A Distributed Garbage Collector with Diffusion Tree Reorganisation and Mobile Objects

Luc Moreau
Department of Electronics and Computer Science
University of Southampton, Southampton SO17 1BJ UK
L.Moreau@ecs.soton.ac.uk

Abstract
We present a new distributed garbage collection algorithm that is able to reorganise diffusion trees and to support mobile objects. It has a modular design comprising three components: a reliable transport mechanism, a reference-counting based distributed garbage collector for non-mobile objects, and an extra layer that provides mobility. The algorithm is formalised by an abstract machine and is proved to be correct. The safety property ensures that an object may not be reclaimed as long as it is referred to locally or remotely. The liveness property guarantees that unreachable objects will eventually be reclaimed. The mobility property certifies that messages are always forwarded towards more recent mobile object positions.

1 Introduction
Distributed object systems provide programmers with the capability to refer to remote objects and to activate remote computations (generally called remote method invocation) [27, 28]. In this context, distributed garbage collection is a valuable technology as it automatically maintains pointer consistency: it ensures that an object will not be reclaimed as long as it is referred to locally or remotely.

The distributed agent model of computing [2, 18] is an alternative to the traditional approach of distributed computing because it is able to deal with intermittent connections. According to the agent paradigm, users delegate a task to a program, which attempts to solve it, autonomously given some resource constraint, possibly by migrating to remote sites.

Mobile computations put an extra burden on the distributed garbage collector as they typically abandon chains of forwarding pointers. It is desirable to short-cut these chains not only to accelerate the access to remote mobile objects, but also to make them independent of the previous hosts they visited (which is precisely one of the goals of the mobile agent model).

For this purpose, we have designed NeXene [20] a distributed extension of Scheme [25] with mobile objects and primitives to control resource consumption [21]. The goal of this paper is to describe its distributed garbage collector based on distributed reference counting. Its major features are:

- It uses a new flexible way of reorganising diffusion trees which is suitable for immobile and mobile objects.
- It separates concerns in different modules: (i) Reliable message-parsing and FIFO handling is provided by the transport mechanism; (ii) The distributed garbage collector deals with pointers to immobile objects; (iii) Object mobility is provided as a layer on top of the garbage collector.
- The distributed garbage collector can be implemented as a language independent library.

NeXene [20] is a distributed Scheme based on the message-passing library Nexus [7], which essentially provides two mechanisms: the remote service request is a form of remote procedure call, and global pointers provide for global naming in a distributed environment. Nexus runs on a variety of hardware and protocols, including workstations or supercomputers, and TCP/IP or UDP.

The garbage collector we present in this paper is based on reference counting. As other reference-counting algorithms, ours is unable to reclaim distributed cycles. However, we should observe that there is a range of applications that do not create distributed cycles. In particular, Tel and Mattei [29] have shown that the problem of termination in distributed systems is equivalent to distributed GC. Reference counting can be used because processes form a hierarchy. Groups [21] also have a hierarchical organisation and can be reference counted.

A preliminary and very schematic description of the distributed garbage collector appeared in [20]. This paper, which covers it in details, is organised as follows. In Section 2, we describe the Nexus programming model and define some terminology that we shall use in the rest of a paper. In Section 3, step-by-step, we intuitively present our algorithm for distributed garbage collection. In Sections 4 and 5, the algorithm is formalised by an abstract machine, and its correctness is established by proving two properties: safety and liveness. Implementation issues are discussed in Section 6. The mobility layer is studied in Section 7. Finally, a comparison with related work concludes the paper.

Copyright 1998 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must also be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

ICFP’18 ©2018, Baltimore, USA.
2 The Nexus Programming Model

The Nexus [7] philosophy is derived from *active messages* [30], where each message contains at its head the address of a user-level handler executed on message arrival. Computations execute on a set of sites and consist of a set of threads. An individual thread may read and write data shared with other threads executing in parallel on the same site.

Nexus defines two abstractions: the global pointer and the remote service request. The global pointer (GP) provides a global name space for objects, while the remote service request (RSR) is used to initiate communications and to invoke remote computations. A GP is a name for an object, and it specifies a destination to which a communication can be directed by an RSR. GPs can be created dynamically; once created, a GP can be communicated between sites by including it in an RSR. As far as distributed garbage collection is concerned, a GP can be regarded as a remote object reference, sometimes called a reference [17].

Practically, in Nexus, an RSR is specified by providing a global pointer, a handler identifier, and some arguments. Issuing an RSR causes the arguments to be transferred to the site designated by the global pointer, after which the routine specified by the handler is executed. Both a copy of the arguments and the pointed object are available the RSRs handler. As opposed to the traditional remote procedure call, a remote service request does not return a result; if a result is needed, another RSR has to be used.

Handlers may be executed in a new thread of control. If handlers are not threaded, Nexus provides a FIFO ordering of RSls in addition to a reliable transport protocol.

We define the notion of *owner*, *emitter*, and *receiver*. The *owner* of a GP is the site where the GP is pointing at. The *emitter* of a GP is a site sending an RSR containing the GP; the emitter may or may not be the owner. The *receiver* of a GP is a site receiving an RSR containing the GP; the receiver may or may not be GP's owner.

3 The Intuition

In this Section, we progressively describe our new distributed garbage collection algorithm. First, we present how reference counting can be integrated with the Nexus programming model. Initially, we deal with two sites only; then, we show that a straightforward extension to more sites fails to be satisfactory. Finally, we introduce our solution.

3.1 Two sites

Each site uses a thread safe, conservative, mark and sweep garbage collector [10] to reclaim unused space locally. Conservativeness is required as Scheme data are passed to Nexus, written in C, and are pointed by Nexus data structures.

Let us consider a data on a site s1 and a GP pointing at this data. The purpose of a distributed garbage collector is to ensure that the data on s1 is not reclaimed as long as GP remains reachable locally or remotely. In order to deal with distribution, we use a reference counting technique. The first time GP is sent to a remote site during an RSR, it is associated with a counter initialised to one. Afterwards, every time the same GP is sent, its associated counter is incremented by one.

