# A garbage detection protocol for a realistic distributed object-support system

# Un protocole de détection des miettes pour un système d'objets réparti réaliste

**Rapport de Recherche INRIA 1320**

Marc Shapiro
INRIA, projet SOR

David Plainfossé
INRIA, projet SOR et UPMC-RXF

Olivier Gruber
INRIA, projet Sabre et UPMC-MASI

November 1990

**Abstract**

We present a new protocol for the distributed detection of garbage, suitable for a low-level distributed object-support system. It is based on realistic assumptions: messages may be lost or duplicated; sites may crash; an object may migrate or be deleted. The protocol uses only information local to each site, or exchanged between pairs of sites; no global mechanism is necessary. It is parallel and should scale to extremely large systems. It takes into account the object-finding protocol. Its interface is designed for maximum independence from other components.

**Résumé**

Nous présentons un nouveau protocole pour la détection répartie des miettes. Il se prête à une mise en œuvre au niveau système. Il se base sur des hypothèses réalistes : un message peut se perdre ou être dupliqué ; un site peut crasher ; un objet peut migrer ou être effacé. Ce protocole n'utilise que des informations locales à un site, ou échangées entre deux sites ; aucun mécanisme global n'est nécessaire. Il est parallèle et se prête à des systèmes de très grande échelle. Il tire parti des autres composantes du systèmes, comme le protcole de recherche des objets. Son interface permet une grande indépendance entre composantes.

# Contents


**1 Introduction** **1**

**2 Garbage detection** **2**

2.1 Centralized algorithms . . . . . 3

2.2 Distributed garbage collection . . . . . 4

2.2.1 Distributed reference counting . . . . . 4

2.2.2 Distributed tracing . . . . . 5

2.2.3 Conclusion . . . . . 6

**3 System model** **6**

3.1 Objects . . . . . 6

3.2 Spaces . . . . . 6

3.3 Communication . . . . . 8

3.4 Finding an object . . . . . 8

3.5 Notations . . . . . 9

3.6 Global GD protocol . . . . . 10

**4 Failure-free parallel system** **10**

4.1 Initial state . . . . . 10

4.2 Sending a reference . . . . . 10

4.3 Migrating an object . . . . . 11

4.4 Local garbage collection . . . . . 12

4.5 Global garbage detection protocol . . . . . 13

4.6 Elimination of extra indirections ("path compression") . . . . . 13

4.7 Interaction between garbage detectors . . . . . 15

4.8 Inter-space cycles of garbage . . . . . 15

4.9 Alternatives for detection of inter-space garbage cycles . . . . . 16

4.10 Conclusion . . . . . 17

**5 Distributed system with failures** **17**

5.1 Message transmission in non-zero time . . . . . 17

5.1.1 An example . . . . . 17

5.1.2 Sending and receiving messages . . . . . 18

5.1.3 GD protocol accounting for messages in transit . . . . . 20
5.2 Unreliable message transmission . . . . . 20
5.2.1 Lossy channels . . . . . 21
5.2.2 Non-FIFO ordering . . . . . 22
5.3 Deletion and termination . . . . . 23
5.3.1 Normal and abnormal deletion . . . . . 23
5.3.2 Disconnection and termination of spaces . . . . . 23

**6 Recapitulation** **26**

**7 Conclusion** **29**

# 1 Introduction

Recent development of the object-oriented technology has sparked interest in low-level support systems for user-defined objects. A number of operating systems [5, 16, 12] and database systems [11, 13, 17] offer such support. Instead of the untyped abstractions of previous-generation systems (process/file/message for operating systems, relations for databases), object-support systems provide for user-defined objects with strong type and strong semantics.

One important aspect is that an object may contain *references* to other objects. A program's activity creates objects and modifies the references between them; an object for which no reference remains has become inaccessible *garbage* and could be de-allocated. In many existing systems and languages, the decision to dispose of an object as it becomes unneeded is a manual, error-prone process. In contrast, in some language environments such as Lisp or Smalltalk, a built-in *garbage collector* automatically detects unaccessible objects and disposes of them.

Automatic garbage collection (GC) is a valuable service, as it frees programmer resources and is safer than manual collection. Unfortunately GC algorithms have been in the past perceived as too complex and language-specific for inclusion in general-purpose systems. This perception may be changing [4, 19]

Many published distributed GC algorithms are based on very strong assumptions and/or expensive, non-scalable mechanisms, making them unsuitable for a low-level object-support system. Typically, messages are assumed to be delivered reliably and failures (such as a site crash) are not considered. Some algorithms suppose message transmission to be instantaneous. Some need a form of global rendez-vous. The assumption of atomicity pervades much of the distributed-GC literature: operations on references and on objects are assumed never to fail. This assumption is well approximated (in the general case) only in the context of atomic transactions. Whereas some of these assumptions may be reasonable for a particular hardware or a particular application area, they are too strong for a low-level, general-purpose object-support system.

In contrast, we propose a protocol for distributed garbage detection based on reasonable, weak assumptions. Messages may be lost, delivered out of order, or duplicated. Nodes may crash. Objects may migrate or be deleted. The protocol is fully parallel, and bases itself only on local and pair-of-sites information. Since no global mechanism is necessary, it should scale to any number of nodes. No assumption is made w.r.t. the semantics of objects; for instance any object may either persist or disappear after a crash. Our

protocol is simple.

Here is the basic idea of our protocol. Each disjoint *space* maintains a list of potential incoming and outgoing references, called respectively the Object Directory Table (ODT) and External Reference Table (ERT). Both the ODT and the ERT are conservative estimates. Local garbage collection proceeds from the union of the local root and the ODT and remove entries in the ERT, which in turn allows previously-pointed-to ODTs to be collected.

Our limiting assumptions are that crashes are fail-stop, and that messages are either lost or delivered unmodified in finite time. We consider both reliable and unreliable communication media. We only consider passive objects.

The interface between the global collector and other components (i.e. the mutator and the object finder) is limited to the ODT and ERT. Updates to an ODT or an ERT can occur in parallel with other activity. We make minimal assumptions about the mutator's behaviour. This ensures maximum independence of the collector, and makes it suited to implementation at the operating system level.

The paper proceeds as follows. First, section 2 discusses the principles of garbage detection and collection, and previous work on distributed GC. Then section 3 describes our model and notations.

The following sections discuss the algorithm itself. Section 4 presents a simplified version, in an idealized environment with no faults and simplified functionality. In section 5, we introduce the full functionality, and examine the consequences of failures; the algorithm is adapted to each relaxation of assumptions, without modifying its global structure. Section 6 recapitulates the whole GD protocol. We conclude in section 7.

The reader who wishes to refer to the algorithm, without justification, may jump directly to section 6.

# 2 Garbage detection

Garbage collection in a low-level object-support system poses three distinct problems:

- distinguishing references from other data in objects,
- given these references, detecting garbage objects,
- disposing of garbage objects, possibly taking into account their semantics.

In this article, we ignore the former and latter problem which are language-dependent, and focus only on an OS-level realization of *garbage detection* (GD).

The purpose of GD is to distinguish possibly-useful objects, called *live*, from those known to be unusable, called *garbage*. Any object referred from a live object is itself live. By definition, any object reachable from a predetermined set (called *the root*) of live objects, via the transitive closure of the "refers to" relation, is live. The complement of this set is garbage. Once an object becomes garbage, it remains so forever: it is a *stable* property.

Following Dijkstra [7] we distinguish two rôles: the *mutator* and the *collector*. The mutator is the user-defined program, which allocates objects and modifies references, causing some objects to become garbage. The collector is the system component, which detects garbage and deletes it. Dividing the work of the collector into two parts, garbage detection and garbage disposal, we will speak of the *detector* rôle.

The existence of the collector is transparent to the mutator. Running the collector impacts the performance and economics of the mutator, by freeing space, but not its semantics. The collector may not delete objects which the mutator will access in the future, nor may it detectably modify them. Whenever the mutator and the detector operate in parallel, some degree of cooperation is neccessary to ensure their respective correctness: at the very least, access to a reference must be an indivisible operation.

## 2.1 Centralized algorithms

There are two families of well-known GD algorithms: reference counting and tracing.

In reference counting, each object carries a count of the number of references pointing to it. Every operation on a reference (such as assigning a pointer variable) must indivisibly increment and/or decrement the associated count(s). When some count reaches zero, its object is garbage.

