection of global garbage are identified by applying the marking rules given in Section 2. Initially, running objects whose names belong to *In_Table*, together with objects whose names are stored in *Out_Table* are marked black. Inactive objects whose identifiers belong to *In_Table* are marked grey. The other objects are marked white. When marking is finished, the structure of black objects is used to detect global garbage (objects A, B, C and I for the node N of Figure 2). The references on black objects, as well as the references contained in these objects form a subgraph of the node's object graph. The subgraph is sent to the global garbage collector as a list of edges, where each edge is sent as a pair *(s,d)* that identifies the source and destination objects of a reference embedded in a black object; the source and destination objects are identified by their unique name and their activity attribute (i.e., *running*, *inactive* or *root*). In our example, we get for node N the list $(< A, root >< B, inactive >)$. $(< C, running >< A, root >)(< I, running >< J, unknown >)$. As it is not known on node N whether object J is running or not, the activity attribute of J is sent as *unknown* to the global garbage collector. The actual activity attribute of J will be known by the global garbage collector when merging the informations sent by all the local garbage collectors.

3.3 Global Garbage Collector

The global garbage collector is a logically centralized service that maintains global snapshot of the system's state relevant to garbage collection. This global snapshot is built by merging the subgraphs sent by the local garbage collectors. Since local garbage collectors do not synchronize with each other when sending information to the global garbage collector, the global garbage collector must be able to detect whether G represents a consistent vision of the system state. This issue is examined before giving a detailed description of the global garbage collector.

3.3.1 Global consistent states and garbage collection

Let us consider a distributed system composed of n processes p_i, $1 \leq i \leq n$ communicating through message passing on reliable communication channels c_{ij}, $1 \leq i,j \leq n$, where c_{ij} denotes the communication channel between p_i and p_j. Message transmission is assumed to be finite, but not necessarily bounded. Each process p_i has a private local state. The state of any communication channel is the set of messages sent by process p_i and not yet received by process p_j. The execution of a process consists of a sequence of *events*. The events are classified according to three categories: *send*, *receive*, and *internal* where internal events modify only the process local state. A global system state is comprised of the processes local states and the communication channels states. A global state is said *consistent* (also called *global snapshot* [9, 10]) if for each message captured as received in a process local state, the message is captured as sent in the sender local state. An interesting feature of global snapshots is that they can be used to detect stable properties [9]: if a stable property P holds in a global snapshot, it holds in the current state of the system.

Consistency of a global system state may be determined by timestamping events using *vector timestamps* [10, 11]. Each process p_i has a clock VT_i consisting of a vector of length n, where n is the number of processes. With each event of process p_i, VT_i ticks by incrementing its own component of its clock, $VT_i[i]$. Clock tick is considered to occur before any event; the timestamp of an event is the clock value after ticking. Each message carries a timestamp which is the sender's clock. The receiver p_i of the timestamped message updates its clock with the componentwise maximum of its clock and the timestamp contained in the message, that is, $VT_i := sup(VT_i, t)$, where t is the timestamp of the message and $sup(C, C') = [max(C[1], C'[1]), ...max(C[n], C'[n])]$. Figure 3 shows an example of events timestamping using vector timestamps (arrows denote message transmission). Assuming that VT_i corresponds to the clock of process p_i timestamping its local state ls_i, a global state is consistent [10] if $\forall i \; \forall j \;\; VT_i[i] \geq VT_j[i]$.

For instance, in Figure 3, the system state in C_1 is a consistent state, while the system state in C_2 is not consistent ($VT_3[2] > VT_2[2]$): the message sent from p_2 to p_3 is captured as received in p_3's state and not yet sent in p_2's state.

