Hierarchical Distributed Reference Counting

Luc Moreau

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

Abstract

Massively distributed computing is a challenging problem for garbage collection algorithm designers as it raises the issue of scalability. The high number of hosts involved in a computation can require large tables for reference listing, whereas the lack of information sharing between hosts in a same locality can entail redundant GC traffic. In this paper, we argue that a conceptual hierarchical organisation of massive distributed computations can solve this problem. By conceptual hierarchical organisation, we mean that processors are still able to communicate in a peer to peer manner using their usual communication mechanism, but GC messages will be routed as if processors were organised in hierarchy. We present an extension of a distributed reference counting algorithm that uses such a hierarchical organisation. It allows us to bound table sizes by the number of hosts in a domain, and it allows us to share GC information between hosts in a same locality in order to reduce cross-network GC traffic.

1 Introduction

Massively distributed computing has emerged over the last few years as a potentially powerful paradigm of computation. It has taken different shapes: *(i)* In the I-Way experiment [9], multiple sites, communicating over wide-area networks across the American continent, are involved in a distributed computation. *(ii)* The World Wide Web is regarded by many as a support for planet-wide computing: amongst others, WWW programming [5], object systems on the WWW, such as Java and RMI [36], Globe [14, 18] or W3Objects [16], *(iii)* Software agents are autonomous programs, possibly relying on mobility [3, 22] to achieve a task delegated to them; they may cooperate with other agents in order to form agent societies [26]. *(iv)* Amorphous computing [1] involves a multitude of individuals cooperating together to provide a coherent behaviour.

For a long time, the programming language community has defended the idea that garbage collection is an automatic memory management technique that provides the programmer with a powerful abstraction of memory resources, which eases programming. This idea has been taken on board in many languages, including the recent and widespread language Java. Distributed garbage collection, which extends the idea of automatic memory management to distributed memory, has been the focus of much attention [17, 30]. Distributed *reference counting* has been a popular implementation technique of distributed garbage collection because it is simple to implement and can be nicely integrated with sequential garbage collectors [2, 24, 28, 39]. Its variant, *reference listing*, [4, 30, 34, 36], associates objects not only with reference counters, but also with the list of hosts that have references to them; reference listing is useful to offer some fault-tolerance.

Designing a garbage collector for massively distributed computations is, however, a challenging task. We have identified two problems that hamper the scalability of distributed garbage collection.

1. *Reference listing does not scale.* In the presence of massively distributed computations on the Internet, the number of hosts that have access to a given reference can become very high, which requires the garbage collector to maintain very large tables.

2. *Locality is not taken into account.* Distributed GC algorithms usually offer no mechanism by which sites in a same neighbourhood, e.g., two hosts in a cluster, may share information to reduce GC-related traffic with another distant host, e.g., another cluster on another continent.

The first problem is typical of any algorithm that must maintain tables of hosts; the second one is true for all reference counting algorithms that we have encountered.

Our thesis is that massively distributed computations may be conceptually organised in a hierarchy [19, 38] and that distributed garbage collection may rely on this organisation to overcome the two aforementioned scalability problems. An example of hierarchical organisation derived from network interconnectivity is as follows: at the lowest level, we have a COP, i.e. a cluster of PCs, connected to a laboratory network, itself in contact with a departmental backbone, which takes part in a nation-wide network, from which international connections depart. Other hierarchical organisation are conceivable, more specific to the problem being solved.

For every hierarchy, we identify a distinguished site called the *gateway*. The intuition of our algorithm is that the gateway acts, in the hierarchy, as the representative of the

rest of the computation; symmetrically, it acts in the rest of the computation, as the representative of the hierarchy. For instance, if a host in a COP has access to a reference, the rest of the computation does not have to identify the precise host, but the COP gateway as holder of the reference. Similarly, a pointer being duplicated between two sites in the UK does not have to be observed by its owner in the US, as long as reference counters are correctly maintained by the UK gateway.

In this paper, we first present a flat and reference-listing variant of our algorithm for distributed reference counting and diffusion tree reorganisation (Section 2). We then convey the intuition of a hierarchical organisation (Section 3). Afterwards, we illustrate the design of hierarchical reference counting by several scenarios (Section 4). This is followed by a formal presentation of the algorithm (Section 5) and some implementation issues (Section 6). The paper ends by a discussion of related work (Section 7) and a conclusion (Section 8).

Terminology This algorithm has been designed as part of NeXeme [25], a distributed implementation of Scheme, based on the message-passing library Nexus [10]. In NeXeme, computations can proceed in different memory spaces, called *sites*. As in Nexus, there is a notion of *global pointer* GP which is a first-class name for an object; a GP specifies a destination to which a communication can be directed via a form of remote method invocation. As far as garbage collection is concerned, GPs are references to possibly remote objects. In addition, NeXeme provides a function *owner* that returns the site that owns the object at which a GP is pointing.

2 Flat Distributed Reference Counting

We previously sketched a new algorithm for distributed reference counting [25]. We formalised this algorithm and proved its safety and liveness [24]. In this section, we briefly explain a variant of this algorithm that uses reference listing [4, 30]; we also discuss some of its problems if used in massively distributed computations.

Each site owns two tables called *Receive* and *Send* tables, noted Rec_T and $Send\_T$ in the algorithm. A Send-table records triples of information: the global pointers that were sent to remote sites, the sites where they were sent to, and the number of times they were sent; according to the terminology of [4, 30], a Send-table maintains a reference listing. Symmetrically, a Receive-table also records triples of information composed of the global pointers that were received from remote sites, the sites that emitted them, and the number of times they were received; in addition, we assume that a GP constructed (and therefore owned) by a site is entered in its Receive table.

Figure 1.1 displays the situation where a global pointer GP, owned by s_1, is copied from s_1 to s_2, which we model by the message $COPY(s_1, s_2, GP)$. If the GP is copied for the first time, a new entry is created in the Send-table of s_1 for GP, the destination s_2, and the initial value 1; for every new copy towards s_2, the counter is incremented. Symmetrically, a GP received by s_2 from its owner s_1 is entered in the Receive-table; multiply receiving a GP from s_1 increments the counter.

