Tree Rerooting in Distributed Garbage Collection: Implementation and Performance Evaluation

LUC MOREAU l.moreau@ecs.soton.ac.uk
Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ. United Kingdom.

Received June 15, 2000; Revised February 22nd, 2001

Editor: Takayasu Ito

Abstract. We have recently defined a new algorithm for distributed garbage collection based on reference-counting [20, 24]. At the heart of the algorithm, we find *tree rerooting*, a mechanism able to reduce third-party dependencies by reorganising diffusion trees. In reality, the algorithm describes a spectrum of algorithms according to the policy used to manage messages. In this paper, we present the implementation of the algorithm and evaluate its performance. We have implemented two policies, which are extremes of the spectrum, respectively using and not using tree rerooting. In addition, two different strategies for managing action queues have been implemented. The conclusions of our experimentations are the following. Tree rerooting offers more parallelism during distributed GC activity; we explain this phenomenon by the length reduction of causality chains in the distributed GC. Grouping messages per destination dramatically reduces the number of messages, but requires a more complex implementation as messages have to be sorted per destination. Speed up of 100% has been observed on some benchmarks.

Keywords: distributed garbage collection, distributed reference counting, performance evaluation, benchmark

1. Introduction

Over the last decade, distributed symbolic computing has found useful applications outside research laboratories. Environments for developing distributed applications are now shipped by major software suppliers, and are used to produce advanced applications involving complex interactions between multiple clients and servers. Java, which plays a dynamic role in this context, is bundled with the RMI communication layer [12] able to activate methods on remote objects.

In particular, RMI provides a distributed garbage collector (also written distributed GC or DGC) that turns out to be a very 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 author has recently published an algorithm for distributed garbage collection [20]. This algorithm based on distributed reference counting was developed and prototyped as part of NeXeme [23], a distributed implementation of Scheme, based on the message-passing library Nexus [9]. This algorithm has been studied in detail, and mechanical proofs of safety and liveness have been carried out using the proof assistant Coq [24]. Furthermore, the algorithm describes a family of algorithms and it may be optimised in several ways. In this paper, we focus on a

real implementation of the algorithm, and we undertake a comparative study of its variants and optimisations.

The algorithm was implemented in C and uses the Nexus library [9] for communications. Local garbage collection is handled by the Boehm-Demers-Weiser collector [10]. The implementation is about 5000 lines of code, plus an extra 2000 for instrumentation. Plans to implement the algorithm in Java are underway; combined with the NexusRMI stub compiler [4], it would give access to a multi-language garbage-collected distributed environment.

This paper is organised as follows. First, we state our assumptions regarding the distributed computing model and we define some terminology in Section 2. We analyse current distributed reference counting algorithms in Section 3, and explain the design rationale of our distributed GC algorithm. Then, we summarise its principles in Section 4. The overall implementation design is presented in Section 5. Some benchmark programs are described and algorithm performance is evaluated in Section 6. Finally, a comparison with related work and a summary conclude the paper.

2. Models and Terminology

In this section, we introduce the models of distribution and communication that we have adopted as well as some useful terminology. This paper studies distributed reference counting on a network of processes that can communicate only by exchange of messages. A computation executes on a set of *processes* and consists of a set of *threads*. An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same process. In this context, a process denotes a memory space holding a number of objects. Processes may be distributed across several physical machines, but also a single machine can host several processes; in both cases, the only means by which processes communicate is by exchange of messages. We shall assume that communications are reliable and ordered between any pair of processes.

In the description of our garbage collector implementation, an *address* denotes the location of an object in a process; such an address is only meaningful within its process. The need for distributed garbage collection comes from the usage of a notion of *remote reference* by which a process can refer to an object in a possibly remote process. As opposed to an address, a remote reference can be communicated to other processes. We define the *owner* of an object as the process where the object is located; by extension, the owner of a reference denotes the owner of the object the reference points at.

2.1. The Communication Model

In this Section, we present the characteristics of the communication model that is expected by our implementation. Even though they are heavily influenced by Nexus [9], they are generic because they form the essence of a distributed object system.

Host pointers and a mechanism for initiating remote computations are two key facets of a distributed object system. A *host pointer* is a network representative of an object, able to refer to an object in a possibly remote process; alternatively, it may be called global pointer [9] or network pointer [3]. We take a view of message passing such that the arrival of a message initiates a computation to handle the message [9]. Therefore, remote method invocation as in Network Objects [3] or Java RMI [12] is implemented by two messages, to activate the computation and to transport the result, respectively.

The actual interface to the communication layer is library dependent. Generally, a communication is specified by providing a host pointer, a handler identifier, and a data buffer, in which data are serialised. Initiating the communication causes the data buffer to be transferred to the process designated by the host pointer, after which the routine specified by the handler is executed, potentially in a new thread of control. Both the data buffer and the object referred to by the host pointer are made available to the handler. Host pointers must be a predefined data type supported by the communication library; they must be serialisable and transportable to remote processes.

2.2. Garbage Collection and Reachability

In each process, objects are allocated in a heap managed by a local garbage collector. The purpose of a local garbage collector is to collect objects that are no longer locally reachable. Local reachability is defined in terms a set of roots. The *roots* are locations holding addresses of heap objects; generally, roots are processor registers, program stacks and global variables. An object in the heap is *locally reachable* if its address is held in a root or if its address is held in another locally reachable heap object [14].

The communication library and its host pointers offer the means to refer to objects in remote processes. In this context, the purpose of a *distributed* garbage collector is to collect objects that are no longer globally reachable. An object o is *globally reachable* if one of the following conditions holds:

1. if o is locally reachable in a process taking part in the computation,

2. if there is a globally reachable object that contains the address of o,

3. if there is a globally reachable object that contains a host pointer referring to o.

(Note that a more detailed description of how a host pointer refers to an object will be given later in the paper.)

Following the custom in DGC literature, an application is regarded as composed of two separate activities: the *mutator* performs the actions specified by the program, whereas the garbage collector takes care of all activities related to automatic memory management.

3. Background

Distributed reference counting and its derivatives are by and large the most commonly found techniques used to implement distributed garbage collection. Even though such techniques are not able to reclaim distributed cycles, they are frequently adopted because they are easy to interface with uniprocessor garbage collectors, which only need to provide finalizers (that cause reference decrement messages to be sent) and a notion of a weak pointer (so that DGC tables do not cause objects to be preserved incorrectly by the collector) and do not need to support special forms of tracing.

Reference counting was initially conceived in the context of uniprocessor applications [5]. In a reference counting system, each object is associated with a reference counter. An object's counter is incremented every time a new reference to the object is created, and decremented every time such a reference is destroyed. A reference counter equal to zero indicates that there is no reference to the object and therefore the space occupied by the object may be reclaimed safely.

The reference counting technique may naïvely be extended to a distributed setting by introducing two messages *increment* and *decrement*, but unfortunately this results in an incorrect algorithm. The essence of the problem is summarised in Figure 1, where a reference *ptr* to object o of process $\mathcal{P}_1$ is passed from process $\mathcal{P}_2$ to process $\mathcal{P}_3$, immediately followed by process $\mathcal{P}_3$ deleting its reference to *ptr*. A naïve extension of reference counting would send an *increment* message to increment the counter on $\mathcal{P}_1$ when *ptr* is sent to $\mathcal{P}_3$ and a *decrement* message to decrement the counter on $\mathcal{P}_1$ when $\mathcal{P}_3$ deletes its reference. A race condition between *increment* and *decrement* messages may cause the counter on $\mathcal{P}_1$ to be decremented before it is incremented, possibly making it null temporarily; a null counter would make the object ready for collection, even though there still is a live reference *ptr* on $\mathcal{P}_2$.

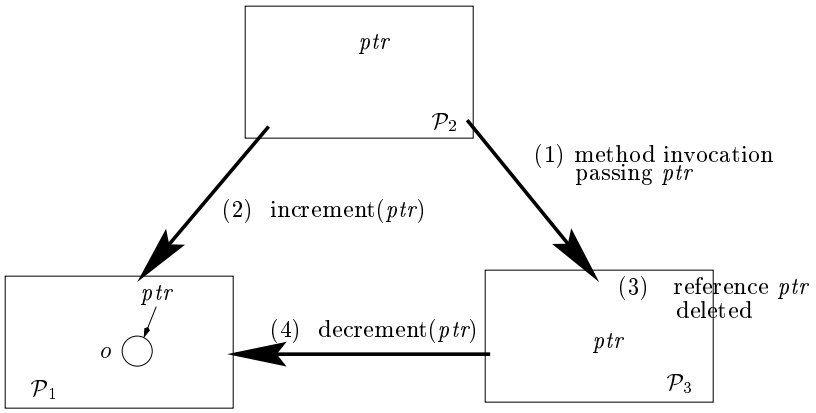


Figure 1. Naive Extension to Distributed Context

Different approaches have been proposed to address the shortcomings of the naïve extension, which essentially fall into three categories described below.

Triangular Protocols The scenario of Figure 1 involves three different processes, namely the owner, the sender and the receiver of a reference. Lermen and Maurer [15, 31] were the first to introduce a protocol between these processes in order to make the copy of a reference safe. Birrel *et al.* [3] also proposed another algorithm, which was adopted in Network Objects [3] and in Java RMI [12]. We summarise these algorithms below.

Lermen and Maurer [15, 31] introduce two new messages *create* and *acknowledge*. When a reference is duplicated, a *create* message is sent to its owner, which increases the owner's reference counter and is followed by an *acknowledge* message to the reference receiver. When a reference is deleted, a decrement message is sent to the owner; the receiver ensures that no race condition can take place between a *create* message and a *decrement* message by sending a *decrement* message only after an acknowledgement for this reference has been received.

In this protocol, the owner is involved *every time* the emitter sends a reference to a receiver. 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.