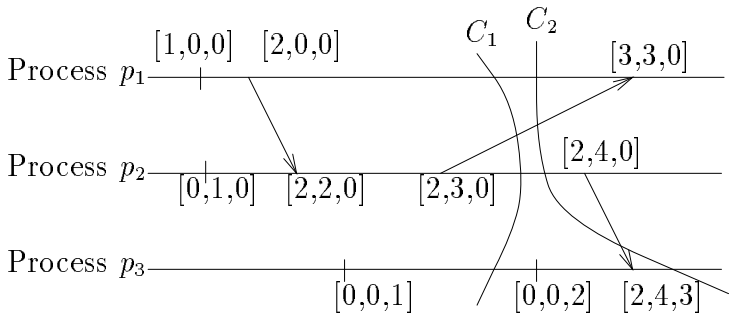

Figure 3: Vector timestamps

3.3.2 Protocol for detecting global garbage

The principle of the protocol used for detecting global garbage is to record a global snapshot of the state of the system relevant to garbage collection. The local garbage collectors communicate data used for obtaining this snapshot though message passing. Two types of messages are used for the detection of global garbage objects: an *Info* message is sent by a local garbage collector to the global garbage collector, and contains the information needed to detect global garbage; a *Delete* message is sent by the global collector to a local garbage collector to notify that some objects are garbage. Events on each node are timestamped by using vector timestamps.

The global garbage collector maintains a global snapshot of the system state by recording a global graph G, which is the union of the subgraphs Gi sent by the local garbage collectors, as well as the timestamps $VT1, ..., VT_N$ of the last information received from the local garbage collectors. G is recorded as a set of edges labelled with the node that sent them. An edge is a tuple $(< id_1, act_1 >< id_2, act_2 >)$, where id_1 and id_2 are object names and act_1 and act_2 are object activity attributes (*i.e. running*, *inactive* or *root*).

An *Info* message sent from node N to the global collector contains N, *Edges, Trans,* and VT_N, where *Edges* is the list of references for the subgraph sent by N; *Trans* is the state of the communication channels, required for having a global system state; finally, VT_N is the timestamp defining the computation time of the *Info* message. Upon receipt of *Info(N,Edges,Trans,*VT_N*)*, the global garbage collector replaces the edges of G that are labelled with N by the edges contained in *Trans* and *Edges*. The timestamp of the old subgraph sent by N is replaced by VT_N. The global garbage collector then checks

whether G represents a consistent state of the system (*i.e.* $\forall i \; \forall j \quad VT_i[i] \geq VT_j[i]$). If G is not consistent, nothing is done (the global garbage collector waits until a consistent state is detected). Otherwise, the global garbage collector processes G using the centralized marking algorithm described in Section 2, where initially only root objects are marked black. On marking termination, white and grey objects are garbage. The names of white and grey objects labelled with node N are gathered in a list l. A *Delete* message containing l is then sent to N.

Upon receipt of a *Delete* message, the local garbage collector running on node N removes from *In_Table* and *Out_table* every object x whose name is contained in the message. Thus, x will be deleted on the next activation of the local garbage collector.

3.3.3 Recording the state of the communication channels

The global garbage collector needs to record the states of the communication channels in order to establish a global state of the system. The only information relevant to garbage collection is the list of references carried by in-transit messages. Hence, as far as recording the state of the communication channels is concerned, this is the only information to be sent from the local garbage collectors to the global garbage collector. Note that a reference on an object x contained in a message may be considered as in-transit in the global snapshot recorded by the global garbage collector although it is received in the real state of the system, without incorrect behavior of the global garbage collector. Indeed, this implies only that G has an extra-edge containing a reference on x which results in a delayed detection of x as garbage. Consequently, each reference carried by a message need not be acknowledged by its receiver immediately upon receipt. Each time a message is sent, references contained in the message are recorded in *Trans*. They are removed from *Trans* (and the corresponding memory is freed) once there is no doubt the message is received. Assuming, FIFO channels, a message m sent from N to M is received when M has acknowledged a message sent by N later than m. The protocol that is used to acknowledge references consists in piggybacking on each message from M to N the acknowledgment of the last message received by M from N.