As the Send-table is implemented as a root of the local garbage collector, the presence of GP in the Send-table prevents its space to be reclaimed on s_1. When GP becomes garbage on s_2, GP is removed from the Receive-table of s_2; then, a decrement message $DEC(s_2, s_1, GP, s_2)$ is sent from s_2 to s_1, which in turn removes the entry in the Send-table of s_1, as displayed in Figure 1.2.

The novelty of our algorithm is exhibited in Figure 1.3, when a site s_2, which does not own a GP, sends a copy to a third site s_3. *(i)* An entry for GP and s_3 is added to the Send-table of s_2. *(ii)* When GP is received by s_3 for the first time, an entry is added for GP in its Receive-table. However, the entry records s_1, the owner of the GP, and not s_2, the GP's emitter. *(iii)* A new message $INC\_DEC(s_3, s_1, GP, s_2)$ is sent from s_3 to the owner s_1, to inform the owner of the arrival on s_3 of a GP originating from s_2 (Figure 1.3). *(iv)* When receiving the INC_DEC message, the owner s_1 adds a new entry for GP and s_3 in its Send-table, and then sends a $DEC(s_1, s_2, GP, s_3)$ message to s_2. *(v)* The decrement message sent to s_2 decrements the entry in its Send-table, and removes the entry because it becomes null.

The effect of the INC_DEC message (followed by the DEC message) is to reorganise the diffusion tree of GP. GP was diffused from s_1 to s_2 and then to s_3, but the tables are now recording that two copies of GP owned by s_1 exist on s_2 and s_3. The benefit of this reorganisation is that if GP becomes garbage on s_2, its space can be reclaimed, whereas in Piquer's indirect reference counting [29], a *zombie* pointer would have to be maintained on s_2 as long as GP is used on s_3. Note that the algorithm correctness relies on in-order message delivery; indeed, it is essential to prevent a DEC message from overtaking an INC_DEC message, as this may result in an undesirable object reclaiming.

Further copying of GP from s_2 to s_3, which do not own GP, increase the respective Send and Receive tables (Figure 1.5). Note that it is no longer required to involve the owner with an INC_DEC message, because this only has to be performed the first time the GP is received. At any time, a DEC message may be sent to erase those entries and restore the system in the situation of Figure 1.4.

Thanks to this reorganisation mechanism, this algorithm is able to avoid zombie pointers resulting from computations jumping from node to node. This algorithm however suffers from some defects if used in the context of massively parallel computations [9, 13, 14].

1. *Reference listing does not scale.* The reference listing method requires Send-tables to record all the sites that have access to a given GP. In the presence of the Internet, this may potentially imply very large tables.

2. *The algorithm does not take locality into account.* In Figures 1.3 and 1.4, let us assume that sites s_2 and s_3 are connected via fast communication lines, and that communication with s_1 is slow. The technique to reorganise diffusion trees may force a communication with a site with slow communication. Similarly, in Figure 1.5, sites s_2 and s_3 may be poorly connected and DEC messages from s_3 could be grouped with other messages from sites in the neighbourhood of s_3. The lack of locality awareness also hampers the scalability of other algorithms, such as [4, 28, 36].

In the next section, we present a hierarchical organisation of sites that allows us to avoid these two problems.
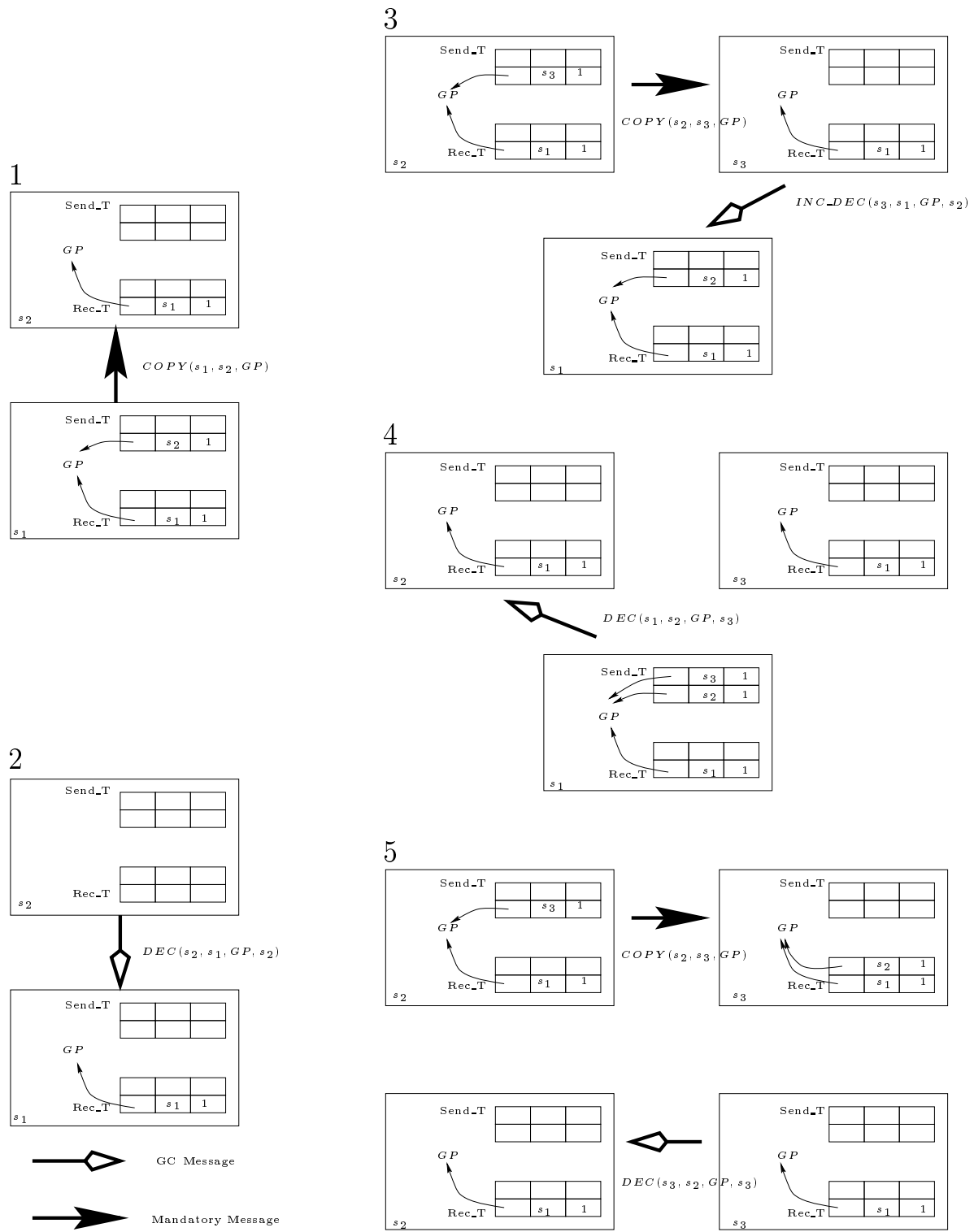
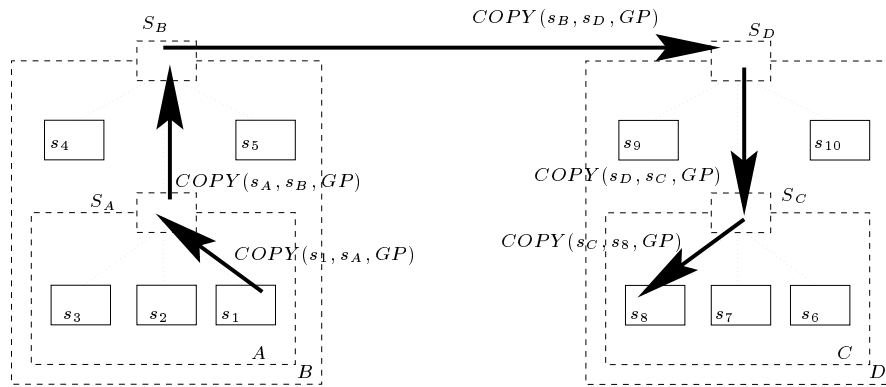Figure 1: Flat Distributed Reference Counting