Birrel *et al.* also present a triangular algorithm as part of Network Objects, a distributed object language with remote method invocation [3]. Similarly to Lermen and Maurer, their protocol ensures that reference counters are not decremented prematurely. For this purpose, synchronisations are introduced between garbage collection and mutator activities.

The owner of an object maintains a "dirty" set, which contains identifiers for all the processes that have a reference to the object. When a client first receives a reference, it makes a *dirty* call to the owner. When the reference is no longer locally reachable, as determined by the client's local GC, the client makes a *clean* call and deletes the reference. In order to avoid conflicts between dirty and clean calls, the emitter keeps the reference reachable until it receives an acknowledgement from the receiver. The acknowledgement may be returned as part of the result when the reference is passed as an argument to the remote method call; otherwise, an extra message may be required when the reference is returned as the result of a remote method call.

In Birrel's algorithm, distributed reference counting activity is synchronous with the application. In particular, unmarshalling may be suspended by dirty calls. Furthermore, the emitter of a reference is only allowed to free its reference after the method invocation has terminated on the receiver: this may potentially maintain the existence of a pointer in the emitter for garbage collection purpose only, even though its presence is not required by the mutator.

Other triangular protocols have been proposed. The distributed variant of the Train GC [11] relies on a reference-counting style pointer-tracking mechanism (on top of which cycle collection is built). This pointer tracking bears a resemblance to Birrel's algorithm but attempts to minimise the number of exchanged messages.

Mancini and Shrivastava [16] also use a triangular protocol as the basis of a fault-tolerant version of distributed reference counting.

Weighted Reference Counting Weighted reference counting (WRC) [2, 34, 6] avoids race conditions between messages by sending decrement messages only. WRC associates a weight with each object and with each reference. When an object and its first reference are created, they are given a same weight. The algorithm maintains the invariant that the weight of an object is equal to the sum of weights of references pointing at it. Whenever a reference is deleted, a message is sent to the owner with the reference weight, whose effect is to decrement the object weight by the reference weight. When a reference is copied, its weight is (equally) divided between the two copies without communication with the owner.

Different solutions may be adopted when the weight of a reference reaches one and cannot be divided further. *(i)* A new object containing a reference to the initial object may be created; such a new object, usually called *indirection cell*, and its associated reference are given equal weights. The new reference is then propagated according to the same algorithm. *(ii)* A message may be sent to the owner in order to request more weight.

Indirect Reference Counting Piquer's indirect reference counting (IRC) [27] does not use increment messages, which resulted in race conditions with decrement messages in the naïve extension. Instead, his algorithm relies on a representation of the path along which references are propagated in a distributed application. While, in general, the propagation path of references can form a cyclic structure, a spanning tree, called the *diffusion tree*, may be obtained by remembering the path along which a reference is propagated to a process for the first time.

Indirect reference counting requires each process in the diffusion tree to maintain a counter for each reference it holds. Every time a process sends a reference remotely, it increases the counter associated with this reference. The diffusion tree is built and maintained at runtime: each process that has received a reference for the first time just needs to remember which process the reference was sent from; by definition, the receiving process is said to be the sender's child in the diffusion tree. When a reference is deleted by a process, indirect reference counting sends a *decrement* messages to the parent of this process in the diffusion tree. If a process sends a reference to another process that already holds a copy of the reference, the sender increases its reference counter, assuming that the receiver will join the diffusion tree as its child; as the receiver already holds a copy, a decrement message needs to be sent back to the message emitter. In a stable situation (when all messages have been propagated), the sum of all counters for a given reference indicates the number of remote copies for the counter.

Indirect reference counting introduces third party dependencies, where a reference is kept in a process for the sole purpose of IRC because its children in the diffusion tree still contain active references. Such references, which Piquer calls *zombie pointers*, are harmful because they prevent the completeness of the garbage collection process (in other words they cause memory leaks), and they further expose the system to failures.

Discussion Let us now discuss these algorithms by examining *(i)* the synchronisations they force with the mutator; *(ii)* the third party dependencies they introduce; *(iii)* the number of messages they generate; *(iv)* the type of network connectivity they require; *(v)* their ability to support mobile objects.

Today, Birrel's algorithm is the most widely used, as it is part of Java RMI [12] and Network Objects [3]. As summarised above, this algorithm however introduces synchronisations between the mutator and the garbage collector, which may slow down the computation.

Birrel's and Piquer's algorithms introduce third party dependencies which prevent the sender of a reference from freeing it. In Birrel's case, dependencies take place for the duration of a remote method invocation. In Piquer's, they last as long as references are reachable in the receiver (which can be much longer than in Birrel's case). Depending on the solution adopted when pointer weights become one, weighted reference counting introduces third party dependencies or synchronisations. An indirection cell introduces an unwanted form of dependency; alternatively, contacting the owner to obtain more weight introduces an undesired delay to the mutator. The frequency at which these events occur is however much lower that in Birrel's and Piquer's because they only happen when pointer weights can no longer be divided.

If we analyse the number of generated messages, all algorithms require only one message when a reference is deleted. WRC and IRC require no extra message when a reference is communicated, though WRC may require a communication with the owner when the weight becomes one. Triangular protocols require up to two additional messages; Lermen and Maurer's solution systematically requires two messages because the sender is indifferent to whether the receiver already holds a reference.

Another interesting aspect for comparison is how processes are required to be connected in order to propagate DGC messages. IRC is unique because DGC messages follow the same path as mutator's messages. Other algorithms require direct connectivity with the owner: this may not necessarily be a valid assumption in the presence of firewalls, which make some part of the network invisible [1]. Finally, except for IRC, none of these algorithms support mobile objects.

Shapiro, Dickman and Plainfossé [30, 29] analyse the problem of third party dependency and are the first to propose a mechanism to collapse chains of pointers. Details of their algorithm is presented in the related work section. At this point, it is important to note that their technique supports mobile objects and partial connectivity: it is able to adapt to the network configuration by preventing the short-cutting of pointers if and when required [1]; on the downside, DGC activity is synchronised with the mutator activity.

Our goals are similar to Shapiro, Dickman and Plainfossé's, because we also want to support mobile objects and to collapse the distributed state, whenever possible according to connectivity (or even according to locality [21]). However, we adopted a modular approach: first, we investigated the collapse of state for non-mobile objects [20, 24], then we studied the requirements for mobile objects [22]. The needs differ substantially and a benefit of our approach is that it helps

us to understand the interactions between the different algorithm components. In addition, our solution does not introduce any synchronisation with the mutator, and keeps the number of messages to be exchanged low, by requiring a maximum of two messages, only when a reference is received for the first time. Since the first publication of our algorithm, Dickman [7] has independently discovered a variant of our rerooting technique; we delay a deeper comparison with his approach until the related work section.

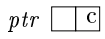
In this paper, we focus only on the distributed garbage collection of non-mobile objects, and the mechanism we have introduced to avoid third party dependencies: *tree rerooting*. In particular, we investigate the performance implications of this technique. Our experimentation may be summarised as follows: tree rerooting introduces more parallelism by reducing the length of causality chains; grouping DGC messages dramatically decreases DGC traffic. In the following section, we describe our algorithm.

4. The Algorithm

The essence of our algorithm can be summarised as follows. When a process $\mathcal{P}$ receives a reference for the first time, indirect reference counting dictates that the process has to join the reference's diffusion tree as the child of the sender process. *Tree rerooting* is the sequence of operations by which a process $\mathcal{P}$ can become a direct child of the owner process (if not already so). Tree rerooting requires a sequence of two messages: the first one is a request from $\mathcal{P}$ to the owner to change parent, and the second one is a message from the owner to the previous parent informing it of the transfer.

In the rest of the section, we present the key features of the algorithm. Our algorithm has been presented, formalised and proved correct in other papers [20, 24]. The complete formal specification of the algorithm in Coq is available from: `http://www.ecs.soton.ac.uk/~lavm/coq/drc`.

At an abstract level, we deal with a notion of remote reference, which we call DGC pointer. A reference counter is associated with each DGC pointer. A DGC pointer *ptr* is conventionally represented by two boxes. The first box is used only in the owner, where it contains the address of the object *ptr* refers to; it is empty in the other processes. The second box contains a reference counter c (initially zero).



We also assume that each participating process maintains a table, called a *receive-table*, that contains the set of DGC pointers received by the process and currently regarded as reachable by the local GC; to this end, the receive-table is *not* considered as a root for the local GC. (This aspect will be discussed further in the implementation section.) In addition to regular messages sent by the mutator, two new messages named INC_DEC and DEC are introduced to maintain accurate reference counters.

Figure 2 illustrates a configuration with three processes $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$. The first process is the owner of an object o and each process also contains a pointer *ptr* referring to o. Each instance of the pointer is associated with a reference counter; receive tables are represented as boxes annotated by the letter "R" in the right-hand side of each process.

Reference counters, receive tables and DGC messages are used in the following rules that regulate the sending of pointers (S), the receiving of pointers (R1, R2, R3), the handling of DGC messages (DEC, INC), and local garbage collection (GC).

In the text, we conventionally use numbers in brackets to refer to actions in figures; they identify successive messages and their respective effect on the local state of a process.