In tracing, each execution of the detector performs a walk of the "refers to" graph, starting from the root; any object not visited, when the walk terminates, is garbage. To distinguish visited objects, different techniques are used. "Mark-and-sweep" uses mark bits; a second, sweep, pass deletes unmarked objects. "Copy-collect" moves visited objects to a distinguished zone of memory called "to-space"; at the end of the walk, objects remaining in the original "from-space" are garbage and can be deleted all at once. "Generation scavengers" use multiple zones, sorting objects by age.

Boehm and Weiser [4, 2] remark that any GD algorithm, to a degree,

considers too many objects as live: some reachable objects will in fact never be used. Conversely, a GD algorithm may consider some garbage objects to be live (for instance, by tracing more objects than strictly reachable), without affecting its correctness. Such *conservative* algorithms can be used even when the minimal root and/or set of references is not known.

We will not discuss the pros and cons of centralized GD algorithms, which are extensively covered in the literature.

## 2.2 Distributed garbage collection

We will now look at some distributed extensions of the centralized algorithms. We conclude that none are appropriate for our purposes; but recognizing their drawbacks suggests a promising direction of study.

### 2.2.1 Distributed reference counting

At first glance, reference counting seems to be the most amenable to distribution. In fact a number of variations have been proposed; see for instance references [2, 18]. However they have strong drawbacks. For instance, messages should be delivered to their destination in the order of their causal dependence [14]. E.g. if site A, holding a reference to object b on site B, sends the reference to site C and immediately deletes its own reference to b, the decrement message from A may arrive to B before the increment message from C, incorrectly causing object b to be discarded.

Some variations such as Weighted References [2, 10] do not have this particular problem, but do remain sensitive to lost messages and crashed sites. Published reference-count algorithms assume that operations on references and objects never fail.

Vestal [18] proposes two different garbage collection algorithms, one based on reference counting, the other based on tracing. The first one, uses a distributed fault tolerant reference counter. I.e., each object maintains a conservative list of sites referencing it, and each site of this list keeps the count of references it has for that object. Atomic update of the list is required when a site first references an object. The cycle-detection algorithm is seeded with some object suspected of being in dead cycle; it essentially consists of a trial deletion of the seed, and checking if this brings all the counts in the cycle to zero.

The basic problem with reference counting is that a strong invariant (equality of an object's count with the number of references to that object) must be maintained at all times. To do so reliably is intrinsically hard.

### 2.2.2 Distributed tracing

Tracing is potentially more fault-tolerant, because each new execution of a tracing detector starts from the root anew, masking out the consequences of previous failures.

Couvert et al. [6] present a distributed mark-and-sweep collector. All sites start a local mark phase; then perform a global rendez-vous at the end of all local mark phases, to exchange information about the global reachability graph; then proceed to parallel local sweep phases. A global rendez-vous is inherently costly and non-scalable.

Vestal's [18] tracing algorithm is inspired from Dijkstra's [7] parallel collector. The object space is split into logical areas in which parallel collection may occur. The space overhead is proportional to the number of objects and to the number of areas, since each object maintains an array of four colours for each existing areas in the system. This algorithm does not take advantage of locality: each collector performs a global transitive closure starting at the root of one area, hence crossing area boudaries. Areas are a logical grouping of objects, and there is no control over site boundary crossing.

Liskov and Ladin [15], describe a fault tolerant distributed garbage detector based on their highly available centralized service. This service is logically centralized but physically replicated, hence achieving high availability and fault-tolerance. A client dialogs with only one replica; replicas stay up-to-date by exchanging background "gossip" messages. The failure asumptions are realistic: nodes may crash (in a fail-stop manner) and recover, messages may be lost or delivered out of order. All objects and tables are assumed backed up in stable storage. Clocks are synchronised, and message delivery delay is bounded. These requirements are needed for the centralized service to build a consistent view of the distributed system.

The distributed garbage detector relies on a local tracing garbage collector, extended with the ability to identify the path between some incoming reference and an outgoing one. Each local collector informs the centralized service about its references to remote objects, about the references it has sent, and about the paths. The root used for tracing is the union of its local root with the set of local public objects. Local collectors query the centralized service about the real accessibility of their public objects to better estimate their root.

Dead inter-site cycles are detected by the centralized service. Based on the paths transmitted, the centralized service builds the graph of inter-site references, and detects dead cycles with a standard GC algorithm.

#### 2.2.3 Conclusion

Reference counting is fragile because a strong equality invariant must be maintained at all times. Tracing is potentially more fault-tolerant, because each execution of the detector forgets the consequences of past faults. Many published distributed tracing algorithms are too complex. One reason may be that they try to track accurately at all times the minimal set of reachable objects.

These observations suggest an interesting direction of research: investigate a distributed combination of tracing (for fault-tolerance), and conservative techniques (for simplicity). Indeed, we have done such a study, resulting in the protocol we will present in the next sections.

# 3 System model

This section gives some definitions and our notation, and lists our assumptions. These are quite reasonable and minimal, and do not restrict the generality of our protocol.

## 3.1 Objects

We assume a large-scale distributed system, supporting a large number of objects. Objects are passive. An object may contain any number of references to other objects. The implementation of a reference is not considered here; we will simply assume that a reference names its target in a way meaningful to the application.

References to deleted objects are allowed. Deletion is stable; once deleted an object stays deleted forever. A deleted object contains no data and no references.

Deletion can be voluntary or the result of a failure. A well-behaved application deletes an object by setting it to the distinguished deleted state. An object can also be deleted by simply dropping it from memory (abnormal deletion); in this case a protocol, explained later (sections 3.4 and 5.3.1), recovers back to the normal case.

## 3.2 Spaces

The universe of objects is subdivided into disjoint *spaces*. At any time, an exisiting object is either located in some space or in transit (migrating) between spaces.

The exact nature of spaces may vary, since different choices allow different implementations. Since there is no global mechanism (as we will see), a space can in turn be divided into sub-spaces. A particularly appealing architecture is the following. On a workstation, each process is a space; our GD protocol executes between processes. On a storage server, each "volume" (a set of logically related persistent objects, e.g. the set of files owned by one user) is a space; unreachable files are collected based on our GD protocol. The next level of the hierarchy considers each machine in a local network to be a space, abstracting away references between processes and volumes within each machine. This structure takes advantage of the locality property.

Each space has its own local root. Each space performs a standard local tracing garbage detection, independently of the activity of remote or global detectors. The root for local collection is the union of the local root and the ODT (see below). Any correct tracing GC algorithm is suitable locally.

The global root is conceptually formed of the union of all local roots, but this global root is in fact never used, only local information being necessary.

We distinguish between a *local* reference (to an object known to be in the same space) and a *remote* one (to an object thought to be in another space). A remote reference is represented as a local reference to a *stub* object, which in turn contains necessary remote information (i.e. the last known location of the referent). Stubs are collected in a structure called the *External Reference Table* or ERT. The ERT is maintained by the local collector and readable by the global detector.

Conversely, local objects potentially accessible from other spaces are listed in a *Object Directory Table* (ODT). The ODT of a space is maintained by a loose cooperation between the global detector (from information contained in remote ERTs) and the local mutator. A mutator is not allowed to read from ODT, only to add entries to it[1].

Spaces may terminate. Normal termination causes all its contained objects to be deleted normally, and all indirections crossing it to be eliminated. A space may also terminate abnormally, e.g. by the occurence of a computational error such as divide by zero, or of a user signal, or of a crash. In this case, a protocol described in section 5.3.2 recovers to normal termination. Crashes are asssumed fail-stop, i.e. a crashed space ceases to compute and to send messages, but otherwise takes no action.

Each space will additionally carry time information; we defer its description to section 5.

[1] This is to prevent the mutator from gaining a reference to an otherwise garbage object.

## 3.3 Communication

Communication between mutators in different spaces occurs via messages. A message may contain both references and objects; the objects it contains migrate from source to destination space.

In addition to mutator messages (which we will call simply messages) our protocol adds a few *control* messages, which are not visible to the mutator.

Two spaces A and B are connected by a single channel. Communication is said to be FIFO if for all A and B, the channel A-B delivers messages in the order they are sent.

Delivered messages arrive in finite (not necessarily bounded) time. When considering that messages can be lost, transmission of sufficiently many messages will eventually cause at least one to be received.