3.4 Properties of the Protocol

It is important to show that the proposed protocol does not detect an object as being garbage when it is not (*correctness*), and that every garbage object is eventually detected (*liveness*). Correctness of the base marking algorithm proposed by Kafura in [5] being shown, proving correctness of our algorithm boils down to showing the correctness of the global garbage collector. Informally, correctness comes from the stability property of garbage (see Section 2), and from the fact that the global garbage collector operates on a global snapshot [9, 10]. Assuming liveness of local garbage collectors, showing that progress is made needs to ensure the following properties:

(P1) In-transit references are eventually known to be received;
(P2) A consistent state is eventually detected by the global garbage collector.

Due to the technique that is used for keeping track of possibly in-transit references, the first property is ensured if the time interval between two successive exchanges of messages from each node N to each node M is finite. However, if a node N stops sending messages to node M after the receipt of a message m from M, M will never know m was received. In order to satisfy *(P1)*, each node periodically multicasts a message to each node to which it did not communicate since the previous multicast.

Although the local garbage collectors send data to the global garbage collector periodically, ensuring that a consistent state is eventually detected by the global garbage collector is not directly satisfied by the proposed protocol. A practical approach for detecting a global snapshot in a reasonable delay is for the local garbage collectors to send information to the global garbage collector at predetermined physical time intervals. The nodes are in this case loosely synchronized to send information to the global garbage collector. However, although this technique is realistic from a practical point of view (see Section 4 for a confirmation), it still does not ensure that a consistent state is eventually detected. Hence, a *panic mode* of the global garbage collector is defined. In panic mode, all the nodes synchronize with each other for building a global snapshot. The global garbage collector records for each node a *panic threshold* PT. PT is the maximum allowable number of *Info* messages sent by a node to the global garbage collector before a consistent state is obtained. When the number of messages sent to the global garbage collector exceeds

PT without obtaining a consistent state, the global garbage collector enters the panic mode and initiates the computation of a global snapshot. The protocol given in [10] can be used for that purpose. In this way, the system can balance garbage collection costs against the urgency of its need for storage.

3.5 Extensions

In this paragraph, the proposed garbage collector is extended for considering features of real distributed systems. In particular, the protocol is extended for coping with unreliable communication channels and large scale distributed systems.

3.5.1 Supporting unreliable communications

Until now, we have assumed reliable and FIFO communication channels between nodes. In true distributed systems, additional features must be taken into account. A message may be lost, duplicated or arrive our of order. Byzantine failures are ruled out: message contents are not altered during transmission. Delivered messages arrive in finite (but not necessarily bounded) time. When considering that messages may be lost, it is assumed that transmission of sufficiently many messages will eventually cause at least one to be received. Two kinds of messages exist in our system: messages sent by the application and garbage collection messages, the latter being used for the detection of global garbage (*Info* and *Delete* messages). It is assumed that the application knows how to deal with unreliable channels (*e.g.* by sending again lost messages and removing duplicates). Our algorithm tolerates message loss, duplication and non-FIFO ordering independently of the solutions adopted by the run-time support of applications.

Message loss: As the local garbage collectors send information periodically to the global garbage collector, the loss of a message *Info* will only cause a delay in the detection of global garbage. The loss of a message *Delete* notifying node N that object x is to be removed is also tolerated since garbage objects remain garbage: the global garbage collector will still detect x as garbage during its next marking phase. The loss of an application message will not cause the incorrect deletion of objects. The only objects that could be incorrectly identified as garbage are the objects whose references are contained in the lost message. All references contained in a message m

sent from N to M are considered to be in transit until M has acknowledged a message sent by N after m (m is either received or lost). Thus, no object is incorrectly identified as garbage. If m contains the last reference on object x, x will eventually be detected as garbage (as soon as N will detect that m is either received or lost).