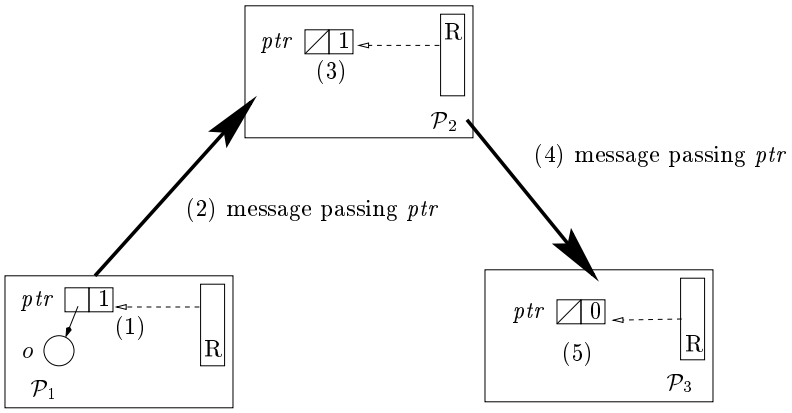


Figure 2. Pointer Diffusion

S: Every time a pointer is sent to a remote process, its associated counter is incremented by one. In Figure 2, *ptr* is sent from $\mathcal{P}_1$ to $\mathcal{P}_2$ (2) and then to $\mathcal{P}_3$ (4); its reference counter is incremented on $\mathcal{P}_1$ (1) and $\mathcal{P}_2$ (3). In a given process, the reference counter associated with a pointer represents the number of branches leaving that process in the pointer's diffusion tree. In $\mathcal{P}_3$, the counter for *ptr* is zero because the pointer has not been passed to other processes.

R1: If a process receives a pointer sent by its owner, and if the pointer does not belong to the receive table, then the pointer is entered in the receive table. The receive table on $\mathcal{P}_2$ points to *ptr* (cf. action (3)); when a pointer is received for the first time, the initial value of its reference counter is set to 0.

R2: If a process receives a pointer sent from a process other than its owner and if the pointer does not belong to the process receive table, then the pointer is entered in the receive table; an INC_DEC message is also sent to the owner,

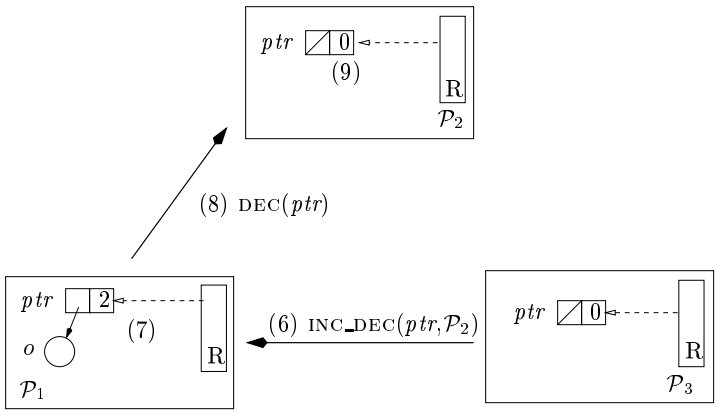


Figure 3. Tree Reorganisation: Rerooting

containing a reference to the emitter $\mathcal{P}_2$. Figure 2 shows that $\mathcal{P}_3$ receives *ptr* from $\mathcal{P}_2$ (5); in Figure 3, action (6) is such an INC_DEC message.

INC: When a process receives an INC_DEC message referring to a pointer and a third process, it increments the counter associated with the pointer and it posts a DEC message to the third process. In Figure 3, after the INC_DEC message is received, the reference counter of *ptr* is incremented (7) by $\mathcal{P}_1$, and a DEC message is sent to $\mathcal{P}_2$ (8).

DEC: When a process receives a DEC message pertaining to a pointer, it decrements the counter associated with the pointer. After receiving the DEC message, the reference counter of *ptr* is decremented by $\mathcal{P}_2$ (9), giving the value 0.

GC: The purpose of the local GC is to detect data structures that are no longer reachable from the set of local roots. As DGC pointers are regarded as regular data structures by the local GC, the local GC can detect when a DGC pointer is no longer locally reachable. When such an event occurs, the pointer is removed from the receive-table, and a decrement message DEC is sent to the pointer's owner.

R3: If a process receives a pointer sent by any emitter and if the pointer is already present in its receive table, then a DEC message is sent back to the emitter.

The key idea of the algorithm is the INC_DEC message followed by the DEC message, whose effect is to propagate reference counter values safely from internal processes of the diffusion tree to the owner (which is the root of the diffusion tree for the current pointer). As this operation grafts a subtree onto the root of the diffusion tree, we call it *tree rerooting*. Tree rerooting does not affect the sum of the reference counters for an object; we can see that the sum of reference counters

remains equal to 2 in Figures 2 and 3: this means that there are two remote copies of *ptr*. In Figure 3, after the tree rerooting operation has taken place, the sum of all the counters is found on the owner. In other words, the effect of the INC_DEC message is to flatten the diffusion tree.

All participating processes maintain another table called *send-table*, not represented in the figures. When a reference counter is increased, the pointer is entered in the process send-table, if it was not present. When a reference counter is decreased and reaches 0, the pointer is removed from the send-table. In order to ensure safety and liveness, send-tables as opposed to receive-tables are defined as roots of local garbage collectors. The role of the send-table is the following. On processes other than the owner, pointers held by the send-table are kept reachable while messages containing copies of these pointers are in transit and until rerooting has completed. On the owner, the send-table is key to the safety of the algorithm: pointers on the owner are kept reachable while a copy is in transit or reachable anywhere in the network [24]. Consequently, the data associated with the pointer will not be reclaimed by the owner's local GC, because the send-table makes the pointer, and therefore the data, reachable from the local GC roots.

Our algorithm avoids the race condition between increment and decrement messages of naïve reference counting by always incrementing the owner's reference counter, with an INC_DEC message, before decrementing it on another process with a DEC message. We should note that the algorithm requires in-order message delivery between any pair of processes, so as to avoid a DEC message overtaking an INC_DEC message; indeed, not doing so may result in the undesirable decrementing of a counter before its incrementing.

A Spectrum of Algorithms An important aspect of our algorithm is that the rerooting operation *is not* synchronous with the mutator's activity. Rerooting requires two messages, but this takes place for the first arrival of a reference at a process only. In networks where full connectivity to the owner is not available, rerooting does not have to be performed. In addition, the algorithm may be optimised in two ways. First, all DGC messages may be delayed and grouped: by grouping DGC messages, one can significantly reduce the amount of DGC-specific traffic. Second, R2 immediately followed by GC may be optimised by a single DEC message to the process that sent the pointer: this optimisation only requires a single message instead of two, and it involves two processes instead of three.

Our algorithm's ability to use different strategies adapted to the network connectivity and message traffic is unique. In fact, the algorithm and its optimisations characterise a whole *spectrum of reference counting algorithms*. At one end of the spectrum, eager activation of rerooting with rule R2 tends to flatten the diffusion tree. At the other end of the spectrum, the activation of rerooting may be delayed as much as possible: in this case, INC_DEC messages are never sent, and the algorithm behaves as Piquer's indirect reference counting algorithm [28]. In our experiments, we shall see that eager rerooting generates more messages than indirect reference counting. However, the increase of messages is compensated by the potential for parallel execution of the DGC activity, which in one of benchmarks results in a reduced overall execution time.

5. Implementation Design

Our implementation of the algorithm relies on two libraries for communications and local garbage collection: we use the Nexus message-passing library for communications [9] and the Boehm-Demers-Weiser local garbage collector [10]. The Boehm-Demers-Weiser garbage collector was used because it is conservative and multithreaded and therefore can easily coexist with the Nexus C library; it is also the garbage collector used by Bigloo at the heart of our distributed Scheme, NeXeme [23]. A major concern was to design a *generic implementation*, which could be made independent of these libraries.

5.1. The Nexus Communication Library

Section 2.1 introduced the broad characteristics of the communication model assumed by our implementation. In this Section, we explain the actual representation of host pointers adopted by Nexus.

According to the Nexus communication paradigm, a communication flows over a virtual communication link identified by a *startpoint* and an *endpoint*. A startpoint can be thought of as a capability granting rights to operate on the associated endpoint. Both startpoints and endpoints can be created dynamically; the startpoint has the additional property that it can be moved between processes; on the contrary, endpoints cannot be copied between processes.

A communication link supports a single communication operation: an asynchronous *remote service request* (RSR). An RSR is applied to a startpoint by providing a procedure name and a data buffer. For the endpoint linked to the startpoint, the RSR transfers the data buffer to the address space in which the endpoint is located and remotely invokes the specified procedure, providing the endpoint and the data buffer as arguments.

We refer the reader to the Nexus literature describing the rationale of Nexus design. In particular, we note that Nexus is portable over multiple communication protocols: the virtual communication link identified by a startpoint-endpoint pair is able to hide the details of the actual communication protocol used.

5.2. Distributed GC Pointers

We consider that each process memory space is divided in two heaps: the first one is managed by a local GC, which can automatically deallocate objects; the second heap is under the responsibility of the communication library. (In the case of Nexus, object deallocation is not automatic.)

Figure 4 shows how the heap managed by the local garbage collector can coexist with the heap managed by the communication library. Users' programs allocate objects in the garbage collected heap. Host pointers pointing at these objects are allocated in the communication library heap. A Nexus host pointer is comprised of a startpoint-endpoint pair. An endpoint contains the address of a user object

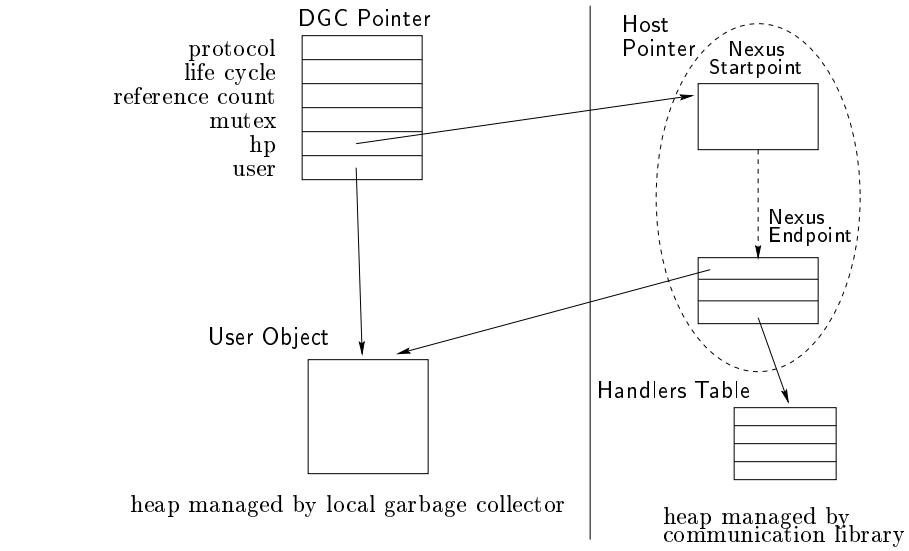


Figure 4. Objects, Host Pointers, and DGC pointers

allocated in the garbage collected heap and is associated with a handler table; a startpoint is bound to an endpoint and may be copied to remote heaps.