The garbage detection algorithm we present tolerates message loss, duplication and out-of-order delivery. A given application (mutator) may be capable of tolerating such failures; if so, the GD algorithm of section 5 will tolerate them too. A different mutator, that does not accept such failures, will run in an environment which avoids them (e.g. it will use lossless FIFO channels); in which case we propose a simplified version of the GD algorithm, adapted to the restricted environment (section 4).

## 3.4 Finding an object

Independently of the garbage detector, an *object-finding* component (e.g. similar to Fowler's [9]) is available. Given a reference, the finder locates the referred-to object. Object deletion and migration, and space crashes, are handled transparently by the object finder. Operation of the finder is fairly independent from garbage detection, but both benefit from each other's actions.

Given a reference $@x$ uttered in space $A$, the finder's algorithm is the following (ignoring disconnected and crashed spaces, which will be considered in section 5.3.2):

- If the object $x$ is found in $A$ in the deleted state, signal the deleted exception.
- If the object $x$ is found in $A$, return.
- If there is a stub $x_A$ pointing to $B$ (see below), and $B$ has an Object Directory Table entry (see below) for $x$, go to space $B$ and recurse. If $x$ is not found it has been deleted; re-create it in the deleted state, and signal deleted.

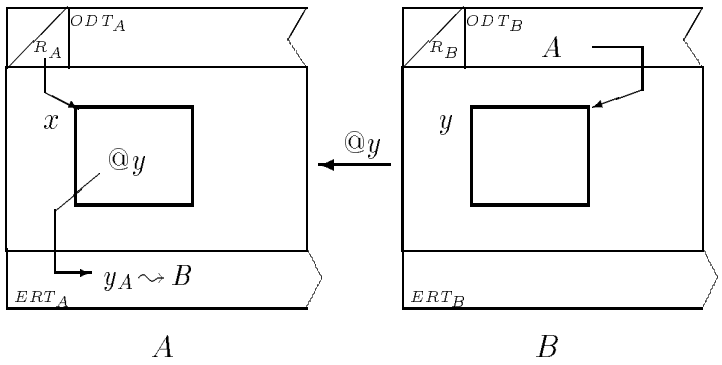


Figure 1: Notation

Attempting to find an object possibly located in a disconnected space may be delayed until that space either reconnects or terminates.

Object finding is accelerated by using location information maintained in the ERTs. Finding an existing object, based on a reference, succeeds even if the available information is stale, because if an object has migrated, a stub containing forwarding information remains behind. In turn, up-to-date location information from the finder is used to update the information in ERTs, making operation of the detectors more efficient.

## 3.5 Notations

The following notations (summed up in Figure 1) will be useful for the rest of the paper. We note spaces with capital letters $A$, $B$, $C$, etc.; objects with lower-case letters $x$, $y$, $z$; and references to such objects as $@x$, $@y$, $@z$. The ODT and ERT of space $A$ are noted $ODT_A$ and $ERT_A$. The local root of $A$ is $R_A$. The stub object, on space $A$, for remote object $y$, thought to be in space $B$, is noted $y_A$, and contains the location information $\rightsquigarrow B$. A $@y$ on space $A$ will in fact point to $y_A$.

On space $B$, an $ODT_B$ entry, at some index $i$, will contain the information that $y$ is potentially referred from space $A$, which we note:

$$ODT_B[i] = (A@y)$$

In Figure 1, object $y$ is not reachable from $B$'s local root $R_B$, but in fact is live, being reachable (via $x$) from $R_A$.

In the figures, an ERT is drawn with a tab pointing outwards and ODTs with a tab pointing inwards.

A message from $A$ to $B$ is noted

$$A \rightarrow B : \mathsf{type}\{contents\}$$

where type is either mutator or a specific control-message type.

### 3.6 Global GD protocol

Briefly stated, the main idea of our protocol is to maintain, in each space's ODT, a conservative estimate of its set of objects accessible from other spaces. The ODT is managed by a conservative protocol involving the local mutator and information extracted from remote ERTs.

Since local GC starts from the union of the local root with the (conservatively estimated) ODT, all non-reachable local objects are true garbage. Each local GC cleans the ERT of useless stubs. In turn, ERTs are used to clean the ODTs, yielding successively better estimates. However, an ODT may possibly never become minimal, because of simultaneous mutator activity. A separate protocol deals with inter-space cycles of garbage (dead cycles).

## 4 Failure-free parallel system

We will now describe the protocol and its application in detail.

We first propose our algorithm under strong assumptions. In the current section, we suppose no failures occur: spaces do not crash, messages are delivered reliably and instantaneously, once and only once, and in FIFO order. This failure-free case is applicable to GD between spaces executing on a single (monoprocessor or multiprocessor) machine.

Later (in section 5) we will relax the assumptions, covering real distributed systems.

### 4.1 Initial state

Let us start the description for some instant where $ODT_A$ and $ODT_B$ contain an exact (i.e. minimal) description of their incoming references. (One such instant is when a space is initially created, and its ODT is empty.)

Let us run $A$'s local GC; it will trace all local live accessible objects from the local root $R_A$ and exact $ODT_A$. Garbage stubs will be deleted. At the end of the local GC, $ERT_A$ contains an exact image of the outgoing references.

### 4.2 Sending a reference

Whenever $A$'s mutator sends a message

$$A \rightarrow B : \mathsf{mutator}\{\ldots, @z, \ldots\}$$

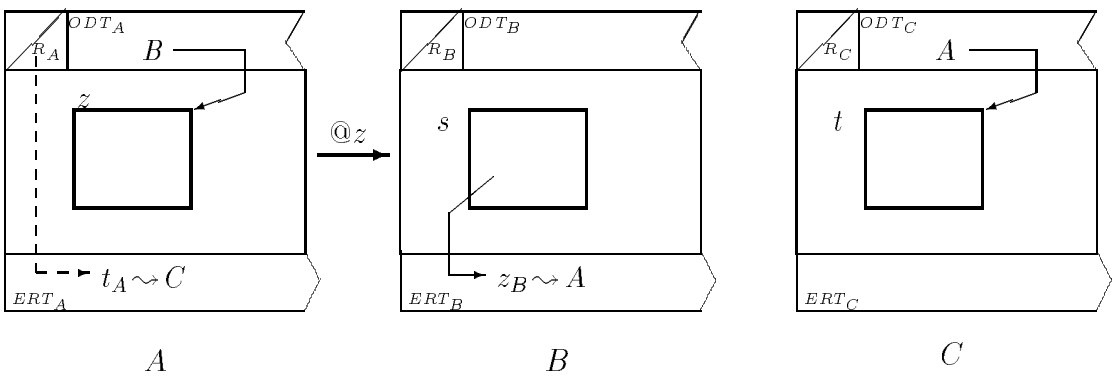


Figure 2: Sending and receiving references

containing a reference $@z$ to $B$ (see Figure 2), we consider that a *potential reference* is created from $B$ to $A$. *Before* the message is transmitted, an ODT entry

$$ODT_A[j] = (B@z)$$

is created (or located if it already exists). If two different spaces possibly refer to a single object of $A$, each will be assigned its own ODT entry.

Note that a single ODT entry is created, whether or not $z$ is local to $A$, whatever number of references are thought to point from $B$ to $A$, and without knowing if the message will succeed.

Now suppose $A$ receives a message from $C$, containing a reference $@t$. If this message reaches $A$'s mutator[2], then a reference will exist from $A$ to $C$. To account for this potential reference, a stub $t_A$ is created, in $ERT_A$, *before* delivering the message to the mutator:

$$t_A \rightsquigarrow C$$

If $t_A$ already exists, a new one is not created (but its location information may be updated if more recent; see [9]). No matter how many references $@t$ exist in space $A$, a single stub $t_A$ exists.

## 4.3 Migrating an object

Let us now consider some object $x$ which migrates from space $A$ to $B$ (see Figure 3) in a message:

$$A \rightarrow B : \mathsf{mutator}\{\ldots, x, \ldots\}$$

[2]We have assumed in this section that the communication medium does not lose messages, but there remains the possibility that the mutator will reject a message, or ignore some information it contains.