We use a table to maintain associations between counters and global pointers that were sent to remote sites. We call this table send-table as it is used when RSls are sent. In a first approximation, the send-table indicates the number of times a global pointer was sent. The send-table is constructed as a root of the local garbage collector. As a result, by its presence in the send-table, GP remains reachable from the local collector roots, which ensures that the space used by the data referenced by GP cannot be reclaimed.

In order to keep reference counters up-to-date, each site has to be able to determine whether a GP has already been received. For this purpose, each site maintains a second table called receive-table[1], which contains the global pointers that have already been received. A GP also appears in its owner's receive table.

So far, the situation is summarised by Figure 1 where GP is sent from s1 to s2; GP is entered in the send-table of s1 with the counter value 1 and in the receive-table of s2.

![Figure 1: GP sent from s1 to s2](image)

In addition to reference counters, the distributed garbage collection algorithm uses control messages, whose purpose is to update counters. A decrement message is aimed at a site and contains a global pointer GP. When the destination site receives such a message, it decrements the counter associated with GP in its send-table; if the counter reaches 0, the entry for GP is removed from the send-table.

We use decrement messages in two different situations. First, when a GP received by a site becomes garbage on this site, GP is removed from the receive table and a decrement message is sent to GP's owner. In Figure 2, as soon as GP becomes inaccessible on s2, a decrement message is sent to s1. GP is removed from the send-table of s1, and the space can then be reclaimed on s1 if no longer used.

Second, when a GP is received by a site that already owns a copy of the GP (in its receive table), a decrement message has to be sent back to the emitter so as to maintain accurate reference counters. Note, we can refine the counter description: a counter in a send-table represents the number of different remote copies of a GP plus the number of messages related to it in transit.

[1] We call our tables send and receive because they are used when sending or receiving global pointers, respectively. Other names may be found in the literature: entry and exit items [15, 23], scions and stubs [24], or incoming and outgoing reference tables [9].
3.2 More than Two Sites

Let us now consider three sites. The right-hand side of Figure 1 displays the situation after GP is passed from s1 to s2. Using the same principle, Figure 3 presents the setting after GP is passed from s2 to s3; send-tables in s1 and s2 contain entries for GP, which also appears in receive-tables in s2 and s3.

![Figure 3: Indirect Counters Along the Diffusion Tree](image)

In fact, the mechanism we describe here bears a strong resemblance with indirect reference counting [22], where the sum of reference counters across the diffusion tree of a GP is the number of its remote copies. Besides, when a site receives a GP that it has already got, a decrement message is sent to the GP emitter, that is, to its parent in the diffusion tree.

However, the nature of Nexus global pointers results in a different usage of the other kind of decrement message. Let us recall that, when a GP becomes garbage, a message is sent to its owner. This design decision is motivated by the fact that a Nexus GP only refers to its owner site, and has no information about the site that emitted it.

Unfortunately, untimely decrement messages may be the consequence as illustrated in Figure 4. If the GP sent to s3 becomes garbage, s3 sends a decrement message to s1, that is, the GP's owner. The effect of the decrement message is to remove GP from the send-table of s1, and possibly to reclaim its space if no longer reachable. This clearly results in an inconsistent situation as GP may still be active on s2.

![Figure 4: Untimely Decrement Message](image)

Besides the incorrectness related to the decrement message, such an indirect reference counter technique may keep some pointers active longer than needed; in other words, this results in a form of memory leak. Indeed, in Figure 3, the space occupied by GP on s2 (as well as its entry in the send-table) cannot be reclaimed even if GP is no longer needed on s2.

3.3 Diffusion Tree Reorganisation

Our solution to both the untimely arrival of messages and memory leaks involves a new type of message, called increment-decrement. An increment-decrement message involves three different sites s1, s2, s3, respectively, the owner, the emitter and the receiver of a GP. When GP reaches the receiver for the first time, an increment-decrement message is sent to its owner. When s1 receives an increment-decrement message, it increments GP's reference counter, and then sends a decrement message to s2 concerning GP.

![Figure 5: Increment-Decrement Message](image)
Let us go back to Figure 3, where s2 sent to s3 a GP owned by s1. If s3 receives GP for the first time, s3 sends an increment-decrement message to s1 concerning GP and s2. The result is to increment GPs counter on s1 (Figure 5) and to send a decrement message from s1 to s2 (Figure 6). The effect is to prune the diffusion subtree rooted at s3, and to graft it to the root s1.

![Diagram showing decrement message](image)

**Figure 6:** Decrement Message

Introducing the increment-decrement message is not sufficient to avoid untimely message arrivals. The increment-decrement message from s3 should be received by s1 before any decrement message from s3 about the same GP. This can be enforced by adding an extra constraint: a site is allowed to send a decrement message to the owner of an object only if there is no pending increment-decrement message for the same object. The constraint can be implemented by in-order message delivery.

In Figure 6, we can observe that if GP is no longer reachable on s2, its space can safely be reclaimed. Such a property is particularly important in the presence of mobile computations, jumping from sites to sites. The diffusion tree reorganisation provided by the increment-decrement message prevents the formation of chains of pointers abandoned by mobile computations.

This Section has described the algorithm principle. The following sections concern its formalisation and correctness proof. We then study its implementation and some optimisations.

4 Abstract Algorithm

In this Section, we formalise our garbage collection procedure into an abstract algorithm. We model our distributed garbage collector by an abstract machine, called the DGC-machine, whose state space appears in Figure 7. In the DGC-machine, we only model messages exchanged by the distributed garbage collector, and we do not model any form of computation.

A finite number of sites take part to the DGC-machine. We define a global pointer as a pair composed of an address and a site; by definition, the address is local to the site. The owner of a global pointer GP = (o, s) is by definition the site s, which we write owner(GP). Three types of messages can be exchanged in the DGC-machine. They follow a same scheme: NAME(emit, receiver, arguments, ...), where the emitter and receiver are sites, and arguments are messages-depending. Table 1 represents the messages associated with the sites and global pointers with numbers or booleans, respectively. Finally, a DGC-configuration is given by a tuple of sites, global pointers, send tables, receive tables, and a pool of messages. A pool of messages is a bag, which represents messages in transit in the system. By transit, we mean messages that are opened to be sent, messages that are being transferred, and messages that are received but not handled yet.

