Implementation and Performance Evaluation of a Distributed Garbage Collection Algorithm

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

We have recently described an algorithm for distributed garbage collection based on reference-counting [14,13,15]; the algorithm describes a spectrum of algorithms according to the policy used to manage messages. In this paper, we describe the implementation of the algorithm and evaluate its performance. We have implemented two policies, which are extremes of the spectrum. The first one uses INC_DEC messages, whose effect is to reorganise the diffusion tree, whereas the other one does not use such messages, which in effect results in Piquer's indirect reference counting [17]. In addition, two different strategies for managing action queues have been implemented. The conclusions of our experimentations are the following. Using INC_DEC messages potentially offers more parallelism in the DGC activity; this phenomenon can be measured by shorter causality chains than with indirect reference counting. Grouping messages per destination dramatically reduces the number of messages to be sent, though requires a more complex implementation as messages have to be sorted per destination.

1 Introduction

Over the last decade, distributed symbolic[a] 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 [20] able to activate methods on remote objects.

In particular, RMI provides a distributed garbage collector (also written distributed GC or even 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 another algorithm for distributed garbage collection [13]. This algorithm based on distributed reference counting was developed and prototyped as part of NeXeme [14], a distributed implementation of Scheme, based on the message-passing library Nexus [5]. This algorithm has been studied in details, and mechanical proofs of safety and liveness were

[a] By symbolic, we mean non-numeric computing.

carried out using the proof assistant Coq [15]. The algorithm in fact describes a family of algorithms and has several optimisations. The focus of this paper is a real implementation of the algorithm and a study of performance of its variants and optimisations.

The algorithm was implemented in C and is using the Nexus library [5] for communications. Local garbage collection is handled by Boehm and Weiser's collector [7]. 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 [3], it would give access to a multi-language garbage-collected distributed environment.

This paper is organised as follows. We briefly sketch our distributed garbage collection algorithm in Section 2. The overall implementation design is presented in Section 3. Some benchmark programs are described and then performance of the algorithm is evaluated in Section 4. Finally, a brief comparison with related work concludes the paper.

2 A Family of Algorithms

Our algorithm has been presented using graphical representations and analysed at length in other papers [13,14,15]. In addition, the formal specification of the algorithm in Coq is accessible from the following URL: `http://www.ecs.soton.ac.uk/~lavm/coq/drc`. Therefore, in this section, we only present the key features of the algorithm.

At a very abstract level, we deal with a notion of *pointer* (network pointer [2] or global pointer [5]). We call *owner* the node where the data referred to by a pointer resides. Pointers can be communicated between nodes by remote method invocation. The purpose of the distributed garbage collector is to ensure that the data a pointer is referring to on its owner cannot be reclaimed as long as live instances of the pointer remain in the distributed environment.

In order to achieve this goal, we use reference counters, and we assume that all participating nodes maintain a pair of tables, called *send-table* and *receive-table*. In addition to regular messages transporting remote method invocations, two new messages are introduced to maintain accurate reference counters. Reference counters, send and 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).

S: Every time a pointer is sent to a remote node, its associated counter is incremented by one. If not present, the pointer is entered in the send table of the emitter node.

R1: If a node receives a pointer sent by its owner, the pointer is entered in the receive table, which contains the set of live pointers on the node.

GC: Once a pointer becomes garbage on a node, the pointer is removed from the receive-table, and a decrement message DEC is sent to the pointer's owner.

DEC: When a node receives a DEC message pertaining to a pointer, it decrements the counter associated with the pointer; if the counter reaches 0, the pointer is removed from the send-table.

R2: If a node receives a pointer sent by any emitter and if the pointer is already live on the node, then a DEC message is sent back to the emitter.

R3: If a node receives a pointer not sent by its owner and if the pointer is not live on the node, the pointer is entered in the receive table, then an INC_DEC message is sent to the owner, containing a reference to the emitter.

INC: When a node receives an INC_DEC message referring to pointer and a third node, it increments the counter associated with the pointer in its send-table and it posts a DEC message to the third node.

The algorithm also assumes in-order message delivery between any pair of nodes. In order to ensure safety and liveness, send-tables but not receive-tables are defined as roots of local garbage collectors. The algorithm is safe because a pointer will remain present in its owner's send-table as long as a reference remains active remotely. Consequently, the data associated with the pointer will not be reclaimed by the local garbage collector, because the send-table makes the pointer accessible from the local GC roots.

The key idea of the algorithm is the INC_DEC message followed by the DEC message, whose effect is to safely propagate reference counter values from internal nodes of the diffusion tree to the owner (which is the root of the diffusion tree for the current pointer). This allows internal nodes that no longer use a pointer to safely reclaim the resource used by the pointer.