$COPY(s_B, s_D, GP)$

S_B S_D

s_4 s_5 s_9 s_{10}

$COPY(s_A, s_B, GP)$ $COPY(s_D, s_C, GP)$

S_A S_C

$COPY(s_1, s_A, GP)$ $COPY(s_C, s_8, GP)$

s_3 s_2 s_1 s_8 s_7 s_6

A B C D

Figure 2: Hierarchical Message Forwarding

3 Hierarchical Organisation

Sites that are in the same proximity or in a same logical organisation are said to be grouped in *domains*; for instance, COPs, i.e. clusters of PCs, or NOWs, i.e. networks of workstations, may be regarded as domains. As domains are designed to follow a hierarchical organisation, domains are defined as sets of sites or subdomains. Sites (or subdomains) that belong to a domain are said to be *siblings*. We associate each domain with a distinguished site that acts as a *gateway* to the parent domain. For instance, in Figure 2, s_1, s_2, s_3 are siblings and belong to a same domain A, with gateway s_A; sites s_4, s_5, s_A are siblings and belong to domain B whose gateway is s_B.

For *garbage collection* purposes, we regard the gateway as a key component of a domain: *(i)* The gateway s_A of domain A acts as the representative in domain B of all the sites of domain A. In other words, elements of domain B are only aware of the existence of s_A and not of its children sites s_1, s_2, s_3. *(ii)* Symmetrically, the gateway s_A of domain A acts as the representative in domain A of all the sites that do not belong to domain A. In other words, sites of A are only aware of their sibling sites and of s_A which acts as a proxy for all the other sites.

We want to use this hierarchical organisation in order to guarantee that: *(i)* reorganisation as in Figure 1.3 can be hidden by a domain gateway, *(ii)* table sizes for a given GP are bounded by domain sizes. Figure 2 presents a *conceptual* way of implementing such a hierarchical organisation: gateways between domains could also act as message forwarders. For instance, in order to send a message from s_1 to s_8, s_1 sends it to the gateway s_A because s_8 is outside domain A, which in turn forwards it to s_B for the same reason, etc.

In practice such a solution is *not acceptable* because it potentially delays the sending of messages as it involves several domain gateways in message forwarding[1]. However, from a theoretical point of view, this solution is suitable because a GP copied from s_1 to s_A would create an entry for s_A in the Send-table of s_1, and reciprocally for the Receive-table

[1]We assume here that we are using a network layer, such as TCP/IP, which already efficiently performs routing between networks. Every GC gateway acting as a forwarder would introduce delays. Note also that hierarchical domains may also be different from the network organisation.

of s_A. Such a property would also hold for every gateway, which would guarantee that table sizes are bounded by domain sizes.

In the next Section, we present an extension of the flat GC that does not increase the cost of sending mandatory messages, but maintains a hierarchical organisation as we just described.

4 Hierarchical Distributed Reference Counting

As in every garbage collector design, it is essential to minimize the impact of garbage collection activity on mandatory computation. In particular, remote method invocation, which copies global pointers, should be executed as efficiently as possible. Therefore, instead of hierarchical message forwarding of Figure 2, we prefer the organisation of Figure 3, where we see that the mandatory message is directly sent from s_1 to s_8, but a conceptual hierarchy is kept for GC purposes. Two new messages, introduced to inform gateways of cross-domain messages, are asynchronously sent to gateways of the hierarchy; they are defined in terms of three sites s_1, s_2, s_3:

- With $DOM\_SEND(s_1, s_2, GP, s_3)$, site (or subdomain) s_1 informs its gateway s_2 that GP was sent to s_3 belonging to another domain.

- With $DOM\_RECV(s_1, s_2, GP, s_3)$, site (or subdomain) s_1 informs its gateway s_2 that GP was received from s_3 belonging to another domain.

In Figure 3, DOM_SEND and DOM_RECV messages are repeatedly sent till they respectively reach s_B and s_D, such that s_B is an ancestor of s_1, s_D is an ancestor of s_8, and s_B is sibling of s_D.

When messages are exchanged within a domain, the flat garbage collection algorithm of Figure 1 is used. If global pointers are sent to or received from sites belonging to other domains, then Figure 4 summarises the hierarchical protocol.

