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 those 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 NeXeme [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-passing 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.

NeXeme [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 Mattern [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.

ICFP'98 09/98, 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 *o*-reference [17].

Practically, in NeXeme, an RSR is specified by providing a global pointer, a handler identifier, and some arguments. Issuing an RSR causes the arguments to be transfered 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 to the RSR 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 RSRs 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 s_1 and a GP pointing at this data. The purpose of a distributed garbage collector is to ensure that the data on s_1 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 RSRs 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 upto 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 s_1 to s_2; GP is entered in the send-table of s_1 with the counter value 1 and in the receive-table of s_2.

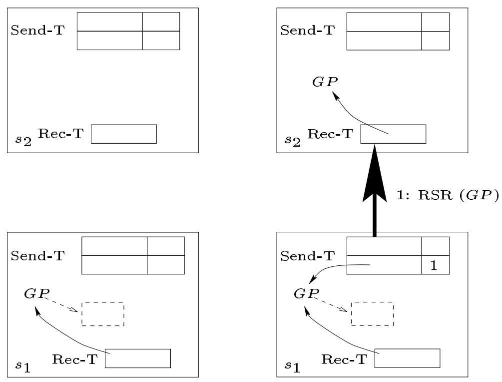

Figure 1: GP sent from s_1 to s_2

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 s_2, a decrement message is sent to s_1. GP is removed from the send-table of s_1, and the space can then be reclaimed on s_1 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. Now, 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 [6].

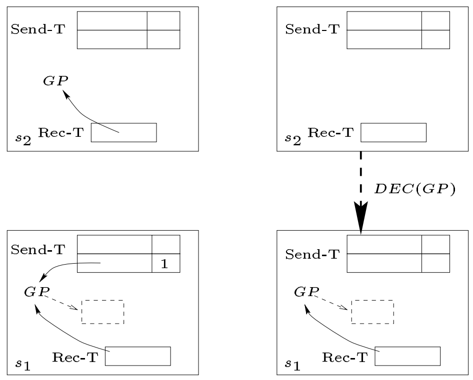

Figure 2: GP freed on s_2

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 s_1 to s_2. Using the same principle, Figure 3 presents the setting after GP is passed from s_2 to s_3: send-tables in s_1 and s_2 contain entries for GP, which also appears in receive-tables in s_2 and s_3.

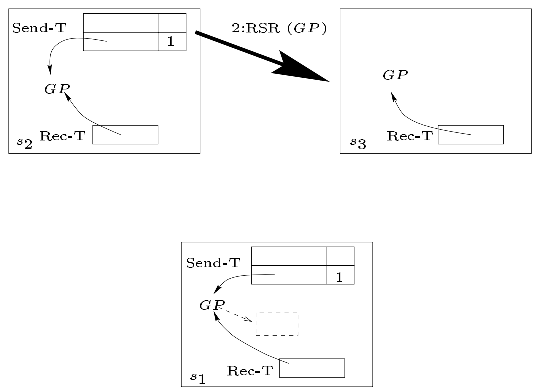

Figure 3: Indirect Counters Along the Diffusion Tree

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 s_3 becomes garbage, s_3 sends a decrement message to s_1, that is, the GP's owner. The effect of the decrement message is to remove GP from the send-table of s_1, and possibly to reclaim its space if no longer reachable. This clearly results in an inconsistent situation as GP may still be active on s_2.

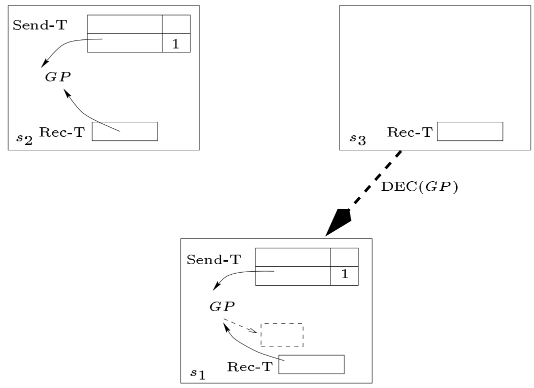

Figure 4: Untimely Decrement Message

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 s_2 (as well as its entry in the send-table) cannot be reclaimed even if GP is no longer needed on s_2.