Optimisations are possible. First, messages may be delayed and grouped. Second, R3 immediately followed by GC, may be optimised by a single DEC message to the node that sent the pointer. Our algorithm in fact defines a spectrum of algorithms[15]. Eager activation of R3 tends to flatten the diffusion tree, whereas delaying INC_DEC messages as much as possible leads to Piquer's indirect reference counting[17].

3 Implementation Design

Our implementation of the algorithm relies on two libraries for communications and local garbage collection. We use Nexus [5], a message-passing library for communications, and we use Boehm and Weiser's local garbage collector [7]. Boehm's 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 [19] at the heart of our distributed Scheme, NeXeme [14]. A major design concern was to make the implementation independent of these libraries.

3.1 The Communication Library

In this Section, we present the characteristics of the communication library that is expected by our implementation. They are heavily influenced by Nexus[5], but they form the essence of a distributed object system. A computation executes on a set of *nodes* and consists of a set of *threads*, each executing in an address space called a *context*. (For the purposes of this article, we do not make the distinction between a node and a context, which we regard as equivalent to a process.) An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context.

"Host pointers" and a mechanism for initiating remote computations represent the essence of a distributed object system. A host pointer is a network representative of an object; it may also be found under the terminology startpoint/endpoint pair or network pointer [2]. Remote method invocation (or remote service request [5]) is typically the mechanism used for activating remote computations.

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 transfered to the context 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 pointed specific data are available to the handler.

In the right-hand side of Figure 1, we see a simple representation of the heap managed by the communication library. In Nexus, a host pointer is formed of a startpoint-endpoint pair. An endpoint points at some user data and is associated with a handler table; a startpoint is bound to an endpoint and may be copied to remote heaps.

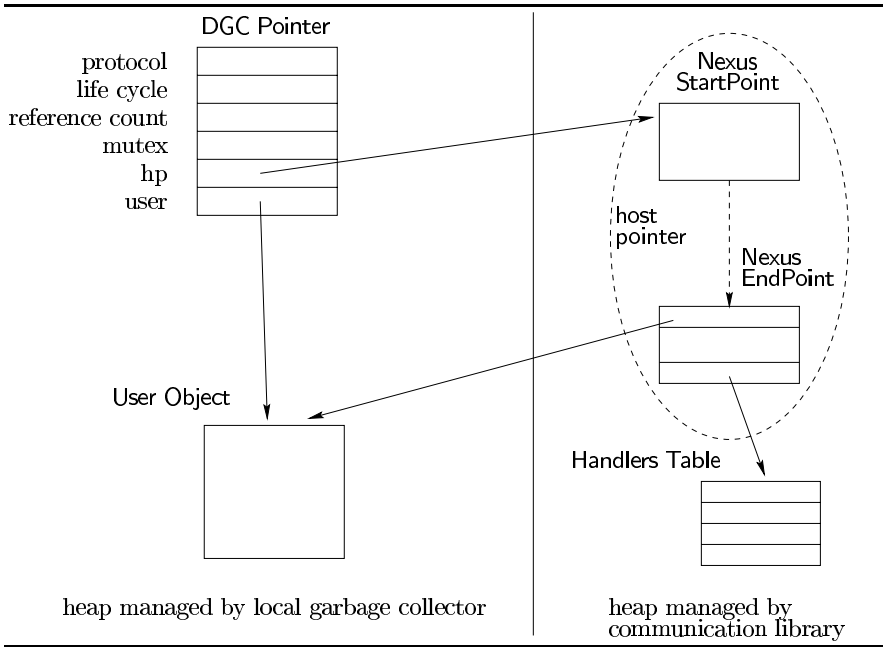

Figure 1: Objects, Communication Pointers, and DGC pointers

3.2 Distributed GC Pointers

Users allocate objects in a garbage collected heap, they create host pointers pointing to these objects, and they communicate these host pointers to remote hosts. The goal of the distributed garbage collector is to ensure that objects allocated in the garbage collected heap are not going to be reclaimed as long as host pointers pointing at them are accessible remotely.

Host pointers may be regarded as network representatives of user objects. We introduce a new kind of objects, called *DGC pointers*, which are garbage-collected network representatives of user objects.

A DGC pointer is a data structure allocated in the garbage-collected heap, containing several fields:

- a user pointer, which points at the user data if the DGC pointer is located on the owner;
- a host pointer, which points, in the case of Nexus, at a communication startpoint;

- a reference counter indicating the number of times that the pointer has been exported to a remote context;
- a mutex to ensure exclusive access to the DGC pointer content;
- information about the life cycle, to be explained below;
- the distributed garbage collection protocol handling this object.