Host pointers may be regarded as network representatives of user objects. In order to define our distributed garbage collector, we introduce a new kind of object, called a DGC *pointer*, that is a garbaged-collected, network representative of a user object.

The combination of Nexus with our distributed garbage collector is intended to provide a component of the runtime system of a distributed language. The language designer has a wide range of options to specify what programming interface is actually made available for the programmer. High-level annotations for distribution such as `future` could be provided [19]; alternatively, DGC pointers can be made first-class objects, which would provide a low-level but powerful programming interface, as in NeXeme [23]. The libraries are directly usable from C, and at the implementation level, DGC pointers are regarded as regular data structures, which can be collected by the local GC; host pointers are invisible to the user and are used only by the communication library.

A DGC pointer is a data structure allocated in the garbage-collected heap. The real implementation refines the "two boxes" notation of Section 4, by using several fields:

- a user address, which points at the user data on the owner, and which is null on any other process; the user data resides in the garbage collected heap;
- a host pointer address, which, for Nexus, is the address of a communication startpoint;

- a reference counter indicating the number of times the pointer has been sent to a remote process;
- a mutex to ensure exclusive access to the DGC pointer content;
- the distributed garbage collection protocol handling this object;
- a state information describing the point in the DGC pointer's life cycle, which we explain below.

Figure 5 displays two processes $\mathcal{P}_1$, $\mathcal{P}_2$. Process $\mathcal{P}_1$ is the owner of an object, and a DGC pointer p points at this object; the same pointer was sent to $\mathcal{P}_2$. On $\mathcal{P}_1$, the user address of p points at the object, whereas it is null on $\mathcal{P}_2$. The host pointer field refers to a Nexus startpoint, which itself points to an endpoint ep on $\mathcal{P}_1$. The endpoint ep only exists in the owner process $\mathcal{P}_1$, and also refers to the object.

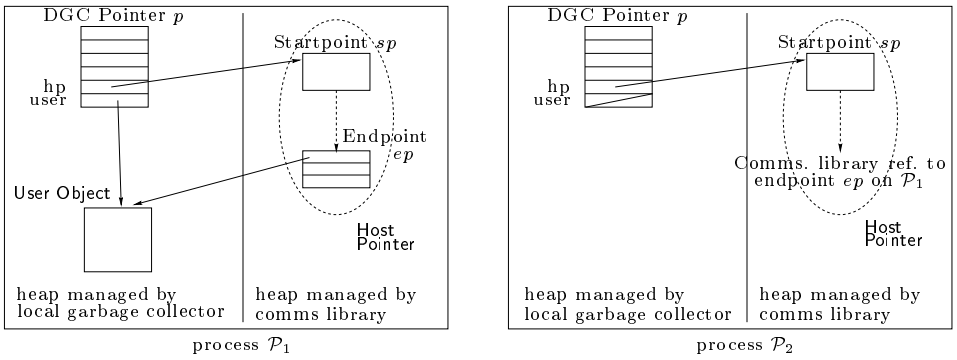


Figure 5. Pointers on two processes

Section 2.2 defined global reachability in an abstract manner. We can restate the third condition in terms of our implementation. An object o is *globally reachable* if one of the following conditions holds:

1. if o is locally reachable in a process taking part in the computation,
2. if there is a globally reachable object containing the address of o,
3. if there is a globally reachable object containing (the address of) a DGC pointer p, if p contains the address of a startpoint sp associated with an endpoint ep, and if ep contains the address of o. (Note that ep and o are required to be in the same process, and sp and ep are allowed to be in different processes.)

The idea of a send-table containing DGC pointers with a non-null reference counter is crucial to our implementation. Indeed, condition 3 of the definition can safely[1] be approximated as follows:

3 (revised). if there is a DGC pointer p in the send-table of the owner of o.

As we have defined send-tables to be roots for the local GCs, the first and third conditions may now be merged into a single one; our algorithm safely approximates the *globally reachability* of an object o if one of the following conditions holds:

1. if o is locally reachable in a process taking part in the computation,

2. if there is a globally reachable object containing the address of o.

As a result, when a local garbage collector detects that an object is no longer locally reachable, it also means that no other DGC pointer to this object exists in the system, whether in a remote process or in a message in transit.

From a local garbage collector's point of view, DGC pointers are regular data structure, for which local reachability can be determined as for any other object. DGC pointers have a life cycle composed of three successive states, and our distributed GC implementation has an explicit representation of this state in a field of a DGC-pointer. The only allowed transitions are from live to phantom and phantom to dead; they are only fired by the local garbage collector detecting the non-reachability of a DGC pointer.

1. *live*: When a DGC pointer is created, it is said to be live, and it remains so, as long as it is locally reachable.

2. *phantom*: The local garbage collector is responsible for detecting when a DGC pointer is no longer locally reachable. Our implementation uses finalization to change the pointer state to "phantom". During this new stage, the pointer is no longer used (not even reachable) by the mutator. It is removed from the receive-table but it remains in use by the DGC so that rule (GC) can be completed, i.e. a DEC message is sent to the owner.

3. *dead*: Once rule (GC) has terminated for a DGC phantom pointer, the local garbage collector can detect that it is no longer locally reachable for a second time; then, the DGC pointer can again be finalized and it enters the new phase "dead". In this third phase, resources used by the associated host pointer in the communication library should explicitly be deallocated.

In the implementation, there may be a delay between the moment a live DGC pointer is detected to be unreachable by the local GC, at which point it is marked as phantom, and the moment it is removed from the receive-table. Therefore, a receive-table contains live pointers, but may also contain phantom pointers, unreachable by the mutator and waiting to be removed. Consequently, rules R1 and R2 need to be refined: their firing requires that a process receives a DGC-pointer that is not currently in the receive-table with the live flag. In this case, our implementation creates a new entry in the receive-table for the newly received pointer, with a live flag.

All DGC pointers have a three-stage life cycle, but a DGC protocol determines the distributed garbage collection policy that regulates this object: the policy determines when, where, or what kind of message pertaining to this pointer has to be sent. The DGC protocol is specified by an explicit field of a DGC pointer. Protocols are also represented by a data structure whose fields are shown in Figure 6.

When a message containing a DGC pointer is received by a process, the DGC protocol uses a receive-table to determine whether the pointer is present locally. Similarly, the protocol maintains a send-table that contains all DGC-pointers with a non-null reference counter. (We remind the reader that the send table is a root for the local garbage collector but the receive table is not.)

The local garbage collector plays an essential role because it detects when a DGC pointer is no longer locally reachable; during finalization, it moves DGC pointers to the phantom or dead stages of their life cycles. During these phases, the DGC or the communication layer may have to send messages; these operations may be long and may require a substantial amount of memory. It is not suitable to perform these operations during the finalization itself; indeed, finalization may be performed inside the garbage collector critical section, and one prefers to leave this section as quickly as possible to avoid starving other threads running in parallel and requiring more memory. There are further implementation-dependent considerations to be taken into account: is the finalizer run in a new thread? what is the priority of the finalizer thread? In our implemenentation, finalized DGC pointers are entered in a finalization queue (cf. (1) in Figure 6).

A dedicated thread, called the DGC *thread*, is responsible for transferring pointers from the finalization queue into an action queue (cf. (2) in Figure 6). This action queue is also directly used by protocols, when for instance, upon receiving a pointer, a DGC message must be emitted. As we do not want the DGC activity to delay the mutator, DGC messages are not immediately sent, but enqueued in the action queue. The same DGC thread is also responsible for activating items from the action queue (3); this operation typically requires calling a procedure of the communication layer.

Finally, the polling thread of the communication library is responsible for reading incoming messages and activating the corresponding handlers. The distributed garbage collector implementation provides some handlers for handling DEC and INC_DEC messages or initialising remote processes.

The protocol data structure contains pointers to the receive and send tables, to the finalize and action queues, to functions for pointer creation, finalization, communication notification, and deallocation, and to all necessary mutexes to protect access to critical resources. Most of our code is parameterised by the protocol managing the current DGC pointer; this organisation facilitates easy dispatch. This implementation technique allows us to have pointers managed by different protocols running at the same time in a single environment.