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: s_1, s_2, s_3, 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 s_1 receives an increment-decrement message, it increments GP's reference counter, and then sends a decrement message to s_2 concerning GP.

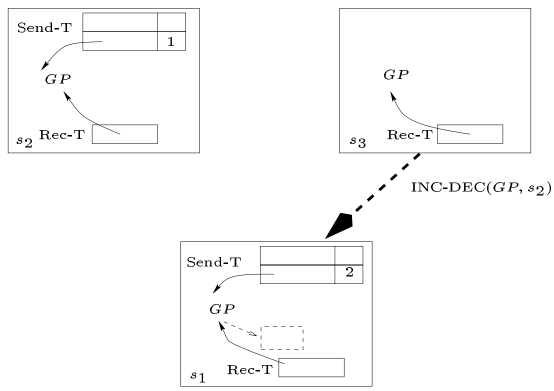

Figure 5: Increment-Decrement Message

Let us go back to Figure 3, where s_2 sent to s_3 a GP owned by s_1. If s_3 receives GP for the first time, s_3 sends an increment-decrement message to s_1 concerning GP and s_2. The result is to increment GP's counter on s_1 (Figure 5) and to send a decrement message from s_1 to s_2 (Figure 6). The effect is to prune the diffusion subtree rooted at s_3, and to graft it to the root s_1.

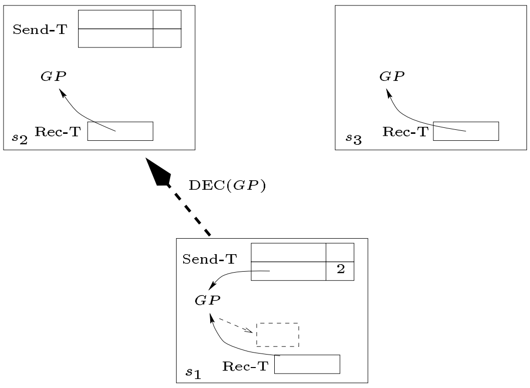

Figure 6: Decrement Message

Introducing the increment-decrement message is not sufficient to avoid untimely message arrivals. The increment-decrement message from s_3 should be received by s_1 before any decrement message from s_3 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 s_2, 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 are concerned with 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 $GP = \langle \alpha, s \rangle$ 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 schema: $NAME(emitter, receiver, arguments, \ldots)$, where the emitter and receiver are sites, and arguments are message-dependent. Send and Receive Tables are represented by functions associating 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 queued to be sent, messages that are being transferred, and messages that are received but not handled yet.

The initial configuration $\mathcal{D}_i$ is defined as follows:

$$\mathcal{D}_i = \langle \mathcal{S}, \mathcal{G}, send\_T_i, rec\_T_i, \emptyset \rangle,$$

where the initial send table $send\_T_i = \lambda s\ GP.0$ contains zero for every entry, and the initial receive table $rec\_T_i = \lambda s\ GP.owner(GP) = s$ 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 $\mathcal{D}_1$ into a configuration $\mathcal{D}_2$, using configuration transformers defined in Figure 9; these transformers are an abstract 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 s_1 and a receiver s_2. Messages in the pool of messages can be handled by a (*receive*) transition. Finally, local collectors call a finalization method [11] 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 s is deleted. There is a side-condition of (*delete*) by which the owner is prevented to send such DEC messages.

In Figure 9, we use the following conventions. Let $\langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$ be a configuration :

- $send\_T(s, GP) := V$ denotes $\langle \mathcal{S}, \mathcal{G}, send\_T', rec\_T, \mathcal{M} \rangle$, such that $send\_T'(s, GP) = V$ and $send\_T'(s, GP') = send\_T(s, GP'), \forall GP' \neq GP$.
- $rec\_T(s, GP) := V$ is similar.
- $post(m)$ denotes $\langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \cup \{m\} \rangle$, 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.