The initial configuration D0 is defined as follows:

\[ D_0 = (S, G, S, G, sendT_r, recT_r, \emptyset) \]

where the initial send table sendT_r = \{s, GP\} contains zero for every entry, and the initial receive table recT_r = \{s, GP\} = \{s, G\} for the owner(GP) is false for every entry, except on sites that own a GP.

Three kinds of legal transitions can evolve the state of the DGC-machine. According to Figure 8, a transition transforms a configuration D0 into a configuration D1, using configuration transformers defined in Figure 9. These transformers are a formal representation of the algorithm.

Using these three transitions, the DGC-machine is able to reach all the different configurations that would be reached in a real implementation. We model the duplication of a global pointer, during a remote method invocation or remote service request, by the (make-copy) transition, which involves an emitter s1 and a receiver s2. Messages in the pool of messages can be handled by a (receive) transition. Finally, local collectors call a finalisation method (HL) on global pointers that are no longer reachable from the set of roots; this is modelled by the transition (delete), which means that a copy of a GP on a site is deleted. There is a side-condition (delete) by which the owner is prevented to send such DEC messages.

In Figure 9, we use the following conventions. Let \((S, G, sendT_r, recT_r, M)\) be a configuration:

1. \(sendT_r(s, GP) := V\) denotes \((S, G, sendT_r, recT_r, M)\), such that sendT_r(s, GP) = V and sendT_r(s, GP') = \(sendT_r(s, GP')\) if GP' \(\neq GP\).
2. recT_r(s, GP) := V is similar.
3. post(m) denotes \((S, G, sendT_r, recT_r, M \cup \{m\})\), with \(\cup\) the union operator on bags.

Finally, configuration transformers are executed atomically.

The configuration transformer MAKE_COPY is used in transition (make-copy). According to MAKE_COPY side-conditions, a transition (make-copy) is permitted if the emitter has access to the GP to be sent. The effect is to increase the emitter's send-table entry for GP and to post a COPY message, modelling a GP duplication. When a COPY message is received, either a DEC or an INC_DEC message is posted, according to the presence of GP in the receive table. Receiving an INC_DEC message increases the owner send-table and posts a DEC message. Receiving a DEC message decreases the send-table. Finally, a DELETE transition is allowed on a site that is not the owner of GP and does not contain GP in its send-table. The effect is to set the receive-table to false and to post a DEC message to the owner.
\[
\begin{align*}
S \in S & \triangleq \{ s_0, s_1, \ldots, s_n \} \quad \text{(Set of Sites)} \\
GP \in G & \triangleq (\alpha, s) \quad \text{(Set of Global Pointers)} \\
\alpha \in \text{Addr} & \triangleq \{ a_0, \ldots \} \quad \text{(Address)} \\
\text{msg} \in \text{Msg} & \triangleq \text{DEC}(s_1, s_2, GP) \mid \text{INC}, \text{DEC}(s_1, s_2, GP, s_3) \mid \text{COPY}(s_1, s_2, GP) \quad \text{(Message)} \\
M & \triangleq \text{BagOf}(\text{Msg}) \quad \text{(Pool of Messages)} \\
\text{sendLT} & : \mathbb{S} \times G \rightarrow \mathbb{N} \quad \text{(Send Tables)} \\
\text{recvLT} & : S \times G \rightarrow \text{Bool} \quad \text{(Receive Tables)} \\
\text{D} \in \text{Config} & \triangleq \langle S, G, \text{sendLT}, \text{recvLT}, M \rangle \quad \text{(DCC-Configuration)} \\
\end{align*}
\]

**Figure 7:** State Space of the DCC-Machine

\[
\begin{align*}
D_1 & \Rightarrow \text{MAKE}, \text{COPY}(s_1, s_2, GP) \Rightarrow D_2 \quad \text{(make-copy)} \\
& \quad \text{if } s_1, s_2 \in S, \ GP \in G, \ \text{with } D_1 = \langle S, G, \text{sendLT}, \text{recvLT}, M \rangle \\
D_1 & \Rightarrow \text{RECEIVE}(m) \Rightarrow D_2 \quad \text{(receive)} \\
& \quad \text{if } m \in M, \ \text{with } D_2 = \langle S, G, \text{sendLT}, \text{recvLT}, M \rangle \\
& \quad \land \ \text{if } m = \text{DEC}(s_1, s_2, GP), \ \text{then } \text{INC}, \text{DEC}(s_1, s_2, GP, s_3) \notin M, \forall s_3 \\
D_1 & \Rightarrow \text{DELETE}(s, GP) \Rightarrow D_2 \quad \text{(delete)} \\
& \quad \text{if } s \in S, \ GP \in G, \ \text{with } D_1 = \langle S, G, \text{sendLT}, \text{recvLT}, M \rangle \\
\end{align*}
\]