Non-FIFO ordering: Our use of timestamps to guard against possibly in-transit references works well if channels are FIFO, *i.e.* if messages are received in the order sent (if at all). With a small extension, our algorithm can cope with out of order messages. Non-FIFO ordering of messages is tolerated if the following acceptance condition is added for receiving application messages: a message m sent from N to M is rejected, *i.e.* considered as being lost, if it arrives after a message sent from N to M after m (all late application messages are considered lost). Late *Info* messages received by the global garbage collector are also eliminated, because they carry out-of-date information. An *Info* message sent by a node N is accepted if the following acceptance condition is verified: $VT < VT_N$, where VT_N is the timestamp of the last information sent by N and VT is the time stamp of the *Info* message. Non-FIFO ordering of *Delete* messages are harmless since garbage objects remain garbage.

Duplicated messages: The duplication of an *Info* message is already taken into account by the acceptance condition of messages on the global garbage collector, given in the previous paragraph. The duplication of a *Delete* message is treated by making the action executed when this message is received idempotent: x's name is remove from *In_Table* and *Out_Table* on node N only if the name still belongs to *In_Table* or *Out_Table*. Since object identifiers are not reused, there can be no confusion of object identifiers. If object names were reused, stronger assumptions, like bounded transmission delay would have to be done.

3.5.2 Supporting large scale systems

Due to the increasing speed of communication networks, it becomes realistic to execute parallel applications on distributed architectures composed of more and more machines. In order to adapt the proposed garbage collector to a large scale distributed system, the bottleneck of the centralized

garbage collection service has to be removed, both for having the garbage collector more available and for avoiding making global garbage collection for the whole system. Due to the structure of the proposed garbage collector, more availability can be obtained by replicating the global garbage collector. A local garbage collector then sends information to a single replica of the global garbage collector (the global garbage collector is seen by its user as a centralized service). Information is propagated to all other replicas in the background. This permits garbage to be collected even if some of the replicas are unavailable. In order to avoid doing garbage collection on a system wide basis, the proposed garbage collector structure can be extended from a two level hierarchy (i.e. global and per-node garbage collection) to a multi-level hierarchy. Following [12], instead of having exactly one global garbage collector for the whole system, a global garbage collector is now associated to a logical memory space; it detects garbage within this memory space and sends information to the upper level global garbage collector, which is associated to the enclosing logical memory space. For instance, for a system composed of two interconnected local area networks, there can be one local garbage collector per node, one global garbage collector per network, and at the top of the hierarchy one global garbage collector for the whole system.

4 Implementation

The proposed distributed garbage collector was implemented in the distributed run-time system of the concurrent object-oriented language Arche [13]. The next two paragraphs focus on the implementation and performance of the garbage collector.

4.1 Implementation of the Garbage Collector

As far as garbage collection is concerned, Arche adheres to the active object model: an object embeds atomic values, references on other objects, and threads of control. The run-time system of Arche is built above the Mach micro-kernel [14] and runs on a 10Mb/s Ethernet network of SUN 3 workstations and PC-compatible machines. It is implemented through a set of Mach multithreaded servers communicating through reliable message passing. In order to reduce the response time of Arche applications, the run-time system implements a dynamic processor allocation strategy. Object states are

accessed though virtual memory, and are swapped to disk when they have not been used for a too long time. Objects are universally named by their address in virtual memory (virtual memory addresses are not reused).

A local garbage collector runs each node and does garbage collection for the objects stored on the node. *Root_Table* notably contains the name of the root of an external naming system, which allows connecting user-defined symbolic names to objects. In order to scan the contents of objects' states, objects' stacks and messages sent between nodes, the Arche compiler generates garbage collection templates. A garbage collection template is a null-terminated array of integers: a positive value X states that X bytes of data have to be skipped before finding the next reference, while a negative value -X indicates that the X following values are references on objects. Both the local and garbage collector are activated periodically at given physical time intervals. The global garbage collector stores the global graph G as lists of edges accessed through a hash table. When the global garbage collector notifies one node that one object is garbage, the object is removed from *In_Table* and *Out_Table*, so that it will be detected as being garbage on the next local garbage collector activation. When an object is to be deleted by the local garbage collector, its internal threads of control are first stopped; the virtual memory region and corresponding swap space are then freed.