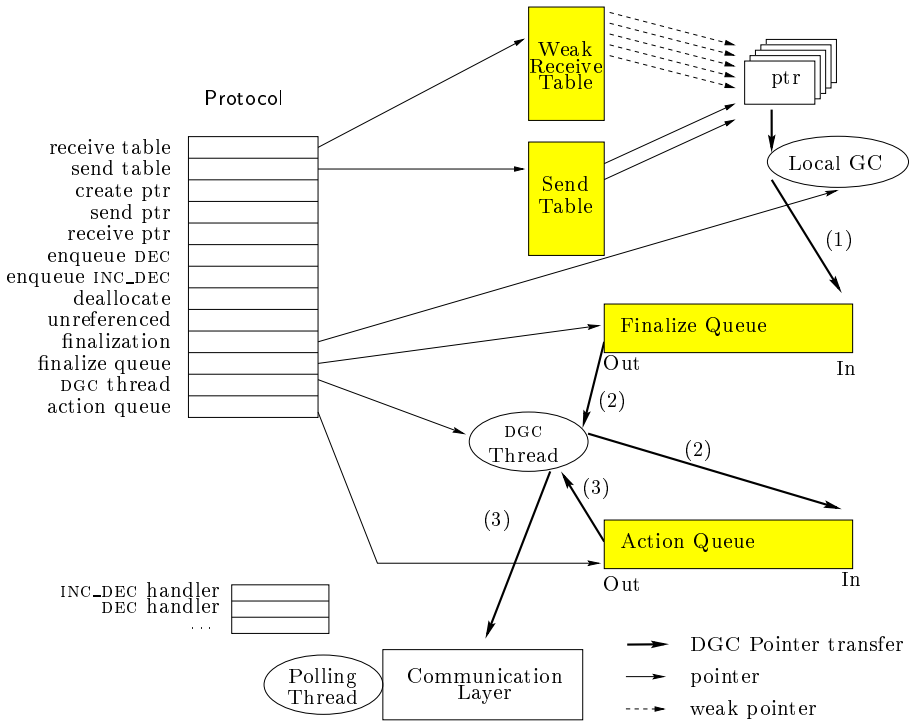


Figure 6. Protocol and associated data structures

5.4. Implemented Protocols

In order to study the benefits of the INC_DEC message on the diffusion tree reorganisation, we have implemented the following protocols.

5.4.1. Tree Rerooting with INC_DEC *Messages* If a DGC pointer p is received by a process, and if p is not a pointer in its live stage present in its receive-table, and if the emitter and receiver are both different from the pointer's owner, then an INC_DEC message is prepared and enqueued in the action queue. When processed, the INC_DEC message will be sent to the pointer's owner. If a DGC pointer p is received by a process, and if p is live and present in its receive-table, then a DEC message is prepared and enqueued in the action queue.

Let us observe that we do not send DGC messages immediately, but add them to the action queue. This approach has two benefits: *(i)* Receiving a pointer p typically occurs as part of receiving a remote method invocation, which activates the mutator; by delaying the sending of a DGC message, we favour mutator over

garbage collection activity. *(ii)* By managing all DGC messages in a same queue, we may optimise them, as explained in Section 5.6.

5.4.2. Indirect reference counting In this protocol, INC_DEC messages are not used. DGC pointers now have an extra field, called *emitter*, which contains a reference to the process that emitted the pointer. If a DGC pointer p is received by a process, and if p is not a live pointer present in its receive-table, then the emitter field for p is set to the process that emitted p. Otherwise, DEC messages are prepared as in the first protocol. Once the pointer is finalized and reaches the phantom state, a DEC message is no longer sent to its owner but to the process contained in the emitter field of the pointer.

5.4.3. The null protocol The null protocol does *not* maintain reference counters for pointers. In our implementation, processes are denoted by null protocol pointers. Such pointers are added to every communication so that recipients of a message can determine its originator.

5.5. Actions

Actions are operations to be performed by the distributed GC. Actions are managed by the action queue. Currently, the algorithm supports three types of actions.

5.5.1. Sending a DEC *message* In the simple case, this consists of sending a single DEC message related to a pointer. In the most complex case, this action sends a message to decrease the reference counters of several pointers, by an amount specified for each pointer.

5.5.2. Sending an INC_DEC *message* In the simple case, this consists of sending an INC_DEC message to a pointer's owner. In the most complex case, the message acts upon the reference counters of several pointers.

5.5.3. Deallocating a pointer This action deallocates all resources used by a host pointer in the heap managed by the communication library.

5.6. Implemented Action Queues

The action queue is a data structure containing actions. Any implementation strategy is valid for action queues, as long as it preserves the safety conditions set by the algorithm: *(i)* DEC messages cannot overtake INC_DEC messages if they are related to a same pointer; *(ii)* deallocation of a pointer cannot take place before the last message to that pointer has been emitted. We have implemented two variants of the action queue.

5.6.1. The FIFO action queue Actions are handled in a fifo manner. No attempt is made to merge messages that are related to the same pointers.

5.6.2. The sorted action queue Each action for sending a DEC message entered in the queue is merged with a similar action related to the same pointer. Therefore, actions for sending DEC messages are associated with a number specifying the amount by which a counter has to be decreased. In order to preserve the safety condition, a DEC message is only extracted from a sorted queue if there is no INC_DEC message waiting to be processed. In order to facilitate these operations, actions are sorted by their type: INC_DEC messages in a queue, while the other actions are maintained in a second queue. As INC_DEC messages have a lower frequency, we have decided not to merge them.

A further optimisation is possible: if a DEC message is sent to the owner of a pointer, and if the sorted queue contains an INC_DEC message for the same pointer (to be followed by a DEC message to a process $\mathcal{P}$), then these messages can be replaced by a single DEC message to $\mathcal{P}$ directly. While this optimisation reduces the number of messages, we have not implemented it, because it would not have been used in any significant manner in the benchmarks we propose in the following section. However, systematically applying this optimisation after delaying INC_DEC messages gives us IRC, which we have implemented.

6. Performance Evaluation

In this section, we evaluate and compare the different protocols and the different action queues that we have implemented. First, we describe the benchmark programs that we have designed; second, we present our experiments results.

6.1. Benchmark Programs

The scientific programming community has developed a series of benchmarks for sequential and parallel languages. Similarly, Gabriel's benchmarks and nofib are respectively used by the Lisp and Haskell communities; Feeley [8] has produced a series of programs to measure the efficiency of MultiLisp. We dramatically lack benchmarks specifically designed for evaluating the performance of distributed garbage collectors.

Unfortunately, we lack real applications to evaluate the overall effect of DGC algorithms. It is a major problem to evaluate algorithms, but we are confident that such applications will become available as platforms such as Java RMI become widely used.

Instead, we have specifically devised two benchmarks that exhibit the performance of our implemented policies: *cycle* highlights the effect of INC_DEC messages, whereas *diffuse* shows the benefits of the sorted action queue. The benchmarks are an abstraction of the behaviour of real programs, where distributed garbage collection may have an impact on the performance and/or robustness of the computation.

The first benchmark *cycle* is reminiscent of the behaviour of mobile agents that migrate while keeping a pointer to their home base; making sure that the mobile agent becomes independent of the locations it has visited is an essential requirement to ensure its robustness. The second benchmark *diffuse* is an abstraction of the behaviour of a distributed search across different WWW servers, where these servers cooperate in order to find reachable pages with a specific content [18]. The problem in this context is detecting the termination of the search, which can be implemented by using distributed garbage collection [31]; ensuring a prompt detection of the distributed termination is of paramount importance in order to obtain an efficient system.

6.1.1. Cycle The first benchmark is aimed at measuring the benefit of reorganising the diffusion tree, using INC_DEC messages; its description can be found in Figure 7. We consider a fixed number N of processes, forming a cycle, where the successor of process $\mathcal{P}_{N-1}$ is process $\mathcal{P}_0$. The first process allocates a DGC pointer, passes it to the second process, which in turn passes it to the third one, and so on. In order to avoid sending a pointer p to a process that has already received it, we create a new DGC pointer p', every $N-1$ steps. The pointer p' is defined so as to point to a newly created object, which contains the address of the current pointer p; as a result, p is locally reachable from p'. Then, we repeat the algorithm with p'.

Algorithm parameter:	N
Sequence of processes:	$\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{N-1}$
Number of cycles:	c

Initial Configuration:

- *Create a* DGC *pointer* p *on process* $\mathcal{P}_0$*; send* $(p, c, N-2)$ *to* $\mathcal{P}_1$.

Process $\mathcal{P}_i$ *receiving* (p, j, k)*:*

- *If* $k > 0$*, then send* $(p, j, k-1)$ *to process* $\mathcal{P}_{i+1 \ mod \ N}$.
- *If* $k = 0$ *and* $j = 0$*, then lose any reference to* p *and terminate.*
- *If* $k = 0$ *and* $j > 0$*, then create a* DGC *pointer* p' *referring to a new object that contains the address of* p*, and send* $(p', j-1, N-2)$ *to process* $\mathcal{P}_{i+1 \ mod \ N}$.

Figure 7. Cycle Benchmark

Let us analyse the reachability of the different pointers. After a DGC pointer p has been communicated successively to $N-1$ processes, a DGC pointer p' is created. As p' is referring to an object that contains p, p remains reachable as long as p' is reachable. When the iterations terminate, the last DGC pointer is released, which in turn releases the previous one, and so on. If indirect reference counting is used, a DGC pointer p on process $\mathcal{P}_i$ remains reachable as long as its child in the

diffusion tree (in process $\mathcal{P}_{i+1\ mod\ N}$) remains reachable. As a result, the length of the reachability chain is $N \times c$ with indirect reference counting. Tree rerooting does not force a pointer p to remain reachable when its child in the diffusion tree is also reachable. As a result, except on its owner, any DGC pointer becomes garbage as soon as the pointer has been sent remotely and tree rerooting has taken place.