**Figure 8:** Transitions of the DCC-Machine

\[
\begin{align*}
\text{MAKE}, \text{COPY}(s_1, s_2, GP) & \quad \text{if } s_1 \neq s_2 \land (s_1 = \text{owner}(GP) \lor \text{recvLT}(s_1, GP)) \\
& \quad \{ \ \text{sendLT}(s_1, GP) = \text{sendLT}(s_1, GP) + 1 \\
& \quad \quad \text{post}(\text{COPY}(s_1, s_2, GP)) \ \} \\
\text{RECEIVE}(\text{COPY}(s_1, s_2, GP)) & \quad \text{if } s_1 \neq s_2 \\
& \quad \{ \ \text{if } \text{recvLT}(s_2, GP) \ \} \\
& \quad \quad \{ \ \text{post}(\text{DEC}(s_2, s_1, GP)) \ \} \\
& \quad \text{else} \\
& \quad \quad \{ \ \text{sendLT}(s_2, GP) = \text{true} \\
& \quad \quad \quad \text{post}(\text{INC}, \text{DEC}(s_2, \text{owner}(GP), GP, s_3)) \ \text{if } s_1 \neq \text{owner}(GP) \ \} \ \} \\
\text{RECEIVE}(\text{INC}, \text{DEC}(s_1, s_2, GP, s_3)) & \\
& \quad \{ \ \text{sendLT}(s_2, GP) = \text{sendLT}(s_2, GP) + 1 \\
& \quad \quad \text{post}(\text{DEC}(s_2, s_3, GP)) \ \} \\
\text{RECEIVE}(\text{DEC}(s_1, s_2, GP)) & \\
& \quad \{ \ \text{sendLT}(s_2, GP) = \text{sendLT}(s_2, GP) - 1 \ \} \\
\text{DELETE}(s, GP) & \quad \text{if } \text{sendLT}(s, GP) = 0, \ \text{recvLT}(s, GP), \text{owner}(GP) \neq s \\
& \quad \{ \ \text{recvLT}(s, GP) = \text{false} \\
& \quad \quad \text{post}(\text{DEC}(s, \text{owner}(GP), GP)) \ \} \ \}
\end{align*}
\]

**Figure 9:** Abstract Garbage Collection Algorithm
5 Correctness

In this Section, we establish the correctness of our distributed garbage collector. Correctness of a distributed garbage collector has two different aspects. A garbage collector has the safety property if an object is never reclaimed when remote references are still accessible. It has the liveliness property if unreachable objects are eventually reclaimed. Let us first study the first property. It is derived from three invariants that we state in the following Lemma.

We are interested in proving the correctness of the algorithm for each GP. We therefore introduce an operator that allows us to select messages "related" to a GP.

Definition 1 Let \( D \equiv \langle S, G, \text{send}L, \text{rec}L, \text{M} \rangle \) be a configuration of the DGC-machine. Let \( G \) be a global pointer of \( G \). The set of messages related to \( G \), written \( M \downarrow G \), is defined as:

\[
\{ m \in M \mid
\begin{align*}
    m &= \text{COPY}(s_i, s_j, GP), \\
    m &= \text{DEC}(s_i, s_j, GP), \quad \text{or} \\
    m &= \text{INC} \cdot \text{DEC}(s_i, s_j, GP, s_k),
\end{align*}
\]

for any \( s_i, s_j, s_k \).

\[ \Box \]

For a given GP, the first invariant expresses the sum of counter values in send tables as a function of the receive table values and the weight associated with messages. By definition, \( \text{DEC} \) and \( \text{COPY} \) messages have a unitary weight, while \( \text{INC} \cdot \text{DEC} \) messages have a null weight.

Lemma 2 Let \( D \equiv \langle S, G, \text{send}L, \text{rec}L, \text{M} \rangle \) be a legal distributed system. We have the following property. For any \( GP \in G \):

\[
\sum_{s_i \in S} \text{send}L(s_i, GP) = \sum_{s_i \in S} \text{INT}(\text{rec}L(s_i, GP)) - 1 + \sum_{m \in M \downarrow GP} \text{Weight}(m),
\]

with

\[
\text{Weight}(\text{DEC}(s_i, s_j, GP)) = 1, \\
\text{Weight}(\text{COPY}(s_i, s_j, GP)) = 1, \\
\text{Weight}(\text{INC} \cdot \text{DEC}(s_i, s_j, GP)) = 0
\]

\[
\text{INT(true)} = 1, \\
\text{INT(false)} = 0.
\]

\[ \Box \]

Proof of Lemma 2

The invariant is initially true for any initial configuration. We then prove that each possible transition of the machine preserves it. Details may be obtained from [19].

Before the next Lemma, we need to define an operator that allows us to select messages that act (or have acted) upon counters maintained by a site \( s_i \), which we call messages under control of \( s_i \).

Definition 3 Let \( D \equiv \langle S, G, \text{send}L, \text{rec}L, \text{M} \rangle \) be a configuration of the DGC-machine. Let \( s_i \) be a site of \( S \). The set of messages under control of \( s_i \), written \( M \downarrow s_i \), is defined as:

\[
\{ m \in M \mid
\begin{align*}
    m &= \text{COPY}(s_i, s_j, GP), \\
    m &= \text{DEC}(s_i, s_j, GP), \quad \text{or} \\
    m &= \text{INC} \cdot \text{DEC}(s_i, s_j, GP, s_k), \\
    \forall m \in M \downarrow (s_i, GP).
\end{align*}
\]

\[ \Box \]

Messages that relate to \( G \) and under control of \( s_i \) written \( M \downarrow (s_i, GP) \) are defined as the intersection of \( M \downarrow s_i \) and \( M \downarrow (GP) \).

Lemma 2 gives the value of the sum of counters for a given \( G \). The next Lemma states that the value of a counter on a site \( s_i \) that is not the GP owner is equal to the number of messages related to \( G \) and under control of \( s_i \).

Lemma 4 Let \( D \equiv \langle S, G, \text{send}L, \text{rec}L, \text{M} \rangle \) be a legal distributed system. We have the following property:

\[
\forall GP \in G, \forall s_i \in S \text{ such that } s_i \neq \text{owner}(GP), \\
\text{send}L(s_i, GP) = \#(M \downarrow s_i, GP).
\]

\[ \Box \]

Proof of Lemma 4

The equality is initially true and is preserved by each transition. The case analysis is available from [19].

By combining the results of Lemmas 2 and 4, we can determine the value of the counter in the owner send-table. The value of the owner reference counter for a given \( G \) is the number of true entries in the receive-tables plus the number of messages \( \text{COPY} \) and \( \text{DEC} \) in transit minus the number of \( \text{INC} \cdot \text{DEC} \) messages aimed at the owner.