DGC pointers have a life cycle composed of three states:

1. *live*: When a DGC pointer is created, it is said to be live, and it remains so, as long as it is accessible from a root of the local garbage collector.
2. *phantom*: The local garbage collector is responsible for detecting when a DGC pointer becomes garbage. Finalization changes the pointer state to "phantom". During this new stage, the pointer is no longer used (not even reachable) by the application, but remains active so that distributed GC information can be propagated.
3. *dead*: Once the DGC has updated all its information related to a DGC phantom pointer, the local garbage collector can detect its inaccessibility, for a second time; then, the DGC pointer can again be finalized and it enters the new phase "dead". In this third phase, the host pointer associated with the DGC pointer is deallocated explicitly.

3.3 DGC protocols

All DGC pointers have a three-stage life cycle, but the 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 an explicit field of a DGC pointer. Protocols are also represented by a data structure whose fields are represented in Figure 2.

When a message containing a DGC pointer arrives to a node, the DGC protocol has to determine whether the pointer is present locally; for this purpose, it uses a "*receive table*" containing all live pointers. Similarly, it maintains a "*send table*" containing all the pointers that were sent remotely. Note that reference counters are directly associated with the pointers themselves. The send table is a root of the local garbage collector but the receive table is not. Such an organisation guarantees that only objects that are sent remotely remain accessible to the local garbage collector.

The local garbage collector plays an essential role in our implementation because it detects when a DGC pointer becomes garbage; during finalization, it moves 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, operations that may be long and require a lot of memory. It is not suitable to perform these operations during the finalization itself; indeed, finalization is usually performed inside the garbage collector critical section, and one prefers to quit this section as quickly as possible to avoid starving other threads running in parallel and requiring more memory. Instead, finalized DGC pointers are entered in a finalization queue.

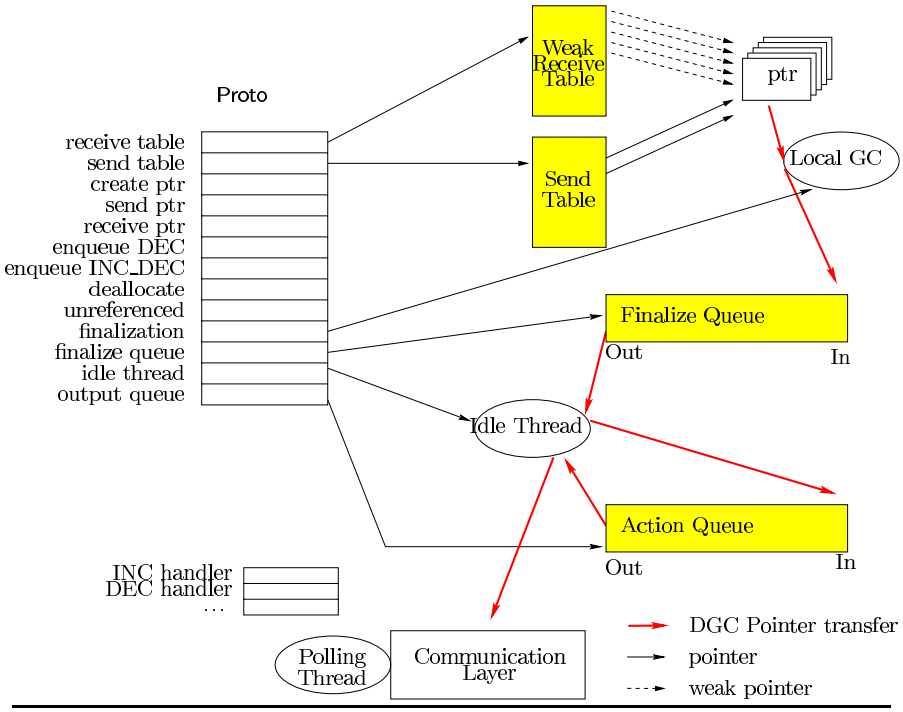

Figure 2: Protocol and associated data structures

A dedicated thread, called the *idle tread*, is responsible for transferring pointers from the finalization queue into an action queue. 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[b], DGC messages are not immediately sent, but enqueued in the action queue. The same idle thread is also responsible for activating items from the action queue; this operation typically requires calling a procedure of the communication layer.

The last active element in this picture is the polling thread of the communication library. It 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 nodes.

The protocol datastructure 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, in a single environment, pointers managed by several protocols running at the same time.