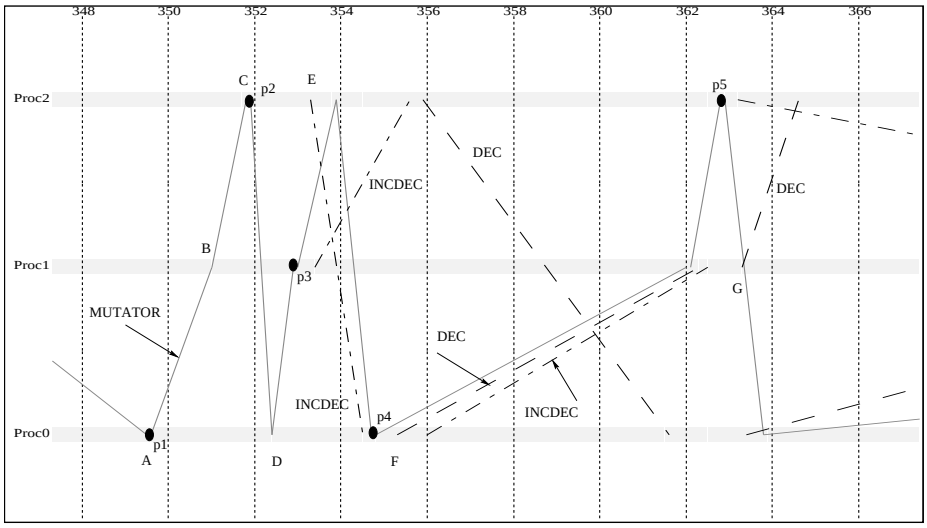


Figure 8. Diffusion tree reorganisation (1 unit=1ms)

Figure 8 displays an example of execution. It displays the timelines of three processes participating in the computation. Thin gray lines represent mutator's messages carrying pointers; bullets represent DGC pointer creation. We see that a pointer $p1$ is passed from process 0 (label A) to process 1, and then to process 2, where a new pointer $p2$ is created (label C), passed to 0, and then 1, and so on.

Dashed-dotted lines represent INC_DEC messages, whereas dashed lines denote DEC messages. We also see that when process 2 receives the first pointer, it sends an INC_DEC message (label E) to process 0, its owner, which in turn sends a DEC message (label F) to process 1. The benefit of this reorganisation is that the pointer $p1$ is no longer needed on process 1: it may be finalized and a DEC message may be sent from process 1 to 2 (label G, at time 363). This example is an illustration that tree rerooting avoids third party dependencies introduced by Piquer's indirect reference counting.

The cycle program is typical of mobile applications, which for instance keep a pointer to their home base while they migrate. Every hop of a mobile program potentially keeps the home base pointer in the send-table of the previous process. Tree rerooting resets the counter for that pointer on the parent, which can then reclaim its space.

In such a benchmark, interesting measurements concern the *(i)* number of messages, *(ii)* total time (including time to cleanup), *(iii)* length of causality chains (to be explained in Section 6.3).

6.1.2. Diffuse In the cycle program, the mutator's activity is sequential, as it repeatedly propagates a single pointer to process after process. Even though a limited distributed garbage collection activity may take place in parallel, there is not much room for optimising DGC activity. Therefore, we designed a second benchmark *diffuse*, which measures the benefit of grouping DGC messages per destination (cf. Figure 9).

Algorithm parameter:	N
Sequence of processes:	$\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{N-1}$
Width:	W
Depth:	D

Initial Configuration:

- *Create a* DGC *pointer* p *on process* $\mathcal{P}_0$*; send* (p, W, D) *to* W *randomly chosen processes.*

Process $\mathcal{P}_i$ *receiving* (p, w, d)*:*

- *If* $d = 0$*, then lose any reference to* p *and terminate.*
- *If* $d > 0$*, then send* $(p, w, d-1)$ *to* w *randomly chosen processes.*

Figure 9. Diffuse Benchmark

We consider a number of process N, such that each process knows about the $N-1$ other processes. A process receives a DGC pointer and two integers w (width) and d (depth). If d is zero, then the pointer is discarded. Otherwise, the same pointer is propagated to w destination processes chosen randomly, with a depth given by $d-1$.

This program diffuses a pointer along a tree of a specified depth and width, whose processes are decided at runtime. We select the width and depth such that the total number N of processes is much smaller than w^d: as a result, a same pointer will be communicated to a same process several times during a short period.

Let us analyse the reachability of the single DGC pointer p used by the diffuse benchmark. The first time a pointer p is sent to w processes, a tree rerooting can take place. We expect each process rapidly to obtain a copy of p as we assumed that the total number of processes N is much smaller than w^d. Indeed, even though no further activity is performed by a process after it has sent a pointer, the frequency at which the pointer is sent keeps the pointer reachable in a process for the duration of the execution. When rerooting has taken place, each non-owner process will be

a direct child of the owner in the diffusion tree. When the program terminates, non reachability can be detected by child processes and remaining DGC-messages can be propagated, which will eventually make the pointer unreachable in the owner.

The first three interesting measures are similar to the ones for cycle, the last two are specific to this benchmark: *(i)* number of messages, *(ii)* total time (including time to cleanup), *(iii)* length of causality chains (to be explained in Section 6.3), *(iv)* counters values in a DEC message, *(v)* number of DGC pointers in a single DEC message.

6.2. Measure Quality

Measuring the performance of a distributed garbage collector is not a trivial task because, as for any other distributed algorithm, execution may be non-deterministic due to processes or threads scheduling and messages propagation. The benchmark diffuse even uses random number generators.

The absence of a global clock has also influenced the design of our benchmarks. In order to measure a time duration, we made sure that the measures were taken on a single process. Graphical visualisations display timelines for several processes; some of these timelines may be shifted with respect to each other because time 0 is not defined globally.

In addition, there are issues that are specific to garbage collection. There is a strong partnership between the distributed garbage collector and local garbage collectors: it is their cooperation that provides automatic distributed memory management. In particular, several DGC events are triggered by finalization initiated by local garbage collectors, when some objects are detected to be garbage. For instance, sending a DEC message when a pointer is no longer needed or deallocating communication resources are both started by finalizers. Consequently, the performance of our DGC algorithm cannot be measured independently of the local collector[2].

Furthermore, specific properties of local garbage collectors come into effect. For instance, in our case, the local garbage collector is incremental, and needs to allocate objects in order to reclaim existing garbage and activate finalizers. In order to circumvent this feature, we created a thread whose sole purpose is to allocate garbage to be sure that objects that are relevant to our experiments get finalized.

6.3. Comparison

For each benchmark, we display execution runs that give *some intuition* about the patterns of message exchange and the overall execution time. Then, we extract and discuss specific measurements that are not time related. We have successfully run our programs on Linux 2.0 and SunOS 5.6, with communication taking place over a 10Mb/s ethernet. As they all presented similar patterns of communication, we present here runs where all processes ran on a single Linux machine.

Figure 10 displays the messages that were sent for two runs of the cycle programs (a) using tree rerooting and (b) using indirect reference counting (IRC). Mutator

messages are represented by gray lines. (We invite the reader to download coloured versions.) In Figure 10 (a), mutator activity is interleaved with DGC message exchanges and is spread between 2233 and 4629 ms. In Figure 10 (b), mutator activity is between 1850 and 2061 ms. The overall computation terminates after 18000 ms in Figure 10 (a) but only after 35000 ms in Figure 10 (b).

With IRC, the distributed garbage collection activity does not take place when the mutator is proceeding, but only starts after the mutator has accomplished its execution and has released the last pointer, in the last process, after the end of the last iteration. A sequence of DEC messages is then propagated in a direction opposite to the one the pointer was propagated. On the other hand, as previously illustrated in Figure 8, DGC activity is interleaved with the mutator computation when INC_DEC messages are used. A very large number of DGC messages may be propagated at a very early stage *in parallel*. As a result, we can see that the time to clear all pointer allocated by the cycle benchmark is halved when using tree rerooting.

This behaviour may be explained as follows. We consider the configuration reached after n iterations of the cycle benchmark, and we assume that the last pointer in the last process is still regarded as live. We imagine that all INC_DEC and subsequent DEC messages have been propagated. We then obtain a configuration such that for each iteration of cycle, there remains a single instance of a DGC pointer that is live.

Consequently, once the last pointer in the last process is detected to be garbage, a domino effect will follow, freeing the pointer it is referring to, and so on. The number of DEC messages that remains to be propagated is given by the number of iterations that were completed. On the contrary, with IRC, no tree rerooting took place, and the number of DEC messages is given by the number of iterations multiplied by the number of processes in a cycle.

The benchmark was designed so as to model mobile computations, hopping from processes to processes. If we consider the limit of a very high number N and a single iteration, after all INC_DEC and DEC messages were propagated, there remains a single DEC message to send, whereas in the case of IRC, N messages still have to be sent.

Figure 10 also shows that using tree rerooting reduces the total duration of the benchmark (in this figure, 1 unit is $1ms$). The time to complete the execution of the mutator is longer with tree rerooting then with IRC, because some DGC activity takes place (including the activation of local collectors). However, when using tree rerooting, about three times more messages are exchanged than in IRC, but messages can be propagated in parallel, which results in an halved execution duration.