Lemma 5 Let \( D \equiv \langle S, G, \text{send}L, \text{rec}L, \text{M} \rangle \) be a legal distributed system. The following property holds:

\[
\forall GP \in G, \text{ such that } s = \text{owner}(GP), \\
\text{send}L(s, GP) = \sum_{s_i \in S} \text{INT}(\text{rec}L(s_i, GP)) - 1 + \sum_{m \in M \downarrow (s, GP)} \text{Weight}(m) - \#(m \mid m = \text{INC} \cdot \text{DEC}(s_j, s, GP, s_k), \\
\forall s_i, s_j, s_k).
\]

\[ \Box \]

Proof of Lemma 5

The result is obtained by rewriting Lemma 4 into Lemma 2, and simplifying the equality.

The next Theorem establishes the safety of the algorithm. If \( G \) is accessible on a site different from its owner, i.e., if a receive table has an entry for a given \( G \), then \( G \) is present in the owner send-table; therefore, cannot be reclaimed on the owner.

Theorem 6 (Safety) Let \( D \equiv \langle S, G, \text{send}L, \text{rec}L, \text{M} \rangle \) be a configuration. The following statement holds:

\[
\forall GP \in G, \text{let } s = \text{owner}(GP), \forall s_i \neq s, \\
\text{if } \text{rec}L(s_i, GP), \text{ then } \text{send}L(s_i, GP) > 0.
\]

\[ \Box \]
Proof of Theorem 8
Lemma 5 defines the value of the send-table on the owner of a GP, which can be rewritten as follows:

$$sendT(s, GP) = \sum_{s_j \in S} X_i - Y_i,$$

with

$$X_i = INT(recT(s_i, GP)) +$$

$$\#\{m | m = DEC(s_i, s_j GP)\}$$

or

$$m = COPY(s_i, s_j, GP)\},$$

$$Y_i = \#\{m \mid m = INC_DEC(s_i, s_j GP, s_j),$$

$$\forall s_j{\in}\},$$

A case analysis on the different transitions allows us to conclude that $$X_i - Y_i \geq 0.$$ The interesting case concerns a transition receive(DEC(s_i, s_j, GP)), where the difference after transition is smaller than before transition. It however remains positive by the side-condition on the receive transition. We can therefore conclude that:$$sendT(s, GP) \geq 0.$$ If there is a $$s_i$$ such that recT(s_i, GP), then there exists at least one site s_j such that $$X_j - Y_j > 0,$$ which guarantees that $$sendT(s, GP) > 0.$$ We proceed as above. Let us assume that $$X_j - Y_j = 0$$ for every $$s_j.$$ Therefore, for every $$s_j$$ such that recT(s_i, GP), there is $$s_{0}$$ and at least one message INC_DEC(s_j, s_i, GP, s_i), so that $$Y_j = X_j$$ (cf. Lemma 5). This means that $$sendT(s, GP) > 0,$$ which implies recT(s_i, GP). Let us examine the sequence of sites $$s_j, s_{i}, \ldots.$$ The sequence is either finite or infinite.

- If it is finite, let $$s_n$$ be the last site such that recT(s_i, GP) = true. By the same reasoning, there is a following site $$s_{n+1}$$ in the sequence, which contradicts the fact that $$s_n$$ was the last.

- If the sequence is infinite, there must be a loop as we have only a finite number of sites S. Let $$s_t$$ be the first site of the sequence occurring twice: $$s_{t_0}, s_{t_1}, s_{t_2}, \ldots.$$ There are INC_DEC messages only produced by MAKE_COPY(s_i, s_{t+1}, GP) transitions if recT(s_{t+1}, GP). As these transitions are performed atomically, such a sequence cannot exist, because a transition (make-copy) should have posted a DEC message instead of an INC_DEC message.

□

In order to prove the GC liveliness, we first assume that control messages are eventually processed. The liveliness property follows from previous Lemmas. The next two Lemmas define the value of reference counters when control messages related to a given global pointer are processed. First, Lemma 7 establishes that reference counters in send-tables of sites different from the owner become zero when control messages are processed.

**Lemma 7** Let D = $$(S, G, sendT, recT, M)$$ be a legal distributed system. For any GP $$\in G$$, for any $$s_i \in S$$ such that $$s_i \neq owner(GP),$$ If there is no message related to GP, then: $$sendT(s_i, GP) = 0.$$ □

**Proof of Lemma 7**

The proof is a consequence of Lemma 4, with $$M \downarrow s_i(GP) = \emptyset.$$ □

Lemma 7 ensures that as control messages are processed, global pointers will not remain accessible from local roots longer than needed by the user's program. Lemma 8 defines the value of the owner's reference counter after processing control messages.

**Lemma 8** Let $$(S, G, sendT, recT, M)$$ be a DGC-configuration. Let GP $$\in G,$$ such that no message related to GP is in transit. If $$s = owner(GP),$$ then:

$$sendT(s, GP) = \sum_{s_j \in S} INT(recT(s_i, GP)).$$

□

**Proof of Lemma 8**

The proof is a consequence of Lemma 5, with $$M \downarrow s_i(GP) = \emptyset.$$ □

Next, we assume that local collectors also enjoy the liveliness property, i.e., the ability to reclaim unreachable objects. We can conclude that if GP is unreachable on all machines but the owner, the value of the reference counter on the owner will become zero, which in turn guarantees that GP and its associated object may be reclaimed if unreachable on the owner.

**Theorem 9 (GC Liveness)** Let GP be a global pointer inaccessible on sites different from its owner, such that no message related to GP is in transit. If $$s = owner(GP),$$ then $$sendT(s, GP) = 0.$$ □

**Proof of Theorem 9**

Immediately from Lemmas 7 and 8. □

These results also show that the effect of reorganising the diffusion tree is to flatten it so that all sites that have access to a global pointer become direct children of its owner.

6 Implementation Issues

This section is concerned with some implementation aspects of the distributed garbage collector. First, we describe the different modules of our implementation, then we present a number of optimisations, and finally, some specific details are covered.