3.4 Implemented Protocols

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

1. *The algorithm that uses* INC_DEC *messages* (as described in Section 2).

   If a DGC pointer p is received on a node, and if p is not a live pointer present in the 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 on a node, and if p is live and present in the 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)* The activity of receiving a pointer p typically occurs when the mutator proceeds; by delaying the sending of a DGC message, we favour the mutator over garbage collection activity. *(ii)* By managing all DGC messages in a same queue, we may optimise them, as explained in Section 3.5.

[b] The term "mutator" is usually used to denote the "useful" component of an application, in contrast to the collector that deals with collecting garbage.

2. *Indirect reference counting*[17].

   In this protocol, INC_DEC messages are not used. DGC pointers now have an extra field, called *emitter*, which contains a reference to the node that emitted the pointer. If a DGC pointer p is received on a node, and if p is not a live pointer present in the receive-table, then the emitter field for p is set to the node 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 node that initially emitted the pointer.

In addition, we also provide DGC pointers that are not reference counted; they are handled by a third protocol:

3. *The null protocol*

   The null protocol does not maintain reference counters for pointers. In our implementation, nodes are denoted by null protocol pointers. Such pointers are added to every communication so that recipients can be informed of the message origin node.

Each protocol supports an *unreferenced* method (similar to the unreferenced method on RMI object in Java[20]). This method is called *every time* a reference counter reaches the value zero.

When a pointer's reference counter reaches the value zero, the pointer is removed from the send-table; the pointer then may still be live, if used locally, or if the local garbage collector has not detected yet it has become garbage. The unreferenced method must be distinguished from finalization; a finalizer is executed only once, when the pointer is inaccessible, whereas the unreferenced method may be invoked as many times as the reference counter reaches zero.

3.5 Implemented Action Queues

The policy to handle messages is independent of the protocol. We present here two policies to handle action queues. Currently, the algorithm supports three types of actions:

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, it sends a message to decrease the reference counters of several pointers, by an amount given for each pointer.

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.

3. *Deallocating a pointer*. This action consists of deallocating all resources used by a host pointer in the heap managed by the communication library.

A safety condition of the algorithm is that DEC messages cannot overtake INC_DEC messages if they are related to the same pointer. A further constraint is that deallocation of a pointer cannot take place before the last message to that pointer has been emitted.

The action queue is a data structure containing actions; actions may be entered in or retrieved from an action queue. Any implementation strategy is valid for action queues, as long as it preserves the safety conditions set by the algorithm. We have implemented two variants of the action queue:

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.

2. *The sorted action queue.*

   Each action for sending a DEC message entered in the queue is merged with a similar action related to a same pointer. Therefore, actions for sending DEC messages are associated with a counter 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 node s), then these messages can replaced by a single DEC message to s directly.

4 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 used; second, we present our results.

4.1 Benchmark Programs

The scientific programming community has developed a series of benchmarks for sequential and parallel languages. Similarly, the Lisp programming community has been using Gabriel's benchmarks [6], and Feeley [4] 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. Therefore, we have devised two benchmarks that exhibit the properties of our implemented policies: *cycle* highlights the effect of INC_DEC messages, whereas *diffuse* shows the benefits of the sorted action queue.

Each of our benchmarks was specifically designed to evaluate a given feature of the implementation. Unfortunately, we also lack real applications that evaluate the overall effect of the algorithm. We are confident that such applications will become available as platforms such as Java and RMI get widely used.

Cycle

The first benchmark is aimed at measuring the benefit of reorganising the diffusion tree, using INC_DEC messages. In theory, we would consider a very high number of nodes. The first node allocates a DGC pointer, passes it to the second node, which in turn passes it to the third one, and so on. The result is a very unbalanced diffusion tree. In practice, we consider a number of nodes N, forming a circle, where the successor of node $N-1$ is node 0. In order to avoid sending a pointer p to a node that has already received it, we create a new DGC pointer p', every $N-1$ steps; p' is defined so as to point to the current pointer p: this organisation ensures that p remains live as long as p' is live. Then, we repeat the process with p'.

Figure 3 displays an example of execution. It displays the timelines of three nodes participating to the computation; one unit in Figure 3 represents $100\mu S$. Plain (or red[c]) lines represent mutators messages carrying pointers. We see that a pointer is passed from process 2 (label A) to process 1, and then to process 0, where a new pointer is created (label C), passed to 2, and then 1, and so on.

Dashed (or blue) lines represent INC_DEC messages, whereas dotted (or green) lines denote DEC messages. We also see that when process 0 receives the first pointer, it sends an INC_DEC message (label E) to process 2, its owner, which in turn sends a DEC message (label F) to process 1. The potential benefit

[c] A coloured version of this document is made available from `http://www.ecs.soton.ac.uk/~lavm/papers/pdcsia99-colour.ps.gz`.