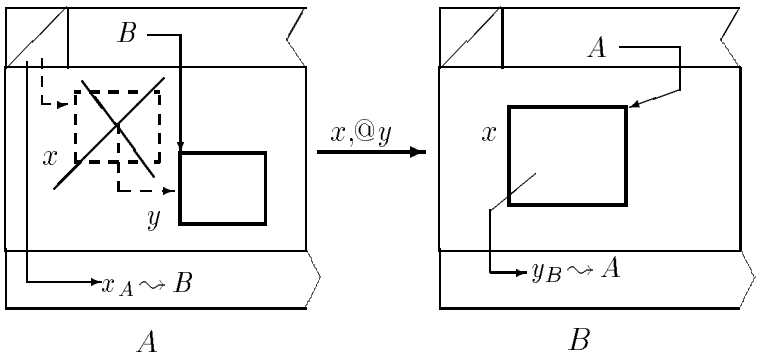


Figure 3: Migration of an object carrying a reference

There may exist references toward $x$ (in space $A$, or from any other space into $A$). Therefore, we consider a potential reference is created from $A$ to $x$ in space $B$. *Before* transmitting the message containing $x$ from $A$, install a stub $x_A \rightsquigarrow B$.

When the message is received by $B$, a potential reference exists from $A$ to $B$; create an entry containing $(A@x)$ in $ODT_B$, *before* delivering the message to the mutator.

That object $x$ may itself contain a reference to an other object $y$. Then, in addition to the migration protocol above, we execute the normal procedure for transmitting $@y$ from $A$ to $B$ (see section 4.2). If any indirections form, they will be eliminated as shown in section 4.6.

## 4.4 Local garbage collection

Starting from a minimal ODT, the mutator's actions can only add entries into the ODT, which therefore either remains minimal or becomes a superset of its minimal value. Therefore, local garbage detection is correct, i.e. either exact or conservative.

For instance, in Figure 2, let us now run $A$'s local garbage collector. Object $z$ is not reachable from the local root $R_A$, but is reachable from $ODT_A$. It is not known if it is globally reachable (in fact, it is not), but conservatively it will be considered live. $t_A$ is reachable by the dashed arrow. If the dashed arrow is removed, then $t_A$ and $t$ are garbage; $t_A$ is removed by $A$'s local GC, whereas $t$ is removed by the protocol in the next section.

## 4.5 Global garbage detection protocol

If only local garbage detection exists, then the ODTs will grow without limit, possibly causing local GC to become inoperative. A global garbage detection protocol is necessary, to remove useless entries in ODTs.

Our global garbage detection protocol does not consist of a global execution of some single detector; it results instead from a loose cooperation of pairs of spaces.

Suppose we remove the dashed arrow in Figure 2, from $R_A$ to $t_A$. Then the next execution of $A$'s local detector will recognize $t_A$ to be garbage. $t_A$ will be discarded, with the side-effect of sending

$$A \rightarrow C : \mathsf{removal}\{@t\}$$

to indicate that $@t$ is no more used by $A$[3]. Upon receiving the removal message, $C$ will delete the corresponding entry in $ODT_C$. At the next execution of $C$'s local GC, $t$ will be recognized as garbage.

To sum up, global GD occurs by pairwise cooperation between spaces. Local GC conservatively updates the ERT; each such ERT is in turn used to conservatively create new versions of the ODTs. Local GC contributes to clean up the local ERT, and hence remote ODTs.

This discussion shows that the GD algorithm sketched above is indeed correct (a conservative estimate of live objects is maintained at all times) and does eventually find some garbage. However, as we will see shortly (in section 4.8), it does not detect inter-space cycles of garbage.

## 4.6 Elimination of extra indirections ("path compression")

An inter-space indirection occurs, for instance if space A of Figure 2 sends a message containing $@t$, to space B. Then we have the situation depicted in Figure 4 (after garbage objects $s$ and $z$ of Figure 2 have been collected). As noted in section 4.2, the action of sending $@t$ in a message from $A$ to $B$ adds an entry $ODT_A[l] = (B@t)$. A stub $t_B \rightsquigarrow A$ is created on reception of the message. Even though $t$ is not located in $A$, $B$'s stub $t_B$ points to $A$ instead of directly to $C$.

Suppose now that $B$'s mutator attempts to use $@t$. The object finder will follow the stub entry to $A$ and from there to $C$, returning up-to-date location information to $B$. This information cannot be installed immediately. First, a control message

[3]For performance, removal messages can be batched and/or piggy-backed on other messages.

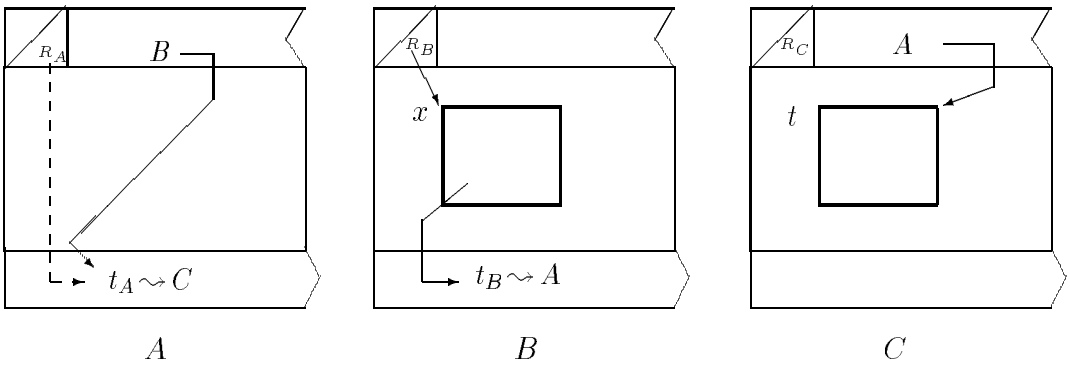


Figure 4: An inter-space indirection

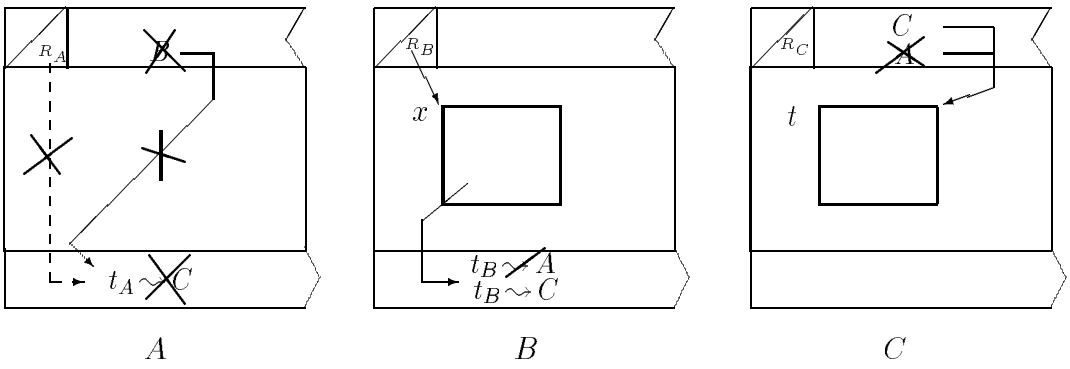


Figure 5: Elimination of an inter-space indirection

$$B \rightarrow C : \mathsf{request}\{@t\}$$

will request $@t$; this will cause $C$ to send :

$$C \rightarrow B : \mathsf{ackreq}\{@t\}$$

As for a mutator message, this will install an entry $ODT_C[m] = (B@t)$. Upon reception by $B$, stub $t_B$ will be updated: $t_B \rightsquigarrow C$. Note that the mutator can continue operating during this exchange.

The global GD protocol will thereafter cause entry $ODT_A[l] = (B@t)$ to be collected. If the dashed line from $R_A$ to $t_A$ is also removed, then $t_A$ becomes garbage, which in turn will cause entry $ODT_C[k] = (A@t)$ to be collected also. The above execution is represented in Figure 5.

It is possible to piggy-back the location hint $(@t \rightsquigarrow C)$ on the mutator message. Then, it is tempting to avoid the indirection altogether by setting the stub directly to $t_B \rightsquigarrow C$. This would be incorrect: we leave as an exercise to the reader to show that this could lead to erroneously believing that $t$ is garbage.

The correct protocol for early indirection elimination is the following.

Receiving space $B$ installs the stub $t_B \rightsquigarrow A$ as before, but informs the finder of the hint ($@t \rightsquigarrow C$). The finder then runs the elimination protocol (in the background) going directly to space $C$. The outcome of indirection elimination is the same as before, pictured in Figure 5.

## 4.7 Interaction between garbage detectors

A local detector never has to wait for a remote one, nor does it ever wait for an event of the global GD protocol or of the object finder. Conversely, neither the object finder nor the global protocol never need to wait for a local GC.