$$
\begin{array}{rcll}
s \in \mathcal{S} & = & \{s_0, s_1, \ldots, s_n\} & \text{(Set of Sites)} \\
GP \in \mathcal{G} & ::= & \langle \alpha, s \rangle & \text{(Set of Global Pointers)} \\
\alpha \in Addr & = & \{\alpha_0, \ldots\} & \text{(Address)} \\
m \in Msg & ::= & DEC(s_1, s_2, GP) \ \mid \ INC\_DEC(s_1, s_2, GP, s_3) \ \mid \ COPY(s_1, s_2, GP) & \text{(Message)} \\
\mathcal{M} & : & BagOf(Msg) & \text{(Pool of Messages)} \\
send\_T & : & \mathcal{S} \times \mathcal{G} \to \mathbb{N} & \text{(Send Tables)} \\
rec\_T & : & \mathcal{S} \times \mathcal{G} \to Bool & \text{(Receive Tables)} \\
\mathcal{D} \in Config & ::= & \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle & \text{(DGC-Configuration)}
\end{array}
$$

Figure 7: State Space of the DGC-machine

$$
\begin{array}{lll}
\mathcal{D}_1 & \Rightarrow & MAKE\_COPY(s_1, s_2, GP) \Rightarrow \mathcal{D}_2 \hfill (\mathit{make\text{-}copy}) \\
& & \quad \text{if } \ s_1, s_2 \in \mathcal{S}, \ GP \in \mathcal{G}, \ \text{with } \ \mathcal{D}_1 = \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle \\
\mathcal{D}_1 & \Rightarrow & RECEIVE(m) \Rightarrow \mathcal{D}_2 \hfill (\mathit{receive}) \\
& & \quad \text{if } \ m \in \mathcal{M}, \ \text{with } \ \mathcal{D}_1 = \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle \\
& & \quad \wedge \ \text{if } \ m = DEC(s_1, s_2, GP), \ \text{then } INC\_DEC(s_1, s_2, GP, s_3) \notin \mathcal{M}, \forall s_3 \\
\mathcal{D}_1 & \Rightarrow & DELETE(s, GP) \Rightarrow \mathcal{D}_2 \hfill (\mathit{delete}) \\
& & \quad \text{if } \ s \in \mathcal{S}, \ GP \in \mathcal{G}, \ \text{with } \ \mathcal{D}_1 = \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle
\end{array}
$$

Figure 8: Transitions of the DGC-Machine

$MAKE\_COPY(s_1, s_2, GP) \quad$ if $\quad s_1 \neq s_2 \ \wedge \ (s_1 = owner(GP) \ \vee \ rec\_T(s_1, GP))$
$\{ \quad send\_T(s_1, GP) := send\_T(s_1, GP) + 1$
$\quad post(COPY(s_1, s_2, GP)) \ \}$

$RECEIVE(COPY(s_1, s_2, GP)) \quad$ if $\quad s_1 \neq s_2$
$\{ \quad$ if $rec\_T(s_2, GP)$ then
$\quad \{ \ post(DEC(s_2, s_1, GP)) \ \}$
else
$\quad \{rec\_T(s_2, GP) := true$
$\quad post(INC\_DEC(s_2, owner(GP), GP, s_1))$ if $s_1 \neq owner(GP) \ \} \ \}$

$RECEIVE(INC\_DEC(s_1, s_2, GP, s_3))$
$\{ \quad send\_T(s_2, GP) := send\_T(s_2, GP) + 1$
$\quad post(DEC(s_2, s_3, GP)) \ \}$

$RECEIVE(DEC(s_1, s_2, GP))$
$\{ \quad send\_T(s_2, GP) := send\_T(s_2, GP) - 1 \ \}$

$DELETE(s, GP) \quad$ if $send\_T(s, GP) = 0, \ rec\_T(s, GP), owner(GP) \neq s$
$\{ \quad rec\_T(s, GP) := false$
$\quad post(DEC(s, owner(GP), GP)) \ \}$

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 *liveness* 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 Lemmas.