In Figure 4.1, site s_2 sends a copy of GP to s_3, a site outside the domain A. A new entry is added to the Send-table of s_2, with GP and the gateway s_A, as if the message had been routed via the gateway s_A. In addition, a message DOM_SEND is sent to the gateway, which acts as if it was forwarding the message. Gateways also maintain Receive
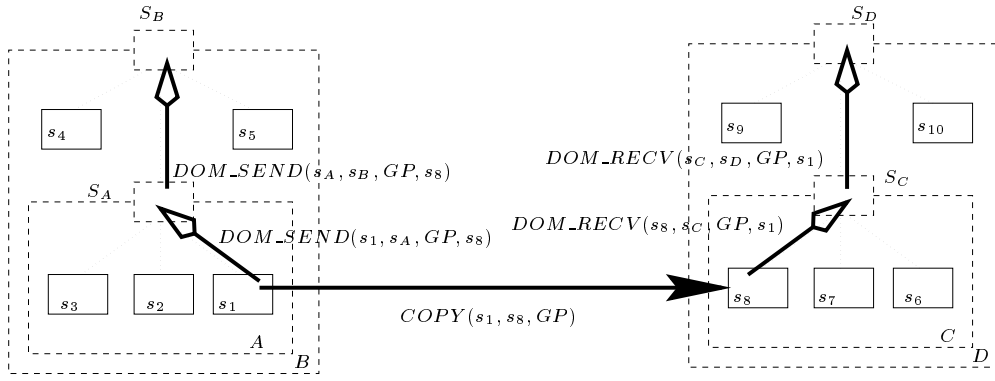
Figure 3: Hierarchical Update of Tables

and Send tables: *(i)* Within domain A, the flat GC algorithm applies. If this is the first time that s_A receives a copy of GP, and if s_1 owns GP, then INC_DEC and DEC messages are sent in the usual triangular mode, and the Receive-table of s_A records that GP comes from s_1. *(ii)* If s_3 (or one of its gateways) is a sibling of s_A, then the Send-table entry of s_A records this gateway. Otherwise, if s_3 (or one of its gateways) is not a sibling of s_A, then the same mechanism applies recursively, and s_A sends a DOM_SEND message to its gateway, etc. Figure 3 illustrates a succession of such DOM_SEND messages. A DOM_SEND message is successively sent to gateways of the hierarchy till it reaches a gateway s_B (in Figure 3), which is a sibling of a gateway s_D, itself an ancestor of the GP receiver s_8.

Figure 4.2 describes the symmetric situation where a site s_1 receives a copy of a GP from s_3 outside the domain A. The system should behave as if GP had been received from the gateway s_A. Hence, a DOM_RECV is sent to the gateway s_A, which implies that the Send-table of s_A records that GP is sent to s_1, and symmetrically for the Receive-table of s_1. If it is the first time that GP is received by s_1 and if GP is owned by s_2, a sequence of INC_DEC and DEC messages is required as in the flat GC algorithm[2]. If s_3 (or one of its gateways) is not a sibling of s_A, then we proceed recursively as illustrated in Figure 3. A DOM_RECV message is successively sent to gateways of the hierarchy till it reaches a gateway s_D (in Figure 3), which is a sibling of a gateway s_B, itself an ancestor of the GP emitter s_1.

In summary, gateways act as the Receive- and Send- tables of domains. The DOM_SEND or DOM_RECV messages consist of incrementing the Send or Receive tables in the gateway, respectively, possibly entailing a triangular re-organisation as in the flat version of the algorithm.

5   The Algorithm

We formalise the algorithm using an abstract machine, called the HGC machine, whose configuration is defined at

the top of Figure 5. A configuration $\mathcal{H}$ is a tuple composed of a set of sites, a set of global pointers, Receive and Send tables for every site, and a bag of messages. Receive and Send tables are functions which for a site s_1, a GP, a site s_2 respectively return the number of times GP was received by s_1 from s_2, or sent from s_1 to s_2. The bag of messages denotes messages that are in transit in the system, i.e. already posted but not yet handled.

We write $owner(GP)$ to denote the site that owns GP. Initially, all table entries are zero, except for hosts that own GPs:

$$
\begin{aligned}
send\_T(s_1, GP, s_2) &= 0, \forall s_1, s_2, GP, \\
rec\_T(s, GP, s) &= 1 \text{ if } owner(GP) = s \\
rec\_T(s_1, GP, s_2) &= 0, \ \forall s_1, s_2, GP, \text{otherwise.}
\end{aligned}
$$

We allow HGC-configurations to perform four transitions. Let $\mathcal{H}_1$ be $\langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$, and let us assume that $s_1, s_2, s_3 \in \mathcal{S}$, $GP \in \mathcal{G}$, $m \in \mathcal{M}$, then:

$$
\begin{aligned}
\mathcal{H}_1 &\Rightarrow MAKE\_COPY(s_1, s_2, GP) \Rightarrow \mathcal{H}_2 &&(make\text{-}copy) \\
\mathcal{H}_1 &\Rightarrow RECEIVE(m) \Rightarrow \mathcal{H}_2 &&(receive) \\
&\quad \text{if } m = DEC(s_1, s_2, GP), \text{ then} \\
&\quad\quad INC\_DEC(s_1, s_2, GP, s_3) \notin \mathcal{M}, \forall s_3 \\
\mathcal{H}_1 &\Rightarrow RELEASE(s_1, GP) \Rightarrow \mathcal{H}_2 &&(release) \\
\mathcal{H}_1 &\Rightarrow DELETE(s_1, s_2, GP) \Rightarrow \mathcal{H}_2 &&(delete)
\end{aligned}
$$

The configuration transformers $MAKE\_COPY$, $RECEIVE$, $RELEASE$ and $DELETE$ are defined in Figures 5 and 6. In Figure 5, if $\mathcal{H}_1 = \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle$, $post(m)$ denotes the configuration $\mathcal{H}_2$ such that $\mathcal{H}_2 = \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \cup \{m\} \rangle$. Similar notational conventions are used for assignments to Send and Receive tables. In the rules above, let us note the side-condition of $(receive)$, which may be implemented by in-order message delivery. We also assume that transitions are executed atomically.