6.1 Implementation Sketch

The distributed garbage collector relies on four modules:

1. **send-table module.** The send-table is a hash-table and, for our purposes, is interfaced by two functions. The function incRefCounter expects a GP and increments its reference counter in the send-table if GP is not present before the call, it is given an initial counter value 1. The function decRefCounter expects a GP and decrements its associated reference counter; GP is removed from the table if its value reaches 0.

2. **receive-table module.** The receive-table is also a hash-table that is manipulated by three functions. The functions putInReceiveTable! and removeFromReceiveTable! both expect a GP and respectively add it to or remove it from the table. Finally, the predicate isInReceiveTable? indicates whether a global pointer belongs to the table.
3. The serialisation-deserialisation module. It is composed of two functions notify-send-gp! and notify-receive-gp! which are called when a global pointer is respectively serialised or deserialised during an RSR. When a global pointer is sent, its reference counter is incremented. When a global pointer is received, the function notify-receive-gp! is called and returns a global pointer to be used in the RSR handler. If the GP has already been received, i.e. it is already in the entry table, a decrement message is sent to the emitter, and the copy of GP in the receive-table is returned. Otherwise, if it is the first reception of GP, it is entered in the receive-table and an increment-decrement message is sent to its owner, provided that the emitter is not the owner.

Tables must be accessed in a critical section as several RSRs may be received and handled in parallel. When receiving an RSR, it is not the appropriate time to send a new GC control messages; handling the request has a higher priority. As a result, requests to send control messages are simply queued by functions belonging to the message module (see next item).

4. The message module takes care of handling and sending control messages. When a decrement message is received for a given GP, the function decrement-reference-count! is called on GP. When an increment-decrement message is received for a GP and a site s, the function increment-reference-count! is called on GP, and a decrement message is enqueued for the GP, aimed at site s. Order of messages between two sites must be preserved and increment-decrement messages should always be sent before decrement messages if they concern the same GP.

6.2 Optimisations
Section 4 described the abstract algorithm used for our garbage collector. A number of optimisations can be implemented to reduce the cost of garbage collection. All optimisations are based on the property that control messages may be delayed because they are not part of the mandatory computation. However, there clearly needs to be a balance between garbage collection activities and mandatory computations, because delaying control messages for a long time may increase the memory requirement of the application too much.

By grouping messages by destination, we can substantially reduce control message traffic. For instance, a decrement message to a given host contains a variable number of global pointers as well as the amount by which their counter should be decremented. In our implementation, message frequency may be controlled by variables specifying the minimum and maximum number of counter updates in every control message.

Similarly, an increment-decrement message concerning a GP and a site s may be merged with a decrement message concerning GP and aimed at s. The increment-decrement message should contain the amount by which the counter should be decremented on s; when received by the owner, it increments GP counter, and then sends a decrement message with the given amount.

When a GP becomes inaccessible, a decrement message is sent to its owner. The algorithm can be further optimised at that time if the increment-decrement message was not yet sent to its owner (and to be followed by a decrement message to a third site s). The decrement message and the increment-decrement message to the owner cancelled each other and can be replaced by the decrement message to s.

(Note that by delaying increment-decrement messages, our garbage collector behaves very similarly as indirect reference counting.)

We can easily see that these three optimisations preserve the safety and liveness of the algorithm.

6.3 Implementation Details
The receive table contains all global pointers owned or received by a site. The receive table should be designed carefully. If a global pointer entered in a receive table remains accessible to the local garbage collector from the root set, it will never be collected, nor the object on the owner host. Therefore, entries of global pointers in a receive table should be marked so that their inaccessibility can be detected. Once such a global pointer becomes inaccessible, it must also be removed from the receive table. Inaccessibility is detected after a local collection by installing finalizers [11] on global pointers. A finalizer is a procedure called by the local garbage collector on an object once it is detected to be inaccessible. In Nerve, such finalizers remove global pointers from the receive table and prepare a decrement message to the global pointer owner. The message itself cannot be sent at garbage-collection time because such an operation requires memory not necessarily available at that moment: instead, inaccessible global pointers are queued (without allocation) by finalizers, and messages are sent to their destination sites, only after the end of the garbage collection. The send tables, as opposed to the receive tables, are a root of the local garbage collector.

The distributed garbage collector was implemented in Scheme. However, nothing prevents us to implement it in C. As a result, Nexus, Boehm and Weiser's collector, and a C library for distributed garbage collection could all be packaged as a language-independent message-passing library, with a distributed garbage collector.

7 Mobility
Nerve global pointers refer to immobile objects, but mobile objects can be implemented using global pointers. For instance, mobile ports [8] feature mobile receiving and sending ends. In this section, we describe another approach able to deliver method calls to mobile objects.

We define a mobile object MO as a record composed of five fields (Figure 10): (i) the object content obj, (ii) information about the object position fpos, (iii) a mobility counter count, (iv) a message queue q, and (v) a lock to guarantee mutual exclusion when accessing the object. (The semantics of locks and their primitives lock, unlock is defined in [20].) Such mobile objects may be referred to remotely by global pointers (a, s), denoting a mobile object at address a on site s.

We support two actions on mobile objects: remote service requests, i.e., method invocation, and migration. Our goal is to ensure that methods will eventually be invoked even though objects migrate; this requires us to forward messages to mobile objects.