The only synchronization requirement is that installation of new information (addition, update, or deletion of an entry), in an ODT or an ERT must be an indivisible operation. Resilience to failure and permanence are not required[4], hence atomicity (in the sense of atomic transactions [8]) is not necessary.

Even while new information is being installed in an ERT or an ODT, the global and local detectors may continue to operate on an old version.

Thus, the detectors run fully parallel with each other, and with remote mutators. A detector may or may not operate in parallel with its local mutator, depending on the local GC algorithm used.

## 4.8 Inter-space cycles of garbage

We now turn to the problem of inter-space cycles of garbage. Consider object $x$ in space $A$, containing a reference to $y$ in space $B$; $y$ in turn contains a reference to $x$. Even if $x$ and $y$ are not accessible from any local root, reclaiming $A$ and $B$ necessitates an extension of the protocol.

The cycle could be reclaimed by the execution of a global garbage detector, but that is precisely what we are trying to avoid.

Another solution exists, which needs very little additional mechanism. The idea is to migrate the cycle to a single space, where it will be collected by the normal operation of local GC [3].

During local GC, two marking "colors" are used. An object accessible from the local root is marked green, whereas one accessible only from the ODT is marked red. At the end of local GC, a red object may be migrated to some space which references it, by the own authority of the local GC. This

[4] But see the definition of a persistent space in section 5.3.2.

has the desirable property of migrating an object, not used in some space, towards a space which uses it.

Two precautions are necessary. First, avoid a "thrashing" or "ping-pong" effect, such that $x$ is migrated to $B$ while at the same time $y$ is migrated to $A$. For this, some total ordering of spaces is enough (such as the order of space names); migrate a red object only to an inferior space in this ordering. With this precaution a garbage cycle will migrate to a single space, where it is collected by the local collector.

Second, migration decided by the GC must be reliable, exactly once, and all-or-nothing. If the object is persistent then its migration must be persistent too. In other words, GC-initiated migration is atomic, even though in section 5 we will allow mutator-initiated migration to fail.

## 4.9 Alternatives for detection of inter-space garbage cycles

Using migration to eliminate inter-space cycles of garbage is a simple strategy, which needs little additional mechanism. The actual collection of cycles is done by the local garbage collector, which is assumed correct.

However this strategy has a number of drawbacks. One is that it must be atomic: whereas the GD protocol (in section 5) will tolerate lossy migrations, it is not allowed to impose a loss on the application. Another drawback is that migration of a particular object is not always possible, either because of hardware constraints, or because the application specified residency in a particular space.

Therefore we are considering a number of alternatives, which appear promising, but need more work.

One solution is to keep migration, but make it virtual; this is similar to some [1] generational schemes, which avoid copying by simply marking cells with their generation number. Here, instead of moving objects between spaces, we would move the space boundaries. "Local" garbage collection remains within a logical space, but may span a number of physical sites. We are confident that we can correctly extend local GC in this way, while still retaining the independence between spaces.

We are also looking at an adaptation of classical termination protocols. For an object which is not locally accessible in some space, the detector estimates a superset (initially, the corresponding ODT entries) of the remote roots from which it might be accessible. The superset is propagated along the ERT stubs to the next ODT possibly adding more potential remote roots. Elements of the superset which are not in the true set of roots are removed. If

there is a cycle of garbage, the superset will become empty after at most two revolutions. We are protected against concurrent mutator activity, because the creation of an inter-space reference is preceded by the creation of an ODT entry.

## 4.10 Conclusion

We have considered separate spaces communicating by messages, where messages do not fail, individual spaces do not crash, and message transmission is instantaneous. These assumptions are valid when considering spaces executing on a single machine. Hence, the protocol described above is suitable for parallel garbage detection for a multiprocessor object support system. We will now examine its extension to a true distributed system.

# 5 Distributed system with failures

In a distributed system, a number of new characteristics must be taken into account. Message transmission is not instantaneous; in-transit references must be taken into account for GD. A message may be lost, losing the references and/or objects it contains. Messages may be duplicated, or arrive out of order. Spaces may be become disconnected for a long period of time and may also crash.

We extend the GD protocol presented above, to adapt to realistic distributed system behaviour. The main change to the protocol is the addition of timestamps to messages and ODTs, to account for references in transit.

## 5.1 Message transmission in non-zero time

The assumption of instantaneous message transmission allows a total ordering of events. When $A$ receives a message from $B$, it knows that previous messages in the opposite direction, from $A$ to $B$, have either been received or are lost. When message transmission takes non-zero time, as in a distributed system, this total ordering is lost. Object $x$ not currently accessible from any root may become accessible when a reference $@x$ in transit is received.

### 5.1.1 An example

Here follows an example of faulty behavior of the GD protocol of section 4 in this case:

1. Space $A$ holds a reference to object $x$ in space $B$.
2. $A$'s reference to $x$ is deleted.
3. $A$'s local garbage collector removes the stub $x_A$, sending $\mathsf{removal}\{@x\}$ from $A$ to $B$.
4. While the removal message is in transit, $B$ again sends $@x$ to $A$, then makes $x$ unreachable from $R_B$.
5. $A$ receives the message, creating a new stub $x_A$; the mutator makes $x_A$ accessible from $R_A$.
6. The removal message arrives to $B$; entry $ODT_B[i] = (A@x)$ is removed.
7. $B$'s local GC executes and removes $x$, even though it is now accessible from $R_A$.

Hence, the GD protocol must be modified to account for references and objects in transit.

### 5.1.2 Sending and receiving messages

To deal with messages in transit, we timestamp events at a space from a local, monotonically increasing "clock". Clocks need not be synchronized with each other; a count of transmitted (mutator or control) messages is sufficient for our purpose. We note $clock_A$ the current clock value of space $A$. In the figures, it is noted in the upper-left corner of the spaces, next to the root.

Each transmitted message is stamped with the value of the clock on transmission. Futhermore when $A$ sends $@x$ to $B$, the $ODT_A$ entry for $(B@x)$ is timestamped:

$$ODT_A[i] = (B@x)/clock_A$$

Each ODT entry is stamped with the clock value of the last corresponding message send. Finally, each space keeps the vector of highest timestamps received from each other space. We note $hts_A(B)$ the highest timestamp received from $B$ by $A$.

For instance in Figure 6, some messages were previously sent from $B$to $A$, at time 20 on $B$'s clock. The figure depicts a new message, carrying $@x$ to $A$, sent at instant 32, $B$'s local time, and which has not yet been received by $A$.

Figure 7 depicts the (non-instantaneous) migration of $y$ from $A$ to $B$.

$$A \rightarrow B : \mathsf{migrate}\{y\}/18$$

A stub object $y_A$ is created in space $A$ before the migration message is transmitted. Upon receiving $y$, at instant 47, $B$ replies

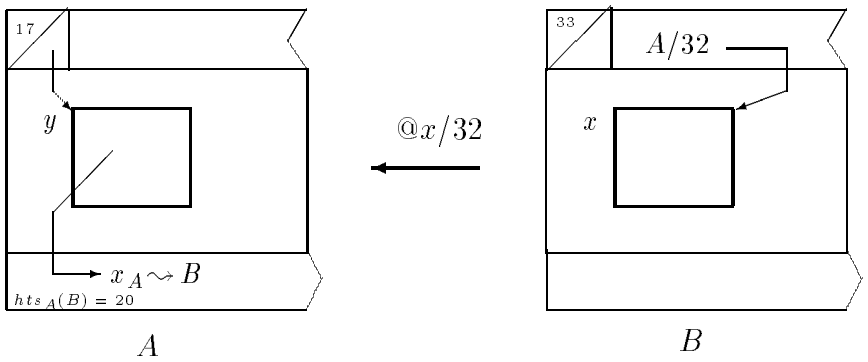


Figure 6: Clock values to deal with non-instantaneous message delivery

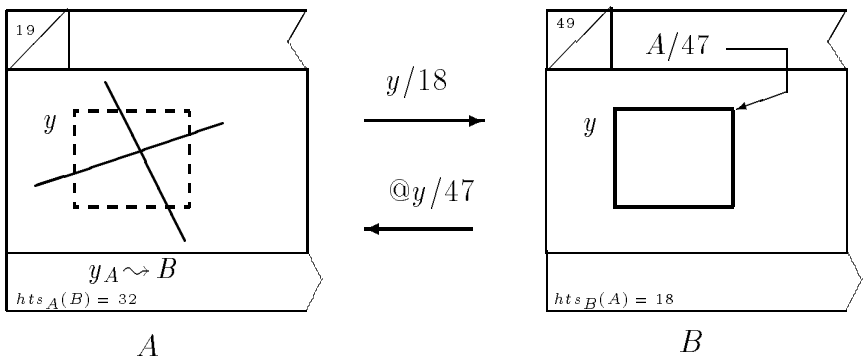


Figure 7: Non-instantaneous migration

$$B \rightarrow A : \mathsf{ackmigrate}\{@y\}/47$$

This has the usual effect of installing an entry

$$ODT_B[i] = (A@y)/47$$

timestamped with $B$'s current time. When (and if) $A$ receives this reply, its timestamp $hts_A(B)$ is updated to 47.