In addition, in order to model the hierarchical domain organisation, we use the following relations:

- $parent(x, y)$: node y is parent of node x.

- $descendant(x, y)$: node y is a descendant of node x, if $x = y$ or $\exists z, parent(z, x) \wedge descendant(z, y)$.

---

[2]The DOM_RECV message to s_A is in competition with the DEC message which respectively increments and decrements the counter for GP in the Send-table of s_A. Even though this may decrement the counter before it is increased, and hence result in a temporary negative value, this has no consequence on the correctness because the DEC message sent from the owner has previously increased a counter on the owner, hereby preventing undesirable reclaiming.

5

1

$INC\_DEC(s_A, s_1, GP, s_2)$ $DOM\_SEND(s_2, s_A, GP, s_3)$

s_A

s_1 s_2

$COPY(s_2, s_3, GP)$

$DEC(s_1, s_2, GP, s_A)$

A

GP owned by s_1 is sent by s_2 outside domain A

2

s_A

$DOM\_RECV(s_1, s_A, GP, s_3)$ $DEC(s_2, s_A, GP, s_1)$

$COPY(s_3, s_1, GP)$ s_1 s_2

$INC\_DEC(s_1, s_2, GP, s_A)$

GP owned by s_2 is received by s_1 from outside domain A

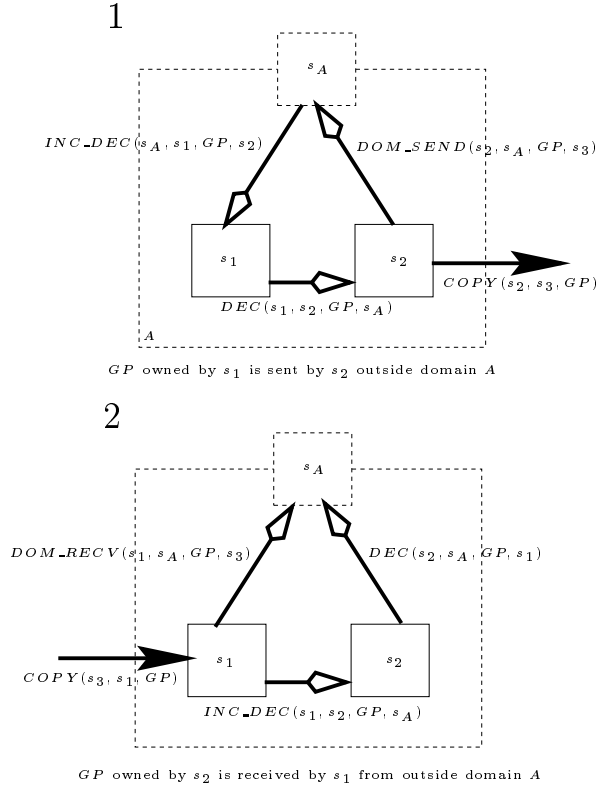Figure 4: Cross-Domain Pointer Copying

- $sibling(x, y)$: node y is a sibling of node x, if $\exists z$, $parent(x, z) = parent(y, z)$.

- $proxy(x, y)$: the representative of y in the domain of x. For the time being, we consider a static hierarchical organisation; the $proxy$ relation is defined as follows.

$proxy(x, y)$
$$= \begin{cases} y & \text{if } sibling(x, y) \\ z & \text{if } \neg sibling(x, y) \wedge \neg descendant(x, y) \\ & \quad \wedge\ parent(x, z) \\ z & \text{if } \neg sibling(x, y) \wedge descendant(z, y) \\ & \quad \wedge\ parent(z, x) \end{cases}$$

Let us first consider the particular case where all sites belong to the same domain, which means that $sibling(x, y)$ holds and that $proxy(x, y) = y$ for any x, y. In such a situation, Figure 5 describes the algorithm for "flat" garbage collection.

Rule (*make-copy*) associated with configuration transformer $MAKE\_COPY$ models the actions that site s_1 has to take before sending a copy of GP to s_2. The $send\_T$ is updated and a $COPY$ message is posted.

Rule (*receive*) and its associated configuration transformer $RECEIVE$ describe how messages are received. When s_2 receives a $COPY$ message from s_1, a triangular reorganisation is initiated if it is the first time GP is received, i.e. the

entry in the receive table for the owner s_g is empty. The triangular reorganisation involves sending an INC_DEC message to the owner, which is followed by a DEC message.

An entry in the receive table pointing at a site that is not the owner of a GP may always be cleared by rule (*release*), which sends a decrement message to the site. Finally, a local garbage collector that proves that a GP has become garbage on site s initiates the transition (*delete*); such a transition can only be fired if Receive and Send tables (except for the owner s_g) are empty, and it results in a DEC message sent to the owner.

If the sites involved in the transitions do not belong to the same domain, rules of Figure 5 remain still applicable, but are now involving proxies of the sites. For instance, receiving a $COPY$ message from s_1, which does not belong to the same domain as s_2, potentially results in a triangular reorganisation with s_p and s_g, respectively proxies of s_1 and the owner (cf. Figure 4.2). In addition, a DOM_RECV message is sent, when s_2 is not sibling with (the proxy of) s_1.

Figure 6 displays how the two messages for hierarchical GC are handled. Receiving a message $DOM\_RECV(s_1, s_2, GP, s_3)$ is similar to receiving a message $COPY(s_3, s_2, GP)$. The only difference is that the gateway s_2 has to act *as if* it was forwarding the message originating from s_3 to s_1 (as in Figure 2). As a result, the entry for s_1 in the $send\_T$ of s_2 has to be incremented.