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* $\mathcal{D} \equiv \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$ *be a configuration of the DGC-machine. Let* GP *be a global pointer of* $\mathcal{G}$*. The set of messages related to* GP*, written* $\mathcal{M} \downarrow GP$*, is defined as*

$$\begin{aligned} \{m \in \mathcal{M} \quad | \quad & m = COPY(s_i, s_j, GP), \\ & m = DEC(s_i, s_j, GP), \quad or \\ & m = INC\_DEC(s_i, s_j, GP, s_k), \\ & for\ any\ s_i, s_j, s_k\}. \end{aligned}$$

□

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, DEC and $COPY$ messages have a unitary weight, while INC_DEC messages have a null weight.

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

$$\begin{aligned} & \sum_{s_i \in \mathcal{S}} send\_T(s_i, GP) \\ & \quad = \sum_{s_i \in \mathcal{S}} INT(rec\_T(s_i, GP)) - 1 \\ & \quad + \sum_{m \in \mathcal{M} \downarrow GP} Weight(m), \end{aligned}$$

with

$$\begin{aligned} Weight(DEC(s_1, s_2, GP)) &= 1 \\ Weight(COPY(s_1, s_2, GP)) &= 1 \\ Weight(INC\_DEC(s_1, s_2, GP)) &= 0 \\ \\ INT(true) &= 1 \\ INT(false) &= 0. \end{aligned}$$

□

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* $\mathcal{D} \equiv \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$ *be a configuration of the DGC-machine. Let* s_i *be a site of* $\mathcal{S}$*. The set of messages* under control of s_i*, written* $\mathcal{M} \downarrow s_i$*, is defined as*

$$\begin{aligned} \{m \in \mathcal{M} \quad | \quad & m = COPY(s_i, s_j, GP), \\ & m = DEC(s_j, s_i, GP), \quad or \\ & m = INC\_DEC(s_k, s_j, GP, s_i), \\ & for\ any\ GP, s_j, s_k\}. \end{aligned}$$

□

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

Lemma 2 gives the value of the sum of counters for a given GP. 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 GP and under control of s_i.

Lemma 4 *Let* $\mathcal{D} \equiv \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$ *be a legal distributed system. We have the following property:*

$$\begin{aligned} & \forall\ GP \in \mathcal{G}, \forall\ s_i \in \mathcal{S} \quad such\ that \quad s_i \neq owner(GP), \\ & \qquad send\_T(s_i, GP) = \#(\mathcal{M} \downarrow (s_i, GP)). \end{aligned}$$

□

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 GP is the number of true entries in the receive-tables plus the number of messages $COPY$ and DEC in transit minus the number of INC_DEC messages aimed at the owner.

Lemma 5 *Let* $\mathcal{D} \equiv \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$ *be a legal distributed system. The following property holds:*

$$\begin{aligned} & \forall\ GP \in \mathcal{G}, \quad such\ that \quad s = owner(GP), \\ & \quad send\_T(s, GP) \\ & \qquad = \sum_{s_i \in \mathcal{S}} INT(rec\_T(s_i, GP)) - 1 \\ & \qquad + \sum_{m \in \mathcal{M} \downarrow (s, GP)} Weight(m) \\ & \qquad - \#(\{m \mid m = INC\_DEC(s_j, s, GP, s_i), \\ & \qquad\qquad \forall s_i, s_j\}). \end{aligned}$$

□

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 GP is accessible on a site different from its owner, i.e. if a receive table has an entry for a given GP, then GP is present in the owner send-table; therefore, cannot be reclaimed on the owner.

Theorem 6 (Safety) *Let* $\langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$ *be a configuration. The following statement holds:*

$$\begin{aligned} & \forall\ GP \in \mathcal{G}, let\ s = owner(GP), \forall s_i \neq s, \\ & \qquad if\ rec\_T(s_i, GP), \quad then \quad send\_T(s, GP) > 0. \end{aligned}$$

□

Proof of Theorem 6
Lemma 5 defines the value of the send-table on the owner of a GP, which can be rewritten as follows:

$$send\_T(s, GP) = \sum_{s_i \in \mathcal{S}} X_i - Y_i,$$

with

$$\begin{aligned} X_i &= INT(rec\_T(s_i, GP)) + \\ &\quad \#(\{m \mid m = DEC(s_i, s, GP) \\ &\quad \text{or } m = COPY(s, s_i, GP)\}), \\ Y_i &= \#(\{m \mid m = INC\_DEC(s_i, s, GP, s_j), \\ &\quad \forall s_j\}). \end{aligned}$$

A case analysis on the different transitions allows us to conclude that $X_i - Y_i \geq 0$. The interesting case concerns a transition $receive(DEC(s_i, s, GP))$, 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: $send\_T(s, GP) \geq 0$.