### 5.1.3 GD protocol accounting for messages in transit

With respect to messages in transit, the only dangerous action for the GD protocol is the removal of an ODT entry (as it could invalidate the conservative nature of the ODT). We modify the protocol, so that ODT entries for messages which are possibly in transit[5] are not collected.

When a stub at a space $A$, pointing to an object $x$ at space $B$, is collected, a removal message is sent to $B$:

$$A \rightarrow B : \mathsf{removal}\{@x, hts_A(B)\}/clock_A$$

B's corresponding ODT entry is

$$ODT_B[i] = (A@x)/\tau$$

We interpret this as follows. The last message sent by $B$ to $A$, that carried $@x$, was sent at $B$'s time $\tau$. The last message received by $A$ from $B$ was sent at $B$'s time $hts_A(B)$. If $\tau = hts_A(B)$, then the above are the same message. If $\tau > hts_A(B)$, then the $@x$ message was not received (it was either in transit or lost). If $\tau < hts_A(B)$, then a later message from $B$ to $A$ advanced $hts_A(B)$ (meaning the $@x$ message was either lost or received). To guard against the case where the $@x$ message may be in transit, the removal message is accepted only if $\tau \leq hts_A(B)$, and ignored otherwise.

For instance, in Figure 6, suppose the reference from $R_A$ to $y$ is deleted. At the next local GC (when $clock_A$ is 18), $y$ and $x_A$ are removed and $A$ sends

$$A \rightarrow B : \mathsf{removal}\{@x, 20\}/18$$

Since the entry in $ODT_B$ for $(A@x)$ is timestamped 32, it is not removed, because a message is in transit (or lost).

Ignoring removal messages introduces the possibility that a garbage ODT entry will never be cleaned. We deal with this problem in section 5.2.1.

## 5.2 Unreliable message transmission

Messages in a distributed system are subject to loss, duplication, non-FIFO ordering, and unbounded duration of delivery.

[5] Some may be lost.

We will not attempt to give a semantics to message duplication and loss, as this depends on the mutator. In particular, duplicate and lost messages containing migrating objects pose a problem. We assume that the mutator knows how to deal with them (e.g. by resending lost messages, and suppressing duplicates); independently of the mutator's recovery actions, our protocol tolerates such events.

Hereafter we consider non-FIFO, unreliable channels; other cases are similar.

### 5.2.1 Lossy channels

Message loss is a fairly common occurrence in real distributed systems. To guard against loss, communication protocols use time-out, retransmission, and acknowledgments. Retransmission in turn may cause duplication.

**Lost control messages.** Suppose that removal messages can be lost or ignored, introducing the possibility that a garbage ODT entry could remain forever. To avoid this, any two spaces $A$ and $B$ will periodically (but infrequently) exchange background control messages containing

$$A \rightarrow B : \mathsf{ERT}\{ERT_{A|B}, hts_A(B)\}/clock_A$$

where $ERT_{A|B}$ denotes the subset of stubs of $ERT_A$ pointing to space $B$. If some ODT entry $ODT_B[i] = (A@x)/\tau$ does not appear in $ERT_{A|B}$, and $\tau \leq hts_A(B)$, then that entry can be removed.

The acceptance condition $\tau \leq hts_A(B)$ guards against references in transit; it is the same as for removal messages (section 5.1.3).

Between any two spaces, we assume that some control message containing ERTs eventually reaches its destination; this guards against the loss of removal messages and of previous ERT messages. This ensures that all garbage ODT entries are eventually deleted.

Messages migrating objects (to resolve inter-space cycles of garbage) are *not* allowed to be lost; indeed we have stated in section 4.8 that such messages are atomic.

**Lost mutator messages.** Our protocol, as described in section 4, already tolerates loss. To see this, remember that its key points are: first, maintaining a conservative estimate of each space's ODT; and second, using local GC to remove garbage entries from ERTs and hence from ODTs.

Even in the presence of message loss, the conservative nature of ODTs is maintained, since an ODT entry is made before the actual message trans-

misssion, and removed only after it is known to point to garbage. Therefore, garbage detection remains correct.

Let us now show that garbage detection remains effective. Local GC remains as before. We need to show, additionally, that even if an object becomes garbage by loss of a message containing the last reference to it, it will be detected. Consider the following example: object $x$ is in space $A$. A single reference to $x$ exists, from $R_A$. Initially $hts_B(A) = 97$. At time $clock_A = 100$, $A$ sends

$$A \rightarrow B : \mathsf{mutator}\{@x\}/100$$

thus installing

$$ODT_A[12] = (B@x)/100$$

then deletes the reference from $R_A$ to $x$. The message is lost; $x$ is now garbage. Now $B$ sends

$$B \rightarrow A : \mathsf{ERT}\{ERT_{B|A}, 97\}/1000$$

where $ERT_{B|A}$ contains no entry for $x$. However, since $100 \not\leq 97$, $ODT_A[12]$ is not removed, i.e. the lost message is conservatively treated as in transit. Later, a new message will reach $B$, causing $hts_B(A)$ to be updated to say 110. The following ERT message

$$B \rightarrow A : \mathsf{ERT}\{ERT_{B|A}, 110\}/1033$$

will then cause $ODT_A[12]$ to be deleted, since $100 \leq 110$.

**Duplicate messages.** Duplication of a message causes no problem. Its only effect will be to redo the same action twice. GD protocol actions are all idempotent.

### 5.2.2 Non-FIFO ordering

Our use of timestamps to guards 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 protocol can tolerate some amount of non-FIFO ordering. The only use of FIFO ordering is the rejection criterion for removal and ERT messages, in case a reference is in transit. An out-of-order message could invalidate this criterion.

However non-FIFO orderings are also acceptable if the following *acceptance condition* is added for receiving mutator messages. Once a removal message or ERT message has been sent by $A$ to $B$, carrying $hts_A(B)$, earlier messages from $B$ to $A$ (i.e. with timestamp less than $hts_A(B)$) are ignored, i.e. considered lost.

This condition allows the application some flexibility in the order in which messages are processed.

## 5.3 Deletion and termination

### 5.3.1 Normal and abnormal deletion

We allow application programs to explicitly delete an object even though it may be reachable. We stated in section 3.4 that the object finder resolves a reference to a deleted object by signalling an exception deleted. If deletion is implemented by forgetting all information about the object, this entails a very expensive exhaustive search through all existing spaces. In this section, we propose a deletion protocol between the mutator, the finder, and the detector. This protocol ensures that object finding and garbage detection are not slowed down by repeated exhaustive search for deleted objects. In keeping with the rest of this paper, the deletion protocol tolerates all kinds of failures, including undisciplined mutator action.

A well-behaved mutator deletes an object by setting its state to deleted. This is called *normal* deletion.

However this will not always work. For instance deletion may occur as a side-effect of lost message or a crash. Or a mutator may enter a "panic mode" in response to some serious error, and zero out a portion of memory to recover to a known state. The following *abnormal deletion* protocol of the finder allows to recover back to the normal case. Consider the resolution of $@x$ in space $A$. If $@x$ points to an object, the algorithm terminates as before.

If $@x$ points to a stub $x_A \rightsquigarrow B$, and $B$ has no corresponding ODT entry $(A@x)$, then $x$ has been deleted abnormally; recreate it in the deleted state, and signal deleted.

If however $B$ is disconnected (see below) then wait for it to either reconnect or terminate, and try again. In the meantime (as an optimization) a search through other spaces can be attempted.

We deal below, in section 5.3.2, with the case where $B$ terminates.

### 5.3.2 Disconnection and termination of spaces

We now examine space disconnection and termination. When a space terminates, it ceases to compute, and all the objects and references it contains are deleted. There is a normal, well-behaved termination protocol, and an abnormal one, where the space is wiped out; if references to it exist, it recovers to the normal case.