The absence of parallelism when using IRC can be explained in a more formal way. The length of the *longest causality chain* is substantially higher in the indirect reference counting algorithm than in the presence of tree rerooting. Causality [17] is a partial relationship between events, which expresses the causal dependencies between events: if event e_1 causally depends on event e_2, then e_2 necessarily occurs *after* e_1. Causal dependencies occur in the following cases:

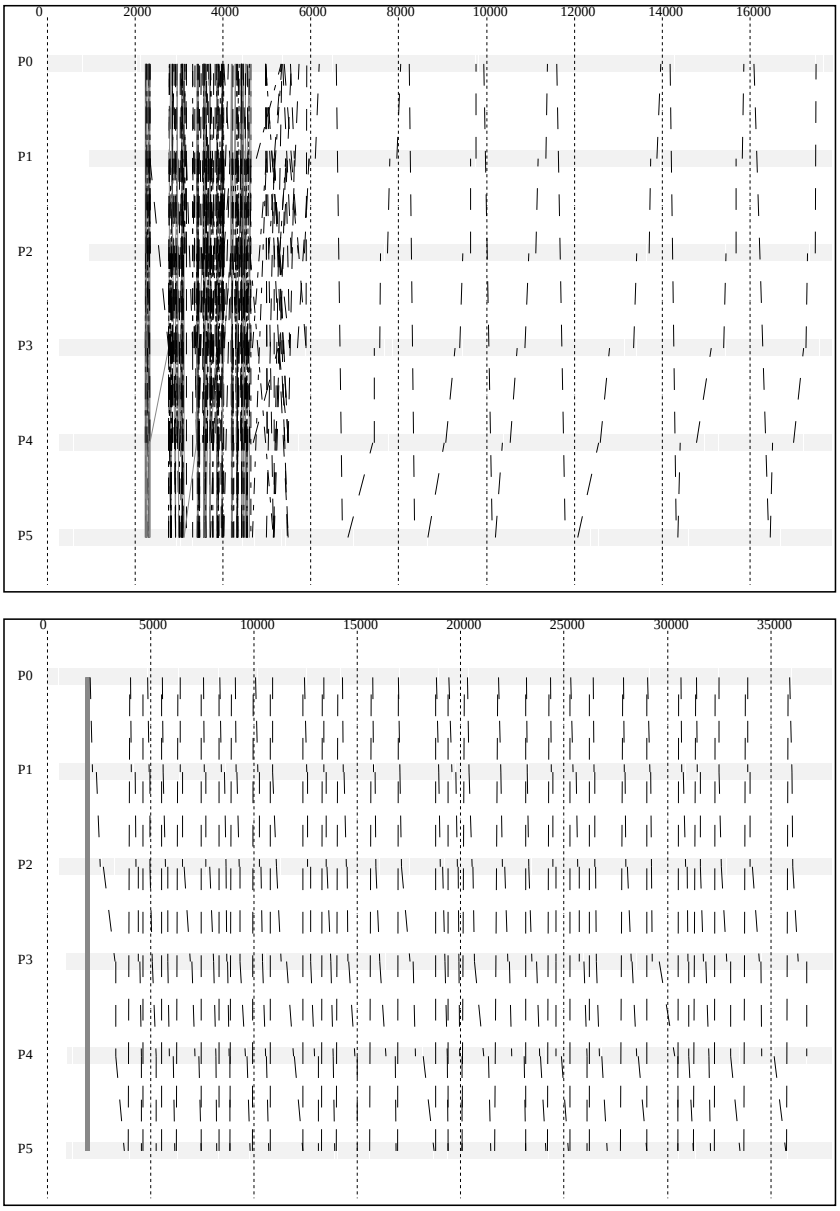


Figure 10. The cycle benchmark with (a) tree rerooting and (b) IRC; one unit is 1ms

A coloured version of these figures is available from
`http://www.ecs.soton.ac.uk/~lavm/papers/hosc01-colour.tar.gz`.

1. in a given process, if a DGC pointer is successively involved in two events e_1 and e_2, then e_2 causally depends on e_1;
2. receiving a message causally depends on sending it;
3. if event e creates a DGC pointer p pointing at another DGC pointer p', then e causally depends on the last event incurred by p' in the same host.

The causality relationship is a partial order from which we can derive a directed acyclic graph, starting at the beginning of the execution and converging to the last event of the computation. We can determine the longest path between the beginning and the end of the computation, which we call the *longest causality chain*. The longest causality chain is an indication of the number of events that have to be executed in sequence. The shorter the chain, the more parallelism we can observe.

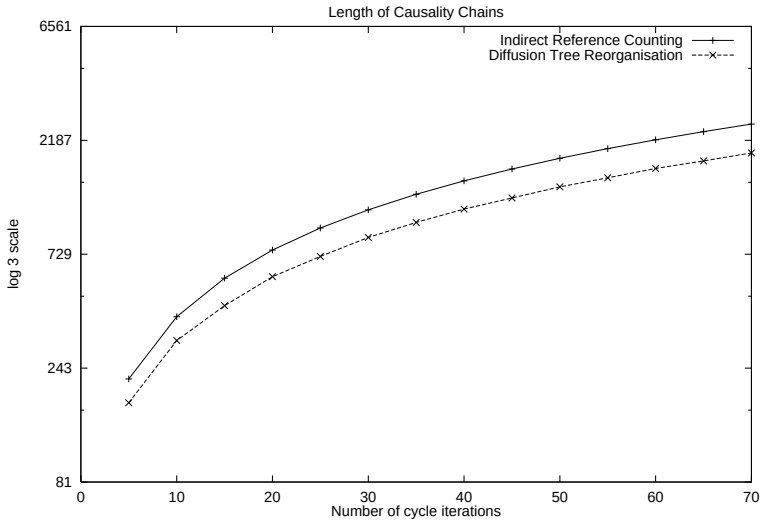


Figure 11. Length of the longest causality chain for cycle with 3 processes

Figure 11 displays the difference between the lengths of the longest causality chains in the two algorithms. In order to show that this difference is proportional to the number of processes involved in the computation, the y axis scale is expressed as the logarithm of the number of hosts. We can see that the space between the curves remains constant as the number of iterations increases.

The second benchmark evaluates the performance of action queue strategies. In order to gain some intuition, Figure 12 displays the messages that were sent for two runs of the diffuse program with (a) a fifo action queue or with (b) a sorted action queue. The DGC protocol used INC_DEC messages; a similar result is obtained with

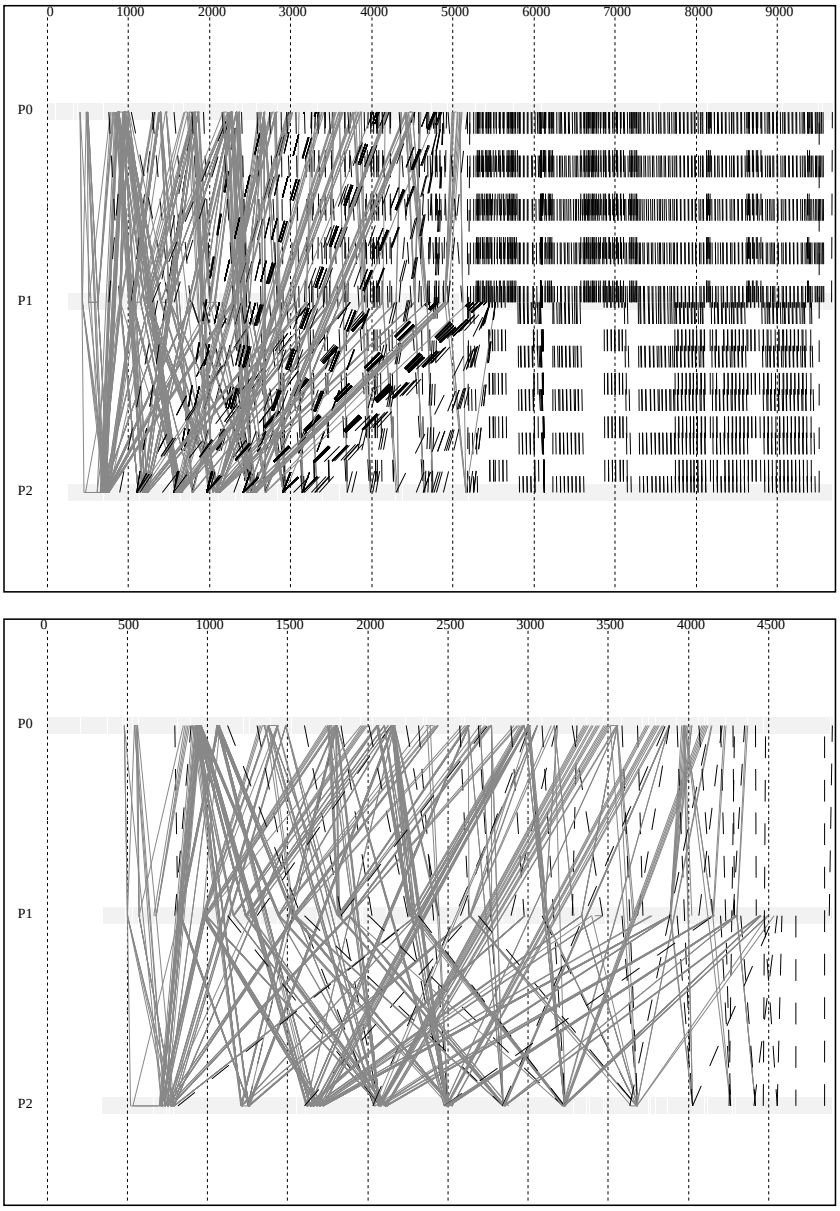


Figure 12. The diffuse benchmark with (a) fifo and (b) sorted action queues; one unit is 1ms

A coloured version of these figures is available from
`http://www.ecs.soton.ac.uk/~lavm/papers/hosc01-colour.tar.gz`.

IRC, because we assumed that w^d is large compared to the number of processes: as a result, the number of INC_DEC messages is of the same order as the number of processes, and therefore negligible.

The mutator duration is in the range 4500–5000 ms for each run. However overall execution ends when the last pointer is reclaimed, which identifies the detection of the distributed termination. In the case of fifo queues, it takes about 10000ms while it only takes 5000 ms with ordered queues. This phenomenon is explained by the dramatic reduction of the number of DGC messages when sorted action queues are used.

The following table gives a small sample of the number of messages exchanged during runs of the diffuse program (again with three processes). The column *mutator* indicates the number of messages exchanged by the mutator, *fifo* and *sorted* respectively indicate the number of DGC messages using the fifo and sorted action queues.

depth	width	mutator	fifo	sorted
6	3	1092	732	104
6	4	5460	3640	394

As the width increases, DGC pointers are more frequently sent to processes to which they were previously sent. Consequently, at any given moment, more DGC messages are waiting to be sent to a same destination. The sorted queue strategy takes opportunity of this situation to group several messages into a single one to a same destination.