4.2 Performances

Costs of the proposed distributed garbage collector is considered in two steps: first, an evaluation of the requirements of the algorithm in terms of number of messages, computation time and memory space is done; second, results of measures on the implemented algorithm are given.

4.2.1 Performance evaluation

From the standpoint of communication requirements, the only foreground messages that have to be sent are those needed for communicating with the global garbage collector. Assuming messages with unbounded size, and assuming that a consistent state is detected, if N is the number of nodes in the system, only $N+1$ messages are required to detect an object x as being garbage (N *Info* messages and one *Delete* message). In actual systems, where messages have bounded size, more than one message may be required for sending information to the global garbage collector.

Memory space is required for local marking, global marking, timestamping, and storing the global graph. Two bits per object are required for local and global marking, as three colors have to be coded. Memory requirements for timestamps are of N integers per node, where N is the number of nodes. Most of the memory requirements of the proposed garbage collection technique comes from the memory space required for storing the global graph G. It strongly depends on the degree of locality exhibited by the running applications and on the selected representation for G (e.g., matrix, list); real figures with a representation of G as a list are given in the following paragraph.

Concerning computation time, mainly three factors have to be considered: local marking, collecting information for the global collector, and global marking. The algorithm used for performing marking [5] has a worst case time complexity of $O(p^2)$, where p is the number of objects in the graph that is marked.

4.2.2 Performance measures

Performance of the local and global garbage collectors is measured on a numerical application which is cyclic. During each cycle, two matrix objects both containing one hundred integers chosen at random are created; a matrix object is created and is filled with the product of the two matrixes; its contents is then printed on the screen. At the end of each cycle, the three matrix objects are garbage. Initially, fifty objects are registered within the external naming system.

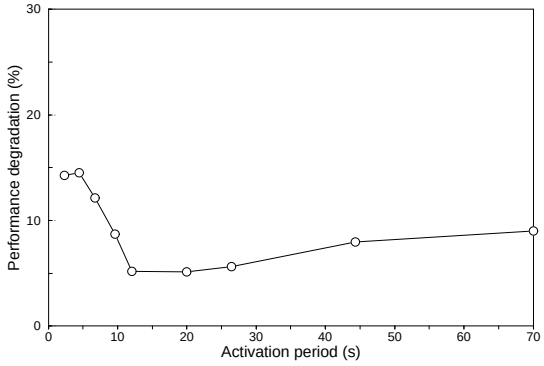

Figure 4: Performance degradation for local garbage collection

Figure 4 shows the percentage of processing power spent in local garbage collection: an average of 8% of the total execution time is spent in local gar-

bage collection for the selected application. Furthermore, the figure shows that a high performance degradation (up to 15%) results from a very short activation period of the local garbage collector; all the processing power is then spent in scanning the contents of the external naming system. Finally, the figure shows that the performance degradation due to local garbage collection increases when the interval between two successive activations of the local garbage collector becomes long; this comes from the fact that objects have been swapped out to disk, so marking involves loading these objects into main memory.

Performance measures of the global garbage collector are given for a distributed system composed of four nodes, running the matrix product application. As the performance of the global garbage collector depends on the locality of references exhibited by applications, let us model it by the *sharing rate* of objects, which is defined as follows: a sharing rate of *0* means that all objects are created on the same node, which executes the whole computation; a sharing rate of *1* means that objects are distributed randomly on the nodes; finally, a sharing rate of n/p means that a percentage of p-n/p object is created on one node, while the remaining percentage of n/p is exported to other nodes. The processor allocation facility of the run-time environment permits to make the objects' sharing rates vary.

Sending information to the global garbage collector periodically at fixed physical time intervals turns out to be realistic: when the objects sharing rate is inferior to *0.75*, measures showed that only one message per node is required in order to obtain a global snapshot. Furthermore, even when the sharing rate of objects is *1*, only *2%* of messages sent to the global garbage collector by a local garbage collector do not lead to the detection of a consistent state.