$$
\begin{array}{rcll}
s \in \mathcal{S} & = & \{s_0, s_1, \ldots, s_n\} & \text{(Site)} \\
GP \in \mathcal{G} & = & \{GP_0, GP_1, \ldots\} & \text{(Global Pointer)} \\
m \in Msg & ::= & COPY(s_1, s_2, GP) \mid & \\
& & DEC(s_1, s_2, GP, s_3) \mid & \\
& & INC\_DEC(s_1, s_2, GP, s_3) \mid & \\
& & DOM\_SEND(s_1, s_2, GP, s_3) \mid & \\
& & DOM\_RECV(s_1, s_2, GP, s_3) & \text{(Message)} \\
\mathcal{M} & : & BagOf(Msg) & \text{(Pool of Messages)} \\
send\_T & : & \mathcal{S} \times \mathcal{G} \times \mathcal{S} \rightarrow \mathbb{N} & \text{(Send Tables)} \\
rec\_T & : & \mathcal{S} \times \mathcal{G} \times \mathcal{S} \rightarrow \mathbb{N} & \text{(Receive Tables)} \\
\mathcal{H} \in Config & ::= & \langle \mathcal{S}, \mathcal{G}, send\_T, rec\_T, \mathcal{M} \rangle & \text{(HGC-Configuration)}
\end{array}
$$

$MAKE\_COPY(s_1, s_2, GP)$ if $s_1 \neq s_2 \ \wedge \ rec\_T(s_1, GP, proxy(s_1, owner(GP))) > 0$

{   let $s_p = proxy(s_1, s_2)$ //s_1 prepares to copy GP to s_2

       $send\_T(s_1, GP, s_p) := send\_T(s_1, GP, s_p) + 1;$

       $post(COPY(s_1, s_2, GP));$

       if $\neg sibling(s_1, s_2)$ then

         $post(DOM\_SEND(s_1, s_p, GP, s_2))$ }

$RECEIVE(COPY(s_1, s_2, GP))$ if $s_1 \neq s_2$

{   let $s_p = proxy(s_2, s_1)$ //s_2 receives a copy of GP from s_1

        $s_g = proxy(s_2, owner(GP))$

       if $rec\_T(s_2, GP, s_g) = 0$ then

         $\{rec\_T(s_2, GP, s_g) := 1;$

          $post(INC\_DEC(s_2, s_g, GP, s_1))$ if $s_p \neq s_g \ \wedge \ s_2 \neq s_g \}$

       else

         $\{rec\_T(s_2, GP, s_p) := rec\_T(s_2, GP, s_p) + 1 \ \ \};$

       if $\neg sibling(s_2, s_p)$ then

       { $post(DOM\_RECV(s_2, s_p, GP, s_1))$ }   }

$RECEIVE(INC\_DEC(s_1, s_2, GP, s_3))$

{   let $s_p = proxy(s_2, s_3)$ //s_2 receives an INC_DEC message

       $send\_T(s_2, GP, s_1) := send\_T(s_2, GP, s_1) + 1;$ //from s_1 which received GP from s_3

       $post(DEC(s_2, s_p, GP, s_1, 1))$ }

$RECEIVE(DEC(s_1, s_2, GP, s_3, n))$

{ $send\_T(s_2, GP, s_3) := send\_T(s_2, GP, s_3) - n$ }   //s_2 receives an DEC message from s_1

$RELEASE(s_1, s_2, GP)$ if $s_1 \neq s_2, s_2 \neq proxy(s_1, owner(GP))$

                                $rec\_T(s_1, GP, s_2) > 0$

                                $rec\_T(s_1, GP, proxy(s_1, owner(GP))) > 0$

{ $post(DEC(s_1, s_2, GP, s_1, rec\_T(s_1, GP, s_2)));$ //s_1 annihilates receive table entry

       $rec\_T(s_1, GP, s_2) := 0$ }

$DELETE(s, s_g, GP)$ if $s_g = proxy(s, owner(GP)),$

                            $s \neq s_g,$

                            $\forall s_i, send\_T(s, GP, s_i) = 0,$

                            $\forall s_i \neq s, s_i \neq s_g, rec\_T(s, GP, s_i) = 0$

{ $post(DEC(s, s_g, GP, s, rec\_T(s, GP, s_g)));$ //GP becomes garbage on s

       $rec\_T(s, GP, s_g) := 0$ }

Figure 5: Flat Garbage Collection

$RECEIVE(DOM\_RECV(s_1, s_2, GP, s_3))$

   $\{$ let $s_p = proxy(s_2, s_3)$ //gateway s_2 is informed that GP

           $s_g = proxy(s_2, owner(GP))$ //is received by s_1 from s_3

        if $rec\_T(s_2, GP, s_g) = 0 \land \neg descendant(s_g, s_2)$ then

          $\{rec\_T(s_2, GP, s_g) := 1;$

             $post(INC\_DEC(s_2, s_g, GP, s_3))$ if $s_p \neq s_g \;\land\; s_2 \neq s_g \}$

        else

          $\{ rec\_T(s_2, GP, s_p) := rec\_T(s_2, GP, s_p) + 1 \};$

          $send\_T(s_2, GP, s_1) := send\_T(s_2, GP, s_1) + 1;$

          if $\neg sibling(s_2, s_p)$ then

           $\{ post(DOM\_RECV(s_2, s_p, GP, s_3)) \;\} \;\}$

$RECEIVE(DOM\_SEND(s_1, s_2, GP, s_3))$

   $\{$ let $s_p = proxy(s_2, s_3)$ //gateway s_2 is informed that GP

           $s_g = proxy(s_2, owner(GP))$ //is copied from s_1 to s_3

        if $rec\_T(s_2, GP, s_g) = 0 \;\land\; descendant(s_g, s_2)$ then

          $\{rec\_T(s_2, GP, s_g) := 1;$

             $post(INC\_DEC(s_2, s_g, GP, s_1))$ if $s_g \neq s_1 \}$

        else

          $\{ rec\_T(s_2, GP, s_1) := rec\_T(s_2, GP, s_1) + 1 \};$

          $send\_T(s_2, GP, s_p) := send\_T(s_2, GP, s_p) + 1;$

          if $\neg sibling(s_2, s_p)$ then

           $\{ post(DOM\_SEND(s_2, s_p, GP, s_3)) \;\} \;\}$

Figure 6: Hierarchical Garbage Collection

Conceptually, when a gateway s_2 receives a message $DOM\_SEND(s_1, s_2, GP, s_3)$ from s_1, it must act *as if* it was forwarding the message to its destination s_3. Therefore, the receive table records the arrival of GP from s_1 (or s_g via the usual triangular reorganisation). In addition, the entry for the proxy of s_3 is incremented in the Send-table of s_2.