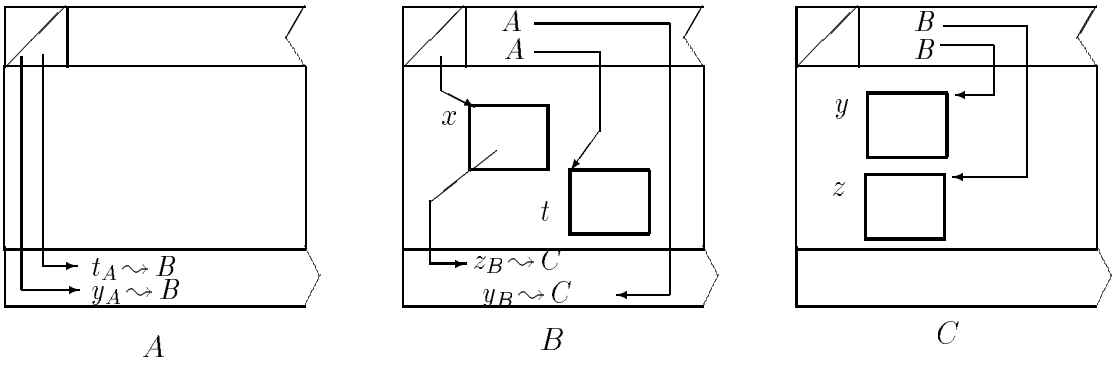


Figure 8: Termination of a space

There is also an intermediate state, where the space is *disconnected*, i.e. it is impossible to communicate with it. Eventually a disconnected space either reconnects or terminates abnormally.

**Normal termination.** Consider the normal termination of space $B$ in Figure 8; the protocol must take into account the reference from space $A$ to $t$, and the indirect reference to from $A$ to $y$ (in space $C$) via $B$. The normal termination protocol for $B$ is the following:

1. Remove all references from the root $R_B$,
2. set all local objects to the deleted state, but leave stubs intact,
3. perform a local garbage collection. The only objects which remain are:
   - stubs referenced from the object directory table $ODT_B$ (indirections)
   - deleted local objects referenced from the object directory table $ODT_B$.
4. Perform indirection elimination for all remaining stubs.
5. Re-create deleted local objects in every space which refers to them.
6. Collect space $B$.

For performance reasons, it is preferable to await an acknowledgment of steps 4 and 5. Otherwise, if a message is lost, then the more expensive abnormal termination protocol, below, might be invoked.

**Disconnection.** When communication with some space appears to be impossible for an extented period of time, it is said to be disconnected. Disconnection can occur for instance when a network partitions.

While some space $B$ is disconnected, no references into $A$ can be resolved (the finder must wait), and garbage detection is partially disabled, since ODT entries of the form $(B@x)$ cannot be removed.

A disconnected space will eventually either reconnect or terminate. User intervention may be necessary to force one or the other outcome. If it reconnects, waiting protocols may proceed. If it terminates, we run the *anormal termination* protocol below.

**Abnormal termination protocol.** Consider now the abrupt, abnormal termination of $B$ in Figure 8. All of its contained objects and references are lost, as well as $ODT_B$ and $ERT_B$. Stub $t_A$ is now a dangling reference. Object $z$ in space $C$ is now garbage; $y$ is not, being reachable from $A$. In the absence of the information lost in $B$'s termination, $C$ cannot distinguish between these two cases. In particular it is incorrect to assume $ERT_B$ is empty, since this could cause $y$ to be incorrectly collected.

A small addition to the protocols accounts for this problem. The finder protocol is changed, to perform an exhaustive search when it encounters a stub pointing to a space which terminated abnormally. This ensures that $t_A$ will be resolved into a $t$ in the deleted state, that $y_A$ is updated to point to $C$, and that a new entry is allocated in $C$'s object directory table $ODT_C[j] = (A@y)$.

Furthermore when this occurs, $B$ is re-created in a special *zombie* state, in which it can only wait for possible indirections through $B$ to be eliminated (by the protocol described in section 4.6). In this state, $B$ waits to receive from every other (non-terminated, non-zombie) space $D$ an empty $ERT_{D|B}$. When it has received them all, $B$ sends its (now empty) ERT to other spaces and awaits an acknowledgment. When all acknowledgments are received, $B$ can be collected.

**Discussion.** Two examples of abnormal termination are a site crash (which crashes all the contained spaces) and killing a process (where the process is a space). In both cases the space terminates abruptly; the abnormal termination protocol recovers to the normal case.

We assume crashes are fail-stop, therefore the only consequence of a crash is disconnection, loss of volatile memory, and halt of computation. Objects and references stored only in volatile memory disappear; objects and references in non-volatile memory persist across crashes, and become active again when the space recovers. Our problem is to ensure that ERTs and ODTs remain consistent through the crash and recovery, that GD of non-crashed spaces continues as unperturbed as possible, and that no objects

which will be reachable after recovery are incorrectly believed to be garbage during the crash.

During the time that space $B$ is crashed, its set of live objects does not change. $B$ also ceases to send messages entirely; therefore, for the duration of $B$'s crash, object directory table entries in other spaces $A$, $C$, etc., containing something like $(B@x)$, will not be collected. $ODT_C$ will continue to contain a superset of the objects potentially reachable from $B$; hence $C$'s garbage detection remains correct. The collecting of objects potentially reachable from $B$ is frozen; however the collecting of objects not reachable from $B$ continues undisturbed.

Without loss of generality, we will suppose there are only two possible outcomes for the crash of some space $B$:

1. $B$ terminates. All the objects it contained at the time of the crash are deleted. $ODT_B$ and $ERT_B$ are lost. $B$ is called a *volatile space.*
2. $B$ recovers and reconnects. An object it contains either persists or is deleted. Its object directory table $ODT_B$ persists. $B$ is called a *persistent space.*

Let us first consider the recovery of a persistent space. Our most important assumption is that, even though some of the objects it contains may be lost(i.e. deleted abnormally), its object directory table persists, hence $ODT_B$ continues to contain a superset of its remotely reachable objects, and the garbage detection protocol remains correct. Unused External Reference Table entries will be recovered by the next run of the local garbage collector; therefore the garbage detection protocol continues to detect garbage.

When a volatile space crashes, all information about the objects it contained is lost. If any references to them remain elsewhere in the system, run the abnormal termination protocol to recover.

# 6 Recapitulation

Although our garbage detection protocol is conceptually simple, many aspects have to be taken into account to ensure its correct execution in the face of faults, and to be sure that the interactions beteen components are correct. Let us recapitulate its important points. We set between square brakets [...] those steps which have been added to the protocol between sections 4 and 5, i.e. are needed to tolerate failures and abnormal events. In this section we ignore the elimination of indirections.

- *Object Directory Table.*
  A space $A$ maintains an Object Directory Table $ODT_A$, containing

one entry $ODT_A[i] = (B@x)/\tau$ per couple {*object x, remote space B possibly pointing to x*}. [This entry is timestamped with the latest date $\tau$ at which $A$ sent $@x$ to $B$].

- *External Reference Table.*
  Each space $A$ also maintains an External Reference Table $ERT_A$ containing a stub object $y_A$ for every object $y$, possibly remotely referenced within $A$ (including, possibly, from $ODT_A$). $y_A$ contains location information $\rightsquigarrow B$, i.e. the space $B$ where $y$ was last known to reside.

- [*Clocks and timestamps.*
  Every space $A$ maintains a local clock $clock_A$, a number which increases monotonically with messages.

  Every (control or mutator) message sent is timestamped with the value of the sender's clock.

  Each space $A$ maintains a vector of highest timestamps most recently received from other spaces, $hts_A(B)$, $hts_A(C)$, etc. Timestamps from different spaces are not comparable.]

- *Local garbage collection.*
  A local tracing garbage collector runs in every space, asynchronously with activity in all other spaces (detectors, finders, and messages). The root for local garbage collection of space $A$ is $R_A \cup ODT_A$.

  When local garbage collection removes an unreachable stub object $y_A \rightsquigarrow B$, it may [optionally] send a removal control message, instructing $B$ to remove its ODT entry for $y$.

  When local garbage collection finds an object $x$ accessible from $ODT_A$ but not from $R_A$, $x$ is atomically migrated to some space which refers it.

- *Object finder.*
  A reference $@z$ uttered in space $A$ is resolved by $A$'s object finder.