If there is a s_i such that $rec\_T(s_i, GP)$, then there exists at least one site s_j such that $X_j - Y_j > 0$, which guarantees that $send\_T(s, GP) > 0$.

We proceed ab absurdo. Let us assume that $X_j - Y_j = 0$ for every s_j. Therefore, for every s_j such that $rec\_T(s_j, GP)$, there is s_k and at least one message $INC\_DEC(s_j, s, GP, s_k)$, so that $Y_j = X_j$ (cf. Lemma 5). This means that $send\_T(s_k, GP) > 0$, which implies $rec\_T(s_k, GP)$. Let us examine the sequence of sites: $s_j, s_k, \ldots$. The sequence is either finite or infinite.

- If it is finite, let s_n be the last site such that $rec\_T(s_n, GP) = true$. 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 $\mathcal{S}$. Let s_l the first site of the sequence occurring twice: $\ldots, s_l, \ldots, s_l, \ldots$. Those INC_DEC messages are only produced by $MAKE\_COPY(s_l, s_{l+1}, GP)$ transitions if $\neg rec\_T(s_{n+1}, GP)$. As those 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 liveness, we first assume that control messages are eventually processed. The liveness 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 $\mathcal{D} \equiv \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$ be a legal distributed system. For any $GP \in \mathcal{G}$, for any $s_i \in \mathcal{S}$ such that $s_i \neq owner(GP)$. If there is no message related to GP, then:* $send\_T(s_i, GP) = 0$. □

Proof of Lemma 7
The proof is a consequence of Lemma 4, with $\mathcal{M} \downarrow (s_i, GP) = \emptyset$. □

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 $\langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$ be a DGC-configuration. Let $GP \in \mathcal{G}$, such that no message related to GP is in transit. If $s = owner(GP)$, then :*

$$send\_T(s, GP) = \sum_{s_i \in \mathcal{S}} INT(rec\_T(s_i, GP)).$$

□

Proof of Lemma 8
The proof is a consequence of Lemma 5, with $\mathcal{M} \downarrow GP = \emptyset$.
□

Next, we assume that local collectors also enjoy the liveness 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* $send\_T(s, GP) = 0$. □

Proof of Theorem 9
Immediate 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. The *send-table module.* The send-table is a hash-table and, for our purpose, is interfaced by two functions. The function `increment-reference-count!` 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 `decrement-reference-count!` expects a GP and decrements its associated reference counter; GP is removed from the table, if its value reaches 0.

2. The *receive-table module.* The receive-table is also a hash table that is manipulated by three functions. The functions `put-in-receive-table!` and `remove-from-receive-table!` both expect a GP and respectively add it to or remove it from the table. Finally, the predicate `is-in-receive-table?` 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 enqueued 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 a 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 two 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 an receive table should be *masked* 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 NeXeme, 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-table, as opposed to the receive table, is 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

Nexus 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 fwd, *(iii)* a mobility counter *count*, *(iv)* a message queue q, *(v)* and 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 $\langle \alpha, s \rangle$, denoting a mobile object at address α 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.

The RSR message, with global pointers GP_1 and GP_2, handler *name* and other arguments denotes the invocation of the method *name* on a mobile object represented by GP_2. When the user issues a RSR message, GP_1 must be set to nil;