Note RSR messages to global pointers GP1 and GP2, handler name and other arguments denotes the invocation of the method name on a mobile object represented by GP2. When the user issues a RSR message, GP1 must be set to nil;
\[ m \quad ::= \quad \text{RSR}(GP_3,GP_2,\text{name},\ldots) \mid \text{UPDATE}(GP_3,GP_2,GP_3,\text{count}) \quad \text{(Message)} \\
\quad f \in \mathcal{F} \quad ::= \quad \text{nil} \mid \text{fed} \mid \text{GP} \quad \text{(Forwarding Field Value)} \\
\quad \sigma \in \text{Store} \quad ::= \quad S \times \text{Addr} \rightarrow \text{MO} \quad \text{(Store)} \\
\quad \text{MO} \quad ::= \quad \{ \\
\quad \quad \text{obj} : \text{any}, \quad \text{(Data Field)} \\
\quad \quad \text{fed} : \mathcal{F}, \quad \text{(Forward Field)} \\
\quad \quad \text{count} : \mathbb{N}, \quad \text{(Mobility Count Field)} \\
\quad \quad q : m^*, \quad \text{(Queue Field)} \\
\quad \quad \text{lock} : \mathcal{L} \} \quad \text{(Lock Field)} \]

\begin{align*}
\text{RECEIVE(} & \text{RSR}(GP_1,\langle \omega_2, \omega_2 \rangle, \text{handler}, \ldots)) \quad \{ \\
\quad & \text{lock}(\sigma(\omega_2, \omega_2).\text{lock}); \\
\quad & \text{if} \quad \sigma(\omega_2, \omega_2).\text{fed} = \text{nil} \quad \text{then} \quad \{ \\
\quad & \quad \text{unlock}(\sigma(\omega_2, \omega_2).\text{lock}); \\
\quad & \quad \text{handle-\text{msg}}(\text{handler}, \sigma(\omega_2, \omega_2).\text{obj}, \ldots) \} \\
\quad & \text{elseif} \quad \sigma(\omega_2, \omega_2).\text{fed} \in \mathcal{F} \quad \text{then} \quad \{ \\
\quad & \quad \text{unlock}(\sigma(\omega_2, \omega_2).\text{lock}); \\
\quad & \quad \text{if} \quad GP_1 \neq \text{nil} \quad \text{then} \\
\quad & \quad \quad \text{post}(\text{UPDATE}(\omega_2, \omega_2, GP_1, \sigma(\omega_2, \omega_2).\text{fed}, \sigma(\omega_2, \omega_2).\text{count})); \quad \text{/* update the emitter */} \\
\quad & \quad \quad \text{post}(\text{RSR}(\omega_2, \omega_2, \sigma(\omega_2, \omega_2).\text{fed}, \text{handler}, \ldots)) \} \quad \text{/* forward the request */} \\
\quad & \text{else} \quad \{ \sigma(\omega_2, \omega_2).q = \sigma(\omega_2, \omega_2).q \uplus (\text{RSR}(GP_1, \langle \omega_2, \omega_2 \rangle, \text{handler}, \ldots)); \quad \text{/* enqueue the request */} \\
\quad & \quad \quad \text{unlock}(\sigma(\omega_2, \omega_2).\text{lock}) \} \} \}
\end{align*}

\begin{align*}
\text{MIGRATE} & (\langle \omega_1, \omega_1 \rangle, \langle \omega_2, \omega_2 \rangle) \quad \text{RECEIVE(} & \text{UPDATE}(\langle \omega_1, \omega_1 \rangle, (\omega_2, \omega_2), \text{GP}, \text{count})) \quad \{ \\
\quad & \text{lock}(\sigma(\omega_1, \omega_1).\text{lock}); \\
\quad & \text{if} \quad \sigma(\omega_1, \omega_1).\text{fed} = \text{nil} \quad \text{then} \quad \{ \\
\quad & \quad \text{raise} \quad \text{object has already moved!} \\
\quad & \quad \quad \text{else} \quad \{ \sigma(\omega_1, \omega_1).\text{fed} := \text{fed}; \\
\quad & \quad \quad \quad \sigma(\omega_1, \omega_1).\text{count} := \sigma(\omega_1, \omega_1).\text{count} + 1; \\
\quad & \quad \quad \quad \text{unlock}(\sigma(\omega_1, \omega_1).\text{lock}); \\
\quad & \quad \quad \quad \text{copy-\text{obj}}((\omega_1, \omega_1), (\omega_2, \omega_2)); \quad \text{/* returns when object is copied*/} \\
\quad & \quad \quad \quad \text{lock}(\sigma(\omega_1, \omega_1).\text{lock}); \\
\quad & \quad \quad \quad \sigma(\omega_1, \omega_1).\text{fed} := (\omega_2, \omega_2); \\
\quad & \quad \quad \quad \text{unlock}(\sigma(\omega_1, \omega_1).\text{lock}); \\
\quad & \quad \quad \quad \text{forward-\text{queue}}(\text{messages}(\sigma(\omega_1, \omega_1).q, \langle \omega_2, \omega_2 \rangle)) \} \} \}
\end{align*}

Figure 10: Object Migration

as the request is forwarded, GP_3 denotes the latest forwarder met.

When an RSR message is received, the object lock is acquired in order to maintain local consistency. If its fed field is nil, the mobile object is local, and the method is called locally (with handle-\text{msg}). Otherwise, if the fed field contains a global pointer GP, the object acts as a forwarder to the next position denoted by GP.

Migration of a mobile object (\omega_2, \omega_1) to a new position (\omega_3, \omega_2) is handled by the function MIGRATE. Migration consists of increasing the mobility counter, copying the object content to the new position, and configuring the previous position as a forwarder. In this implementation, we release the object lock as soon as possible, which requires us to temporarily set the fed field to the value fed; its role is to enqueue all incoming requests until the mobile object has reached its new position.

Mobile objects create chains of forwarding pointers as they migrate. In order to make mobile objects independent of the sites they visited and to reduce the cost of remote method invocation, chains of forwarding pointers must be reduced [26]. We proceed lazily and update forwarders as they are used. The message UPDATE ensures that more recent positions only are stored in the fed field.

Using the following Lemma, one can guarantee that every message will eventually be delivered, provided that the object does not migrate faster than message delivery.

\textbf{Lemma 10 (Mobility)} User messages are forwarded towards objects with a higher mobility counter. \(\square\)

\textbf{Proof of Lemma 10}\n
Each mobile object is associated with a fed field and a mobility counter. We can see that they are always updated at the same time (in MIGRATE and UPDATE). In both cases, the mobility counter is strictly increasing. \(\square\)

Our migration facility is independent of the distributed garbage collector; the diffusion tree reorganisation mechanism is orthogonal to the short-cutting of forwarder chains. We believe that such a modular design facilitates the understanding and implementation of algorithms.