Our hierarchical organisation solves the problem of the flat GC algorithm:

- *Locality*: Gateways are hiding reorganisations within their domains. For instance, the copy of GP owned by a site outside the domain results in a triangular exchange as described by Figure 7, where the gateway acts as a representative for the GP's owner.

  In addition, as the gateway s_A of Figure 7 "centralises" GC information about GP for domain A, we are able to send a decrement message to its owner only when GP has become garbage on all sites of domain A. Such sharing of information in a domain allows us to reduce GC traffic across domains.

- *Table Size*: per global pointer, a table now has a maximum number of entries given by the size of the domain. Indeed, every access to a Send-Table (and similarly for a Receive-Table) in the algorithm of Figures 5 and 6 is of the form $send\_T(s_1, GP, s_x)$, where s_x was defined as $proxy(s_1, s_3)$, for any site s_3. So, entries in Send-Tables always refer to sites in the same domain.
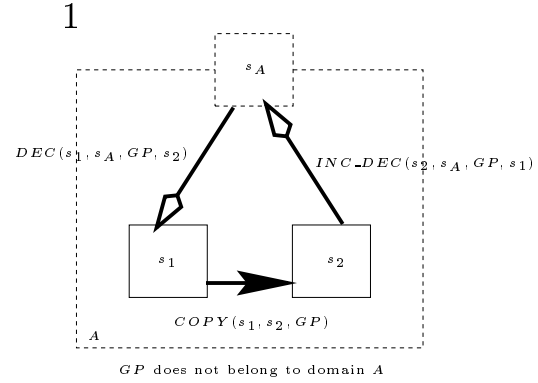

Figure 7: Inside Domain Triangular Reorganisation

6 Implementation

This distributed GC algorithm has been designed as part of NeXeme [25] a distributed extension of Scheme based on the message-passing library Nexus [10]. The flat algorithm has been fully implemented and tested, but (at the time of writing), the hierarchical implementation is still in progress.

An important aspect of the implementation is the design of an efficient function *sibling*. Currently, we define a *hier-*

archical pointer as a structure composed of a global pointer and an *access path*, which is a datastructure representing the path of its owner in the hierarchy. By default, the access path is set to nil, which means that the hierarchical pointer has not exited its domain. Every time a hierarchical pointer is serialised, we have to compare its access path with the access path of the message destination. If the access paths are equal, then the hierarchical pointer is being sent to its domain. Otherwise, from the common path prefix, one can derive the gateways to which the DOM_SEND and DOM_RECV messages must be sent.

For the time being, gateways are built as distinguished processes. We are investigating how to implement them as regular nodes, which could also take part in the computation. So far, we have considered a static hierarchy. We are also studying ways of specifying the hierarchy dynamically and of changing gateways at runtime according to the load of the system.

We have presented here an abstract algorithm, which can be optimised in several ways. *(i)* Several DOM_SEND (or DOM_RECV) could be merged together, in the same spirit as DEC messages that contain a counter value. *(ii)* One message could be avoided in Figure 4.2 by recognising this particular situation, to the detriment of algorithm readability; for instance, the DOM_RECV and DEC messages could be combined in a new message that would follow the INC_DEC message. *(iii)* We can design different strategies for sending GC messages. For the time being, a regular sites sends GC messages as soon as it becomes idle. On the other hand, gateways infrequently send messages to their siblings in order to avoid cross-domain traffic.

7 Related Work

Reference-counting garbage collection was initially developed for uniprocessor systems [6]. Its principle is as follows: every time a pointer is copied or deleted, a reference counter is respectively incremented or decremented. It might seem that this algorithm can be extended straightforwardly to distribution by using two control messages INC and DEC that act on the reference counter residing on the owner of the pointer. Unfortunately, non-causal message delivery may reset the counter even though remote references may still be active. Correct solutions to this problem have been proposed, including weighted reference counting [2, 8, 39] and its optimised version [7], generational reference counting [12], indirect reference counting [28, 29].

Other tripartite exchange of messages including the site that emitted a GP, the receiver of the GP and its owner can be found in the literature, in particular by Lermen and Maurer [21, 37], and by Birrel *et al.* [4]. Our algorithm differs from theirs by the direction in which messages are exchanged and because our solution only requires a tripartite exchange the first time a GP is received. Intuitively, our solution preserves causality because the owner sends a DEC message only after having processed an INC_DEC message which has increased a reference counter.

However, our algorithm has another major benefit as it is able to reorganise diffusion trees: when GC messages are all processed, the diffusion tree is completely flattened, and every site owning a GP directly "depends" from its owner. In the presence of mobile computations jumping from site to site, this allows sites to reclaim the space that was occupied by a mobile program, hereby avoiding *zombie* refer-

ences as in indirect reference counting [28]. To the best of our knowledge, Shapiro, Dickman, and Plainfossé [34, 35] were the first to address the issue of short-cutting chains of pointers. They regard migration as a primitive notion to be supported by the GC; in this paper, we do not deal with migration, however, we have showed that support for mobility could be added as an extra layer, like a library, on top of the current garbage collection algorithm [24].

The distributed collector of Java with Remote Method Invocation [36] is derived from Birrel's network objects [4]. In addition, Java uses a mechanism of lease, by which sites having pointer copies are forced to regularly renew their lease. Such a mechanism supports fault-tolerance and could also be implemented with our algorithm.

Let us note that none of the previously mentioned algorithms is based on a hierarchical organisation as presented in this paper. Therefore, those using reference listing [4, 35, 36] potentially have to manage very large tables in Internet-wide computation. Furthermore, they are not able to share garbage collection information within a neighbourhood, which prevents them from optimising GC information transfer on a local basis, such as per cluster or per network.

We are accustomed to hierarchical memories in uniprocessor systems and memory management has been studied in this particular case [40]. We can regard our schema as a hierarchical organisation of a distributed memory. Other hierarchical organisations have been brought forward. Lang, Queinnec, and Piquer "Garbage collecting the World" uses a hierarchy of nodes that are willing to cooperate for garbage collection; such a similar approach is also adopted in [38] in order to provide scalability. Queinnec [31] also suggests to cluster sites so that they can present a single clock to the rest of the world; clocks are used to provide a causally-coherent distributed memory.