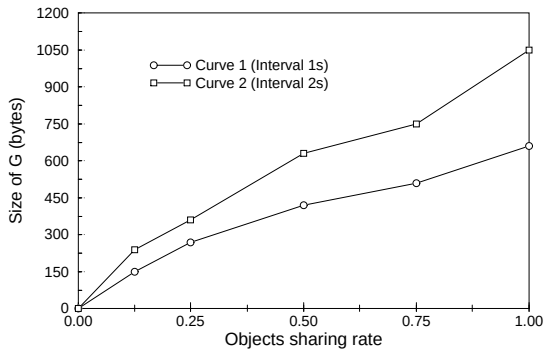

Figure 5: Memory requirements for the global garbage collector

Figure 5 shows the volume of information of the global graph G, where G is stored as a list of edges accessed through a hash table. Curve 1 (resp. curve 2) corresponds to a configuration where data is sent to the global garbage collector every second (resp. 2 seconds). The figure shows that the memory requirements for G highly depends on the locality of references of the application. Moreover, it shows that the larger is the interval between two successive global garbage collections, the greater is the memory space required for storing G.

5 Related Work

Numerous garbage collectors have been proposed since the birth of the first programming languages with dynamic memory allocation. Most of them apply only to non-distributed passive objects (see [1] for a survey of existing propositions). Fewer collectors have been developed for distributed systems (see [15, 16] for examples), but the vast majority of them focus on determining object reachability which, as seen before, is too weak a criterion for detecting garbage in a system of active objects. In particular, as mentioned earlier, algorithms based on reference counting (like for instance [17], which supports unreliable communications), are not suited because they only consider object reachability. Few algorithms detect garbage in distributed systems of active objects [18, 19, 20]; they are compared with our proposition below.

A distributed garbage collector similar to the one proposed in this paper is the garbage collector described in [19] for a distributed system of actors. Like our garbage collector, this technique relies on independent local garbage collectors and a global garbage collector, both using the marking algorithm described in [5]. However, unlike our proposition, the garbage collector of [19] enforces global synchronization to detect global garbage, and assumes reliable communications. Another related garbage collector was developed for the EMERALD object-based programming system [18]. While EMERALD provides active objects, the garbage collector designed for this system is based exclusively on object reachability (all running objects are designated as being root objects). Thus, an object is deleted once and only once it stops executing; unlike our proposition, no running object is detected as being garbage. Like our proposition, the algorithm proposed in [20] detects garbage in a distributed system of active objects by building a global snapshot of

the system state. The difference between the two propositions is the way the global snapshot is obtained: in [20], a two dimensional grid architecture is considered and properties concerning message routing on the grid topology are used for detecting a consistent system state; in our proposition, timestamping of events is used for this purpose, and no assumption is made on the underlying architecture.

Our proposition has some similarities with garbage collection algorithms designed for distributed systems of passive entities. Like the algorithm described in [21], our global garbage collector is based on (possibly out-of-date) information on inter-node references that permits the elimination of global synchronization when detecting global garbage. Unlike [21], node crashes and crash recovery are not considered. Only node unavailability is supported. However, in contrast to [21], we detect garbage in a system of active objects and require neither synchronized clocks nor bounded message transmission delay. We borrow to [12] the hierarchic structure of the garbage collector for its extension to a large scale distributed system. The distributed fault-tolerant garbage collector described in [22] is based on reference counting and works in a system with unreliable nodes and communication channels. In contrast to our proposition, two separate mechanisms are used to detect unneeded computations (computations that become unneeded due to a node crash) and to detect unneeded data structures.

6 Concluding remarks

In this paper was described a distributed garbage collector suited for systems of active objects. Autonomous local garbage collectors detect and reclaim local garbage without communicating with other nodes. A global garbage collector detects remaining garbage by building a global snapshot of the system state, built by merging data sent asynchronously by the local garbage collectors. The key advantage of the proposition is that local garbage collectors need not synchronize with each other when sending information to the global garbage collector. However, this weak synchronization between the local garbage collectors is done at the expense of the time taken for detecting global garbage. As the garbage collector is based on marking, garbage objects belonging to cycles, even if they span multiple nodes, are detected. Moreover, the proposed algorithm can be easily extended to cope with unreliable commu-

nication channels and large scale distributed systems. The proposed garbage collection algorithm was implemented within the distributed run-time support of a concurrent object-oriented language. Performance measures show that the approach undertaken for the global garbage collector is realistic if the applications exhibit good locality of references.