The table shows that increasing the width causes a fivefold increase in the number of messages sent by the mutator. The same increase is also observed in the number of messages sent by the fifo strategy; as a result, the ratio of DGC messages to mutator messages remains high and constant to 66%. On the other hand, the sorted queue strategy generates less messages and only has a fourfold increase: the same ratio diminishes from 9% to 7%.

7. Related Work

Literature on distributed garbage collection is abundant; we refer to Jones' book [14] for a complete chapter on the subject and the garbage collection home page [13] with more than 1600 references. Section 3 gave an overview and a discussion of the current work on distributed reference counting. Two different approaches however deserve a further comparison. Shapiro, Dickman and Plainfossé [30, 29] were the first to study the problem of third party dependency. However, we first draw our attention to Dickman who has independently published an algorithm for restructuring diffusion trees [7] since the first publication of our algorithm.

Dickman's algorithm assumes a model of communication based on remote method invocation, and therefore some implementation details are specific to this model of communication. In this section, we compare the essence of his approach with ours.

Dickman proposes two techniques to reorganise diffusion trees, with the effect of reducing third party dependencies (Piquer's zombie pointers).

The first one consists of rerooting a subtree when the root of the subtree performs a remote method invocation to the owner (the root of the diffusion tree). The request for rerooting is piggy-backed on the method call. When the result is returned, the owner's reference counter is incremented, and the owner is made the parent of the process that initiated the call. A DEC message has to be sent by the rerooted process to its previous parent, in order to decrease its previous parent's reference counter. It is important to observe that the DEC message can only be sent after the termination of the remote method invocation, otherwise race conditions may arise similarly to the naïve extension of reference counting. Consequently, long processing during the remote method invocation will potentially prevent the earlier release of resource on the previous parent. Similarly to Dickman's algorithm, we could piggy-back our INC_DEC message to a remote method invocation, but our DEC message may be sent earlier, allowing an early release of resources. Furthermore, our tree rerooting technique may proceed even though no remote method invocation is performed on the owner.

The second technique proposed by Dickman reduces the depth of the tree in a less radical manner. When a process holding a reference to an object receives a new copy of the reference, IRC and our algorithm dictate that it should send a DEC message to the sender of the message. Instead, choosing between the previous parent and the message sender, Dickman sends a DEC message to the process that has the deeper depth, in effect grafting the current subtree to the process with the shallower depth. The realisation of this algorithm turns out to be non trivial because depths of processes are not propagated instantaneously, and therefore the decision to change parent may be made on possibly out-of-date depth information. We have shown that tree rerooting introduces more parallelism by reducing the longest chain of causality; Dickman's tree structuring reduces chains of causality, but not as much as tree rerooting. Therefore, besides the risk of increasing the chain by a wrong decision to restructure the tree, his technique does not reduce the cost of DGC as radically as ours. However, we believe that it still has some use when the owner is not directly reachable, because for instance hidden behind a firewall. Our algorithm describes a spectrum of algorithm from eager activation of tree rerooting to its lazy activation which leads to indirect reference counting. Dickman introduces a further dimension by allowing a choice in the tree re-structuring: if the reorganisation is closer to the root, it allows more parallelism in the DGC activity and it reduces the number of third party dependencies.

Shapiro, Dickman and Plainfossé [30, 29] address the problem of distributed garbage collection for mobile objects. Our approach separates distributed reference counting from the problem of object migration, for which we devised a specific algorithm [22]. Our comparison therefore focuses on non-mobile objects. They introduce the notion of SSP (stub-scion pair) chains, which essentially is their representation of diffusion trees. They allow tree rerooting, which they call *chain short-cutting*, through a method similar to Dickman's first technique, which piggybacks information to a method invocation to the owner. However, unlike Dickman's

algorithm, a DEC is not immediately sent to the previous parent, but this action is left to the distributed garbage collector adopting a specific protocol to clear unused stubs and scions. Their approach suffers from the same drawbacks as Dickman's algorithm because tree rerooting remains dependent on the mutator activating remote method call on the owner. Baggio [1] showed that SSP chains could adapt to the network configuration. In the absence of direct connectivity, e.g. behind a firewall, short-cutting does not have to take place. A similar flexibility is also present in our algorithm because tree rerooting is optional.

8. Conclusion

The garbage collector we present in this paper is based on distributed 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 [31] 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. Similarly, groups [25] have a hierarchical organisation and can be reference counted.

This paper concludes the investigation of an algorithm for distributed garbage collection based on reference counting. This algorithm has been specified, and its correctness has been proved mechanically. In this paper, we have described a complete implementation and evaluated some of its performance aspects. The conclusion of our experiments is that tree rerooting offers more parallelism in the DGC as causality chains become shorter and that grouping DGC messages reduces DGC traffic. A complete performance evaluation would require real-world applications using distributed GC, but we currently lack such applications. Future work concerns a Java implementation of the algorithm, which would provide a multi-lingual environment, using Nexus/Globus as a communication layer and the NexusRMI stub compiler [4]. In order to support garbage collection of mobile objects, a further study is required to understand the interactions between tree rerooting and object migration [22].

Acknowledgements

The author is grateful to Richard Jones, whose careful reading and suggestions led to various improvements of the technical contents and presentation of the paper. The author also wishes to acknowledge Danius Michaelides, Christian Queinnec, and the anonymous referees for their useful comments.

Notes

1. Safe approximation means that if an object is globally reachable, then it is also globally reachable according to the safe approximation. As a result, when the implementation deallocates an object, it is guaranteed that it is no longer globally reachable.

2. In general, DGC activity cannot be measured independently of the local collector activity. However, in the particular case of the cycle program using indirect reference counting, it is known that once the reference counter of a pointer p_1 becomes zero, the pointer p_2 pointed by p_1 has lost its last active reference; therefore a DEC message may be sent immediately for p_2, without waiting for the finalizer activation. Therefore by using knowledge that is specific to the problem, some benchmarks may be improved. However, we did not implement such a variant because it does not remain valid for other DGC strategies.

References

1. Aline Baggio. System support for transparency and network-aware adaptation in mobile environments. In *ACM Symposium on Applied Computing special track on Mobile Computing Systems and Applications*, Atlanta, Georgia, 1998.
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. 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. Fabian Breg and Dennis Gannon. Compiler support for an RMI implementation using Nexus-Java. Technical report, Indiana University, 1997.
5. George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.
6. Peter Dickman. Optimising Weighted Reference Counts for Scalable Fault-Tolerant Distributed Object-Support Systems, 1992.
7. Peter Dickman. Diffusion Tree Redirection for Indirect Reference Counting. In Tony Hosking, editor, *Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM.
8. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University, 1993.
9. 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.
10. H.-J.Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software – Practice and Experience*, 18(9):807–820, 1988.
11. 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.
12. *Java Remote Method Invocation Specification*, November 1996.
13. Richard Jones. The Garbage Collection Page. `http://www.cs.ukc.ac.uk/people/staff/rej/gc.html`.
14. Richard Jones and Rafael Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.
15. C.-W. Lermen and D. Maurer. A Protocol for Distributed Reference Counting. In *Lisp and Functional Programming*, pages 343–354, 1986.
16. Luigi V. Mancini and S. K. Shrivastava. Fault-tolerant reference counting for garbage collection in distributed systems. *Computer Journal*, 34(6):503–513, December 1991.
17. Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Amsterdam, 1989. Elsevier Science Publishers.
18. Danius Michaelides, Luc Moreau, and David DeRoure. A Uniform Approach to Programming the World Wide Web. *Computer Systems Science and Engineering*, 14(2):69–91, 1999.
19. Luc Moreau. Correctness of a Distributed-Memory Model for Scheme. In *Second International Europar Conference (EURO-PAR'96)*, number 1123 in Lecture Notes in Computer Science, pages 615–624, Lyon, France, August 1996. Springer-Verlag.

20. 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)*, pages 204–215, September 1998. Also in *ACM SIGPLAN Notices*, 34(1):204-215, January 1999.
21. Luc Moreau. Hierarchical Distributed Reference Counting. In *Proceedings of the First ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, pages 57–67, Vancouver, BC, Canada, October 1998. Also in *ACM SIGPLAN Notices*, 34(3):57–67, March 1999.
22. Luc Moreau. Distributed Directory Service and Message Router for Mobile Agents. *Science of Computer Programming*, 39(2–3):249–272, 2001.
23. 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.
24. Luc Moreau and Jean Duprat. A Construction of Distributed Reference Counting. Technical Report RR1999-18, Ecole Normale Supérieure, Lyon, March 1999.
25. 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.
26. Luc Moreau, Victor Tan, and Nicholas Gibbins. Transparent Migration and Ownership of Mobile Agents. Technical report, University of Southampton, 2000.
27. José M. Piquer. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In *Parallel Architectures and Languages Europe (PARLE'91)*, pages 150–165, 1991.
28. José M. Piquer. Indirect Distributed Garbage Collection: Handling Object Migration. *ACM Transactions on Programming Languages and Systems*, 18(5):615–647, September 1996.
29. 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.
30. Marc Shapiro, Peter Dickman, and David Plainfossé. SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection. Rapport de Recherche 1799, INRIA-Rocquencourt, November 1992.
31. 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.
32. P. W. Trinder, K. Hammond, J. S. Mattson, A. S. Partridge, and S. L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 79–88, 1996.
33. Peter Van Roy. On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart. In *Fourth International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDCSIA 99)*, Sendai, Japan, July 1999. World Scientific.
34. 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.

Received Date
Accepted Date
Final Manuscript Date