Rodrigues and Jones [32] dynamically identify groups of processes that will collaborate to reclaim distributed cyclic garbage. Their groups provide locality as communication related to the garbage collection activity is only necessary between members of the group. The same authors [33] also explain how groups that have independently initiated a collection on the same cyle may merge together. Maheshwari and Liskov [23] use back tracing [11] to determine if an object is garbage. Back tracing as opposed to forward tracing preserve locality of the tracing process.

Reference counting garbage collection is only able to reclaim acyclic data structures. Other algorithms may be combined with ours in order to collect cycles, such as Le Fessant, Piumarta, and Shapiro's [20], Rodrigues and Jones' [32], or Lang, Queinnec and Piquer's [19]. The latter seems to be particularly appropriate because it also relies upon a hierarchical organisation of sites that cooperate to eliminate cycles between themselves. Gateways in our algorithm contain Send and Receive tables for a domain and can be used to perform a collaborative garbage collection of the domain. The distributed variant of the Train GC [15] is also able to collect cycles; it combines a reference-counting style pointer-tracking mechanism with a substitution protocol.

8 Conclusion

We have presented a hierarchical organisation for distributed reference counting. Such an approach is particularly suitable for Internet-wide programming because it is able to abstract a whole domain of sites by a single host, which acts as its

representative for garbage collection purposes. Such a hierarchical schema allows us to give bounds on the size of tables involved in reference listing and to reduce cross domain GC traffic.

This algorithm is being implemented as part of NeXeme, a distributed implementation of Scheme [25], offering some support for mobile computations [24] and distributed resource control [27]. We foresee two other applications for this algorithm. First, in [26], we associate reference counters with WWW documents, in order to build an agent architecture that offers link integrity in a publishing environment; the new reference counting algorithm may be used to improve scalability of the system. Second, in [27], we present a distributed model of resource control, suitable for agent-style applications. We believe that the present algorithm can be applied to resource control in order to facilitate resource management among sites in a same locality.

9 Acknowledgement

References

[1] Harold Abelson, Thomas F. Knight, Gerald Jay Sussman, and friends. Amorphous Computing Manifesto. Technical report, MIT, 1996. available from http://www-swiss.ai.mit.edu/ ~switz/amorphous.

[2] 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.

[3] 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.

[4] 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.

[5] Luca Cardelli and R. Davies. Service Combinators for Web Computing. In *Usenix Conference on Domain Specific Languages DSL97*, Santa-Barbara, California, October 1997.

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

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

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

[9] Ian Foster. High-Performance Distributed Computing: the I-WAY Experiment and Beyond. In *Euro-Par'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 3–10, Lyon, France, August 1996.

[10] 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.

[11] Matthew Fuchs. Garbage Collection on an Open Network. In *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995.

[12] 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.

[13] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds, Jr. A Synopsis of the Legion Project. Technical Report CS-94-20, Deparment of Computer Science, University of Virginia, June 1994.

[14] P. Homburg, M. van Steen, and A. S. Tanenbaum. Communicating in GLOBE: an Object-Based Worldwide Operating System. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, pages 43–47, Seattle, Washington, October 1996.

[15] 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.

[16] D. B. Ingham, M. C. Little, S. J. Caughey, and S. K. Shrivastava. W3objects: Bringing Object-Oriented Technology to the Web. In *Proc. Fourth International World-Wide Web Conference*, pages 89–105, Boston, Mass., USA, December 1995.

[17] Richard Jones and Rafael Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[18] I. Kuz, A.M. Kermarrec, M. van Steen, and H.J. Sips. Replicated Web Objects: Design and Implementation. In *Proc. Fourth Annual ASCI Conference*, Lommel, Belgium, June 1998.

[19] 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.

[20] 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.

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

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

[23] Umesh Maheshwari and Barbara Liskov. Collecting Cyclic Distributed Garbage by Back Tracing. In *Proceedings of PODC'97 Principles of Distributed Computing*, 1997.

[24] Luc Moreau. A Distributed Garbage Collector with Diffusion Tree Reorganisation and Object Mobility. In *Proceedings of the Third International Conference of Functional Programming (ICFP'98)*, Septembre 1998.

[25] 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.

[26] Luc Moreau and Nicholas Gray. A Community of Agents Maintaining Links in the World Wide Web (Preliminary Report). In *The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents*, pages 221–235, London, UK, March 1998.

[27] Luc Moreau and Christian Queinnec. Distributed Computations Driven by Resource Consumption. In *IEEE International Conference on Computer Languages (ICCL'98)*, pages 68–77, Chicago, Illinois, May 1998.

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

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

[30] 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.

[31] Christian Queinnec. Sharing Mutable Objects and Controlling Groups of Tasks in a Concurrent and Distributed Language. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings of the Workshop on Theory and Practice of Parallel Programming (TPPP'94)*, number 700 in Lecture Notes in Computer Science, pages 70–93, Sendai (Japan), November 1994. Springer-Verlag.

[32] Helena C. C. D. Rodrigues and Richard E. Jones. A Cyclic Distributed Garbage Collector for Network Objects. In *Tenth International Workshop on Distributed Algorithms WDAG'96*, number 1151 in Lecture Notes in Computer Science, Bologna, October 1996.

[33] Helena C. C. D. Rodrigues and Richard E. Jones. Cyclic Distributed Garbage Collection with Group Merger. In *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98*, Lecture Notes in Computer Science, pages 249–273, Brussels, 1998.

[34] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, Distributed References and Acyclic Garbage Collection. In *Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992.

[35] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Rapport de Recherche 1799, INRIA-Rocquencourt, November 1992.

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

[37] 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.

[38] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable Distributed Garbage Collection for Systems of Active Objects. In *Proc. 1992 International Workshop on Memory Management*, pages 134–147, Saint-Malo (France), September 1992. Springer-Verlag.

[39] 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.

[40] Paul R. Wilson. Uniprocessor Gargage Collection Techniques. In *International Workshop on Memory Management*, Lecture Notes in Computer Science, Saint-Malo, France, September 1992.