As far as time complexity is concerned, the proposed algorithm, as well as other garbage collection algorithms dealing with active objects, has higher worst case time complexity than algorithms dealing with passive entities (see [5] and [1] for an evaluation of the time complexities of both classes of algorithms). This comes from the fact that for detecting whether an active object is garbage or not, both data and activity of objects have to be considered. If an algorithm suited for passive entities was used, less time overhead would result from garbage collection, but no execution would be stopped, thus consuming processing power. Moreover, more work would be left to the programmer, who would be responsible for stopping executions that he/she decides to be unneeded.

Acknowledgments

Thanks to Valérie Issarny and Michel Banâtre for their careful readings of earlier versions of this document.

References

[1] J. Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[2] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall International, 1988.

[3] P. America. Pool-T : A parallel object-oriented language. In *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press Series in Computer Systems, 1987.

[4] B. N. Bershad, E. D. Lazowska, and H. M. Levy. Presto: A system for object-oriented parallel programming. *Software Practice and Experience*, 18(8):713–732, 1988.

[5] D. Kafura, D. M. Washabaugh, and J. Nelson. Garbage collection of actors. In *Proc. of the 1990 ECOOP/OOPSLA Conference*, pages 126–133, 1990.

[6] Roger S. Chin and Samuel S. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[7] G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[8] I. Puaut. Distributed garbage collection of active objects with no global synchronisation. In *1992 International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 148–164, Saint Malo, France, September 1992. Springer Verlag.

[9] K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[10] F. Mattern. Virtual time and global states in distributed systems. In *Proc. Int. Conf. on Parallel and Distributed Algorithms*, pages 215–226. North-Holland Publishing, 1988.

[11] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. 11th Australian Comp. Conf.*, February 1988.

[12] B. Lang, C. Queinnec, and J. Piquer. Garbage collecting the world. In *Proc. of 19th Annual Symposium on Principles of Programming Languages*, pages 39–50, Albuquerque, New Mexico, January 1992.

[13] M. Benveniste and V. Issarny. Concurrent Programming Notations in the Object-Oriented Language Arche. Research report 1822, inria, Rennes, France, 1992.

[14] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proc. of Usenix 1986 Summer Conference*, pages 93–112, July 1986.

[15] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proc. of PARLE Conference*, volume 259 of *Lecture Notes in Computer Science*, pages 432–443, Eindhoven, 1987. Springer Verlag.

[16] L. Augusteijn. Garbage collection in a distributed environment. In *Proc. of PARLE Conference*, volume 259 of *Lecture Notes in Computer Science*, pages 75–93, Eindhoven, 1987. Springer Verlag.

[17] M. Shapiro, P. Dickman, and D. Plainfossé. Robust, distributed references and acyclic garbage collection. In *Symposium on Principles of Distributed Computing*, pages 135–146, Vancouver (Canada), August 1992.

[18] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[19] D. M. Washabaugh and D. Kafura. Distributed garbage collection of active objects. In *Proc. of 11th International Conference on Distributed Computing Systems*, pages 369–376, May 1991.

[20] N. Venkatasubramanian and G. Agha aand C. Talcott. Scalable distributed garbage collection for systems of active objects. In *1992 International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 134–147, Saint Malo, France, September 1992. Springer Verlag.

[21] B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proc. of 5th International Symposium on Principles of Distributed Computing*, pages 29–39, Alberta, Canada, August 1986.

[22] L. Mancini and S. K. Shrivastava. Fault-tolerant reference counting for garbage collection in distributed systems. *The Computer Journal*, 34(6):503–513, May 1991.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399