$$
\begin{array}{rcll}
m & ::= & RSR(GP_1, GP_2, name, \ldots) \mid UPDATE(GP_1, GP_2, GP_3, count) & \text{(Message)} \\
f \in \mathcal{F} & ::= & \mathsf{nil} \mid \mathsf{fwd} \mid GP & \text{(Forwarding Field Value)} \\
\sigma \in Store & : & \mathcal{S} \times Addr \to MO & \text{(Store)} \\
MO & : & \{ & \text{(Mobile Object Record)} \\
 & & \quad obj : any, \quad \text{(Data Field)} & \\
 & & \quad fwd : \mathcal{F}, \quad \text{(Forward Field)} & \\
 & & \quad count : \mathbb{N}, \quad \text{(Mobility Count Field)} & \\
 & & \quad q : m^*, \quad \text{(Queue Field)} & \\
 & & \quad lock : \mathcal{L}\} \quad \text{(Lock Field)} &
\end{array}
$$

```
RECEIVE(RSR(GP1, ⟨α2, s2⟩, handler, ...))
  { lock(σ(s2, α2).lock);
    if σ(s2, α2).fwd = nil
    then { unlock(σ(s2, α2).lock);
           handle_msg(handler, σ(s2, α2).obj, ...) }
    elif σ(s2, α2).fwd ∈ G
    then { unlock(σ(s2, α2).lock);
           if GP1 ≠ nil then
             post(UPDATE(⟨α2, s2⟩, GP1, σ(s2, α2).fwd, σ(s2, α2).count));     /* update the emitter */
           post(RSR(⟨α2, s2⟩, σ(s2, α2).fwd, handler, ...) }                /* forward the request */
    else { σ(s2, α2).q := σ(s2, α2).q § ⟨RSR(GP1, ⟨α2, s2⟩, handler, ...)⟩;  /* enqueue the request */
           unlock(σ(s2, α2).lock) } }

MIGRATE(⟨α1, s1⟩, ⟨α2, s2⟩)
  { lock(σ(s1, α1).lock);
    if σ(s1, α2).fwd ≠ nil
    then raise "object has already moved"
    else { σ(s1, α1).fwd := fwd;
           σ(s1, α1).count := σ(s1, α1).count + 1;
           unlock(σ(s1, α1).lock);
           copy_object(⟨α1, s1⟩, ⟨α2, s2⟩);   /* returns when object is copied*/
           lock(σ(s1, α1).lock);
           σ(s1, α1).fwd := ⟨α2, s2⟩;
           unlock(σ(s1, α1).lock);
           forward_queued_messages(σ(s1, α1).q, ⟨α2, s2⟩) } }

RECEIVE(UPDATE(⟨α1, s1⟩, ⟨α2, s2⟩, GP, count))
  { lock(σ(s2, α2).lock);
    if σ(s2, α2).count < count then
      { σ(s2, α2).fwd := GP;
        σ(s2, α2).count := count };
    unlock(σ(s2, α2).lock) }
```

Figure 10: Object Migration

as the request is forwarded, GP_1 denotes the latest forwarder met.

When an RSR message is received, the object lock is acquired is order to maintain local consistency. If its fwd field is nil, the mobile object is local, and the method is called locally (with $handle\_msg$). Otherwise, if the fwd field contains a global pointer GP, the object acts as a forwarder to the new position denoted by GP.

Migration of a mobile object $\langle \alpha_1, s_1 \rangle$ to a new position $\langle \alpha_2, s_2 \rangle$ 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 fwd field to the value fwd; 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 fwd 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.

Lemma 10 (Mobility) *User messages are forwarded towards objects with a higher mobility counter.* □

Proof of Lemma 10
Each mobile object is associated with a *fwd* 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. □

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 naïve 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, Lermen 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. Lermen 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 Lermen and Maurer's algorithm, whereas it is involved the *first time* in our algorithm. *(ii)* Lermen 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 Piquer [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 Lermen 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 Plainfossé [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-cut SSP-chains, hereby avoiding the equivalent of *zombie* references. Shapiro, Dickman, and Plainfossé [26] and Piquer [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 lease. 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, Queinnec, and Piquer [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 NeXeme, 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 adapting our implementation to the reference listing technique and are implementing a hierarchical domain organisation, which would make it suitable to program the Internet.

10 Acknowledgement

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

References

[1] David I. Bevan. Distributed Garbage Collection using Reference Counting. In *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer-Verlag, June 1987.

[2] Krishna Bharat and Luca Cardelli. Migratory applications. In *Mobile Object Systems: Towards the Programmable Internet*, pages 131–149. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[3] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed Garbage Collection for Network Objects. Technical Report 116, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1993.