There are many alternative designs to mobile objects. One could consider eager position updates, in-order message...
delivery or even causal delivery. Choosing between them is application dependent. We believe that this argues in favour of a library for mobility on top of a distributed garbage collector.

8 Discussion and Related Work

Reference-counting garbage collection was initially developed for uniprocessor systems [4]. It is extended to distributed environments by introducing two types of messages. A decrement message is sent to GP's owner when GP is discarded; an increment message is sent to GP's owner when GP is duplicated. However, this naive extension fails to behave properly when messages are not causally ordered [14].

Numerous solutions to this problem have been proposed. The most famous are weighted reference counting [1, 31, 6] and its optimised version [5], or generational reference counting [9]. However, Lerman and Maurer's [17, 29] solution is closest to our work. They also rely on message ordering between any pair of processors. When a GP is duplicated, a create message is sent to its owner. The owner then sends an acknowledgement to GP's receiver. When a GP is discarded, a decrement message is sent only after the acknowledgement has been received for this pointer. Lerman and Maurer's technique also involves three sites (emitter, receiver, and owner), but it differs from ours: (i) the owner is involved every time the emitter duplicates a GP to the receiver in Lerman and Maurer's algorithm, whereas it is involved the first time in our algorithm. (ii) Lerman and Maurer's schema requires the receiver to maintain a count of both the number of copies made and the number of acknowledgements received. Decrement messages can only be sent when both are equal.

Indirect reference counting, initially defined by Fiquet [22], consists of reference counters distributed along the diffusion tree. It avoids the message conflict that exists in reference counting by using a decrement message only. Indirect reference counting creates zombie pointers [23]; these pointers are no longer used, but cannot be reclaimed because their associated counter is still positive as they are active in the children of the diffusion tree. Our algorithm reinstates a special increment-decrement message, which avoids race conditions, but is able to reorganise the diffusion tree, essentially deleting zombie pointers.

The distributed variant of the Train GC [13] uses an algorithm to track pointers, which essentially is a reference counting mechanism. Like Birrel et al., the emitter always informs the owner of a pointer that a copy of the pointer is sent to another site.

Reference listing [24] is a variant of distributed reference counting. In this approach, send tables associate GPs with list of destination sites instead of a counter. Here, we discuss two variants by Birrel et al. [3] and Shapiro et al. [26, 24].

Birrel et al. [3] present network objects, a distributed object-based language with a garbage collector. The owner of an object maintains a "dirty" set, which contains identifiers for all the processes that have GPs to the object. When a client first receives a GP, it makes a dirty call to the owner. When the GP is no longer reachable, as determined by the client's local gc, the client makes a clean call and deletes GP. With the dirty calls, Birrel et al. reinstate the equivalent of an increment message. In order to avoid conflicts between dirty and clean calls, an acknowledgement message from the receiver of a GP to its emitter guarantees the impossibility to free the pointer on the emitter. Similarly as Lerman and Maurer's algorithm, Birrel's mechanism involves the object owner for every duplication of a reference. Our mechanism is more flexible as, fully lazy, it behaves as indirect reference counting, and fully eager it behaves more like Birrel's; the only difference is that our acknowledgement is sent by the owner in the form of a decrement message and not by the recipient of the reference.

Shapiro, Dickman, and Plainis [26, 24] present a fault-tolerant distributed garbage collector based on reference listing and supporting mobile objects. They introduce the notion of SSP chains. A chain starts its existence by a single SSP (Scion/Stub pair); it increases when sending the reference of a local object, or when migrating an object to some other site. In addition, they propose a technique to short-cuts SSP-chains, thereby avoiding the equivalent of zombie references. Shapiro, Dickman, and Plainis [26] and Fiquet [23] regard migration as a primitive action that must be supported by their garbage collector. Our solution is to regard migration as a library functionality, which relies on our garbage collector for immobile objects. Therefore, we have three different layers: reliable protocol provided by Nexus, distributed garbage collection, and a library for mobile objects.

JAVA Remote Method Invocation comes with a distributed garbage collector [28]. It extends Birrel's reference listing technique with a new approach to fault tolerance, where remote pointers are leased for a period of time. Sites having pointer copies must regularly renew their leases. Our approach can be extended without problem to reference listing so that send-tables contain the sites to which GPs were sent, and a similar lease technique could also be adopted.

The simplicity and portability of our solution is unfortunately counter-balanced by its inability to collect distributed cycles. As a result, it is the programmer's responsibility to avoid distributed cycles or to explicitly break them for collection [3]. Le Fessant, Piumarta, and Shapiro [16] present an extension to reference counting based on timestamp, which is able to deal with distributed cycles. Lang, Quine, and Fiquet [15] are also able to collect cycles that are distributed over a group of sites. The distributed variant of the Train GC [13] is also able to collect cycles. It combines a reference-counting style pointer-tracking mechanism with a substitution protocol, in order to extend the train GC [12] to a distributed environment. As such, it cannot be used in our context because it requires copying objects and we need to be conservative since we interface with Nexus. However, an interesting question is to decide whether their tracking mechanism could be based on our reference counting algorithm.

9 Conclusion

We have presented a new distributed garbage collection algorithm, which is an essential component of NeXene, a distributed implementation of Scheme. This algorithm uses a novel technique to short-cut diffusion trees, is able to deal with mobile objects, and has a modular design. We are now adopting our implementation to the reference listing technique and are implementing a hierarchical domain organisation, which would make it applicable to the Internet.
10 Acknowledgement

This research was supported in part by the Engineering and Physical Sciences Research Council, grant GR/K30773. The Southampton DSSE seminars group: Hans Boehm, David DeRoeck, Ian Foster, Carl Hauser, Carl Kesselman, Andreas Kind, Christian Queinnec, Mario Germain, and Steve Tuecke gave very useful comments and advice during the implementation of NeXyme and its garbage collector. Thanks to the anonymous referees for their comments.

References