  1. If $z$ is know to be deleted, the finder signals an exception.
  2. If $z$ is known to reside in $A$, the finder returns.
  3. If $@z$ refers to a stub $z_A \rightsquigarrow B$:
     (a) If $B$ is connected, and there is an entry $ODT_B[i] = (A@z)$, then recursively invoke $B$'s finder. Possibly update $z_A$ with location information returned.
     (b) [If $B$ is connected and there is no such entry, then $z$ is deleted. Recreate $z$ in the deleted state and signal exception.]
     (c) [If $B$ is disconnected, search other spaces; if this fails to locate $z$, then wait until $B$ reconnects or terminates.]

(d) [If $B$ is terminated, run the abnormal termination protocol.]

- *Sending a mutator message.*
  [Every message from $A$ is timestamped with the current value of $clock_A$.]

  $A$ sends message $\mathsf{mutator}\{\ldots, @x, \ldots\}$ to $B$: First, create ODT entry $ODT_A[i] = (B@x)$, if it does not already exist. [Then, timestamp $ODT_A[i]$ with the current value of $clock_A$.] Finally, transmit the message. If a stub $x_A \rightsquigarrow C$ exists in $A$ then location hint $(@x \rightsquigarrow C)$ is piggy-backed on the message.

  Message

  $$A \rightarrow B : \mathsf{mutator}\{\ldots, y, \ldots\}$$

  means object $y$ is being migrated from $A$ to $B$. Before actually transmitting the message, create (or update) a stub object $y_A \rightsquigarrow B$.

- *Receiving a mutator message.*
  Space $B$ receives

  $$A \rightarrow B : \mathsf{mutator}\{\ldots\}/\tau$$

  [Update $hts_B(A)$ to $\tau$ if most recent.]

  If the message contains a reference $@x$, $B$ installs (or updates) a stub object $x_B \rightsquigarrow A$.

  If the message carries a piggy-backed hint $(@x \rightsquigarrow C)$, receiver $B$ triggers the finder with the hint, in order to eliminate the indirection.

  If the message contains a migrated object $y$[received at time $clock_B = \tau'$], create an ODT entry $ODT_B[k] = (A@y)/\tau'$. Then deliver the message to the mutator. [Finally send

  $$B \rightarrow A : \mathsf{ackmigrate}\{@y\}/\tau'$$

  to $A$].

  [An out-of-order message from $A$ is accepted, only if its timestamp is not less than some value of $hts_B(A)$, previously sent to $A$.]

- *Control messages.*
  When a space $B$ receives from $A$ a message

  $$A \rightarrow B : \mathsf{removal}\{@x, hts_A(B)\}/\tau$$

  delete the corresponding entry $ODT_B[j] = (A@x)/\tau'$ [if $hts_A(B) \leq \tau'$]. [Update $hts_B(A)$ to $\tau$, if most recent.]

  [Consider some pair of spaces $A$ and $B$. Each periodically sends the other the latest version of its ERT, and the highest received timestamp:

  $$A \rightarrow B : \mathsf{ERT}\{ERT_{A|B}, hts_A(B)\}/clock_A$$

]

[When space $B$ receives $\mathsf{ERT}\{ERT_{A|B}, hts_A(B)\}$ from $A$, it examines $ODT_{B|A} \cap ERT_{A|B}$ (where $ODT_{B|A}$ is $B$'s object directory table, restricted to those entries marked as referred from $A$). Every entry in $ERT_{A|B}$ must have a corresponding $ODT_{B|A}$ entry[6]. Conversely, an entry in $ODT_{B|A}$ for which there is no corresponding entry in $ERT_{A|B}$, can be removed, unless its timestamp is greater than $hts_A(B)$.]

- *Recovery after abnormal termination.*
  [If a stub pointing to a terminated space $B$ is encountered, then the latter has terminated abnormally. The space is re-created in the *zombie* state. $B$ waits to receive from every other (non-terminated, non-zombie) space $X$ an empty $ERT_{X|B}$. When it has received them all, $B$ sends its (now empty) ERT to other spaces and awaits an acknowledgment. When all acknowledgments are received, $B$ can be collected.]

# 7 Conclusion

We presented a distributed garbage detection protocol. It is based on weak, realistic assumptions, making it usable for a general-purpose object-support system. Its limiting assumptions are that crashes are fail-stop, and that messages are delivered (if at all) uncorrupted, in finite time.

Until recently, garbage collection has been often judged too language-dependant, too complex and too costly for general-purpose systems. However object-support systems need the valuable service of garbage collection. Our approach is to provide a generic service for distributed garbage detection, building upon existing, language-dependent, local garbage collectors. The cooperation between local activity (mutators and collectors), and the global detection protocol, is limited to simple interactions to maintain the Object Directory and External Reference Tables.

When a mutator sends a reference, it is first added to the local ODT. The local collector traces from the union of the local root with the ODT. The collector removes local garbage objects, as well as garbage stubs of the ERT. No synchronization is needed between the global service and the local collector or mutator.

Our protocol requires only slight modifications to the standard local garbage collector. It is simple and deals gracefully with common error occurences of real distributed systems.

[6] But $B$ may have more recent location for it; in which case it will send the new information to $A$, allowing $A$'s finder to eliminate the indirection.

Scalability is an important property in real distributed systems. In our protocol, garbage collection is done locally, and there is no global mechanism (e.g. no global synchronization). Moreover, we rely only on local information and information exchanged between pairs of sites. For all these reasons, garbage collection is parallel, and our protocol is scalable to very large systems.

For the same reasons, spaces may be organized in a hierarchy. Garbage collection at each level behaves as the one-level protocol presented.

The detection of inter-space cycles of garbage is the weak point of many proposals, and ours is no exception. In this paper we classically propose to migrate locally unreachable objects, leaving cycle removal to local GC. However we are aware of the limitations of this. We propose a few promising alternatives, which need deeper investigation.

Much more work remains. Alternative solutions for dead cycle elimination will be explored. Two implementations of the protocol are planned: one for a family of distributed object-support operating systems (INRIA project SOR), and another for a distributed object-support database system (INRIA project Sabre). Finally, formal proofs of correctness are necessary.

## Acknowledgments

We wish to thank our colleagues of INRIA for their helpful comments and discussions, in particular Bernard Lang, Jean-Jacques Lévy, and Damien Doligez.

# References

[1] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Western Research Laboratory, Palo Alto, CA (USA), February 1988.

[2] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, pages 117–187, Eindhoven (the Netherlands), June 1987. Springer-Verlag.

[3] P. B. Bishop. Computer systems with a very large address space, and garbage collection. Technical Report MIT/LCS/TR-178, Mass. Institute of Technology, MIT Laboratory for Computer Science, Cambridge MA (USA), May 1977.

[4] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9):807–820, September 1988.

[5] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park, Arizona USA, December 1989. ACM.

[6] André Couvert, Aomar Maddi, and René Pedrono. Partage d'objets dans les systèmes distribués. Principes des ramasse-miettes. Rapport de recherche 963, INRIA, January 1989.

[7] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions and consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), 1976.

[9] Robert Joseph Fowler. Decentralized object finding using forwarding addresses. Technical Report 85-12-1, Department of Computer Science, University of Washington, Seattle, WA (USA), December 1985.

[10] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Programming Languages Design and Implementation*, volume 24 of *SIGPLAN Notices*, pages 313–321, Portland OR (USA), June 1989. SIGPLAN, ACM Press.

[11] Olivier Gruber and Patrick Valduriez. Uniform object management for parallel database servers. In *Data Management and Parallel Processing*. Chapman and Hall, London, 1990.

[12] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov. COOL: Kernel support for object-oriented environments. In *Joint ECOOP/OOPSLA Conference*, Ottawa (Canada), October 1990.

[13] Won Kim et al. Features of the Orion object-oriented database system. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. ACM Press/Addison-Wesley, 1989.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[15] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 29–39, Vancouver (Canada), August 1986. ACM.

[16] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[17] Fernando Velez, Guy Bernard, and Vineeta Darnis. The $O_2$ object manager: an overview. Technical Report 27-89, GIP-Altaïr, Rocquencourt (France), February 1989.

[18] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, Dept. of Comp. Sc., U. of Washington, Seattle WA (USA), January 1987.

[19] Mark Weiser, Alan Demers, and Carl Hauser. The Portable Common Runtime approach to interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, Litchfield Park AZ (USA), December 1989. ACM.