[4] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[5] Peter Dickman. Optimising Weighted Reference Counts for Scalable Fault-Tolerant Distributed Object-Support fsystems, 1992.

[6] Ian Foster. A Multicomputer Garbage Collector for a Single-Assignment Language. *Intl J. of Parallel Programming*, 18(3):181–203, 1989.

[7] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[8] Ian T. Foster, David R. Kohr, Robert Olson, Steven Tuecke, and Ming Q. Xu. Point-to-Point Communication Using Migrating Ports. In *Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 199–212. Kluwer Academic Publishers, 1995.

[9] Benjamin Goldberg. Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme. In *SIGPLAN Programming Language Design and Implemantation PLDI'89*, pages 313–320, 1989.

[10] H.-J.Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software – Practice and Experience*, 18(9):807–820, 1988.

[11] Barry Hayes. Finalization in the Collector Interface. In *Proc. 1992 International Workshop on Memory Management*, pages 277–298, Saint-Malo (France), September 1992. Springer-Verlag.

[12] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *Proc. 1992 International Workshop on Memory Management*, pages 388–403, Saint-Malo (France), September 1992. Springer-Verlag.

[13] R.L. Hudson, R. Morrison, J.E.B. Moss, and D.S. Munro. Garbage Collecting the World: One Car at a Time. In *Proceedings of OOPSLA'97*, Atlanta, USA, 1997.

[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] Bernard Lang, Christian Queinnec, and José Piquer. Garbage Collecting the World. In *Proceedings of the Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 39–50, Albuquerque, New Mexico, January 1992.

[16] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. A Detection Algorithm for Distributed Cycles of Garbage. In *OOPSLA'97 Garbage Collection and Memory Management Workshop*. `http://www.dcs.gla.ac.uk/~huw/oopsla97/gc/papers.html`, 1997.

[17] C.-W. Lermen and D. Maurer. A Protocol for Distributed Reference Counting. In *Lisp and Functional Programming*, pages 343–354, 1986.

[18] General Magic. Telescript Technology: Mobile Agents. http://www.genmagic.com/ Telescript/Whitepapers/ wp4/whitepaper-4.html, 1996.

[19] Luc Moreau. A Distributed Garbage Collector with Diffusion Tree Reorganisation and Object Mobility. Technical Report M97/2, University of Southampton, October 1997.

[20] Luc Moreau, David DeRoure, and Ian Foster. NeXeme: a Distributed Scheme Based on Nexus. In *Third International Europar Conference (EURO-PAR'97)*, volume 1300 of *Lecture Notes in Computer Science*, pages 581–590, Passau, Germany, August 1997. Springer-Verlag.

[21] Luc Moreau and Christian Queinnec. Design and Semantics of Quantum: a Language to Control Resource Consumption in Distributed Computing. In *Usenix Conference on Domain-Specific Languages (DSL'97)*, pages 183–197, Santa-Barbara, California, October 1997.

[22] José M. Piquer. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In *Parallel Architectures and Languages Europe (PARLE'91)*, pages 150–165, 1991.

[23] José M. Piquer. Indirect Distributed Garbage Collection: Handling Object Migration. *ACM Transactions on Programming Languages and Systems*, 18(5):615–647, September 1996.

[24] David Plainfossé and Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. In Henry G. Baker, editor, *International Workshop on Memory Management (IWMM95)*, number 986 in Lecture Notes in Computer Science, pages 211–249, Kinross, Scotland, 1995.

[25] Jonathan Rees and William Clinger. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[26] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Rapport de Recherche 1799, INRIA-Rocquencourt, November 1992. Also available as Broadcast Technical Report #1.

[27] Jon Siegel. *CORBA fundamentals and programming.* Wiley, 1996.

[28] Sun MicroSystems. Java Remote Method Invocation Specification, November 1996.

[29] Gerard Tel and Friedemann Mattern. The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.

[30] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication & Computation. In *Proceedings of the 19th symposium on Computer Architecture*, pages 256–266, 1992.

[31] Paul Watson and Ian Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architectures. In *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 432–443. Springer-Verlag, June 1987.