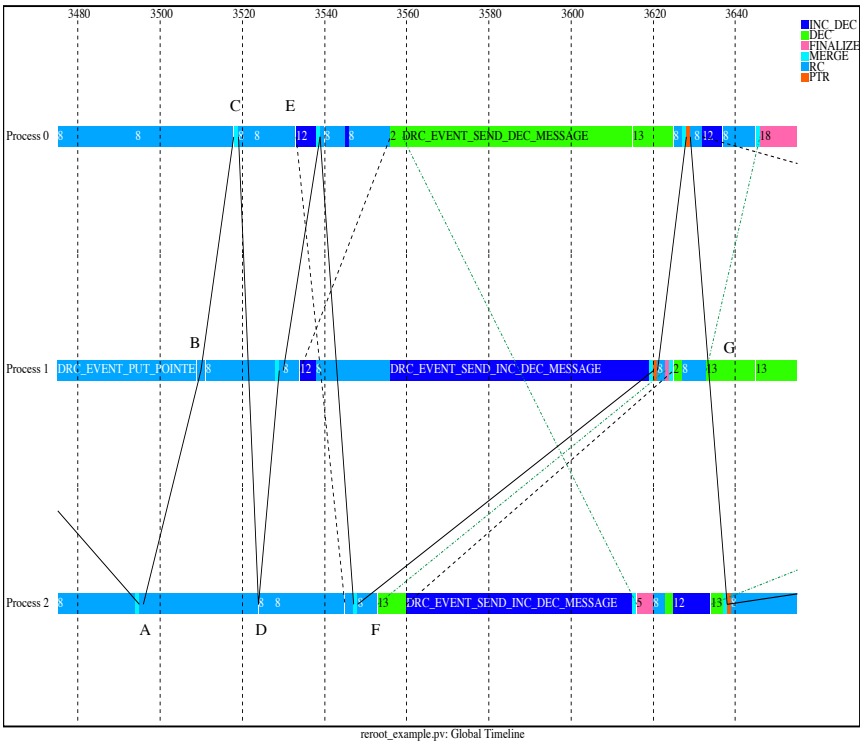

Figure 3: Diffusion tree reorganisation

of this reorganisation is that the pointer is no longer needed on process 1, it may be finalized and then a DEC message may be sent from process 1 to 0 (label G, at time 3635).

This example illustrates that the reorganisation of the diffusion tree avoids "zombie pointers" [17], whose presence is only required by the garbage collector in order to correctly maintain reference counters.

In fact, 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 node. Using the INC_DEC message resets the counter for that pointer, which will then be reclaimed.

In such a benchmark, interesting measures that can be taken concern the *(i)* number of messages, *(ii)* total time (including time to cleanup), *(iii)* length of causality chain (to be explained in Section 4.3).

Diffuse

In the cycle program, the mutator's activity is sequential, as it propagates one pointer node after node. 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, which is measuring the benefit of grouping DGC messages per destination.

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

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

A variant of this algorithm consists of executing several pointer diffusions in parallel in order to measure the reuse of DGC messages for different pointers.

The first three interesting measures are similar to the ones for circle, the last two are specific to this benchmark: *(i)* number of messages, *(ii)* total time (including time to cleanup), *(iii)* length of causality chain (to be explained in Section 4.3), *(iv)* counters values in a DEC message, *(v)* number of DGC pointers in a single DEC message.

4.2 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. Some of our benchmarks even use random number generators.

The absence of a global clock has also influenced the design of our benchmarks. When we wish to measure a time duration, we made sure that the measures were taken on a single node. Some of our graphical visualisations

display timelines for several nodes; some of these timelines may have to be shifted with respect to each other because time 0 is not defined globally.

It is also quite difficult to measure the exact time spent on various DGC sub-activities due to preemption.

In addition, there are issues that are specific to garbage collection. There is a strong partnership between distributed garbage collectors 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[d].

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 purpose is to allocate garbage to be sure that objects that are relevant to our experiments get finalized.

4.3 Comparison

Figure 4 displays the messages that were sent for two runs of the cycle programs (a) using INC_DEC messages or (b) using indirect reference counting (IRC). Dark (red) lines represent mutators messages, whereas light lines (blue or green) represent DGC messages. 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. 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 3, DGC activity is interleaved with the mutator computation when INC_DEC messages are used. A very high number of DGC messages may be propagated at a very early stage *in parallel*.

[d] 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.

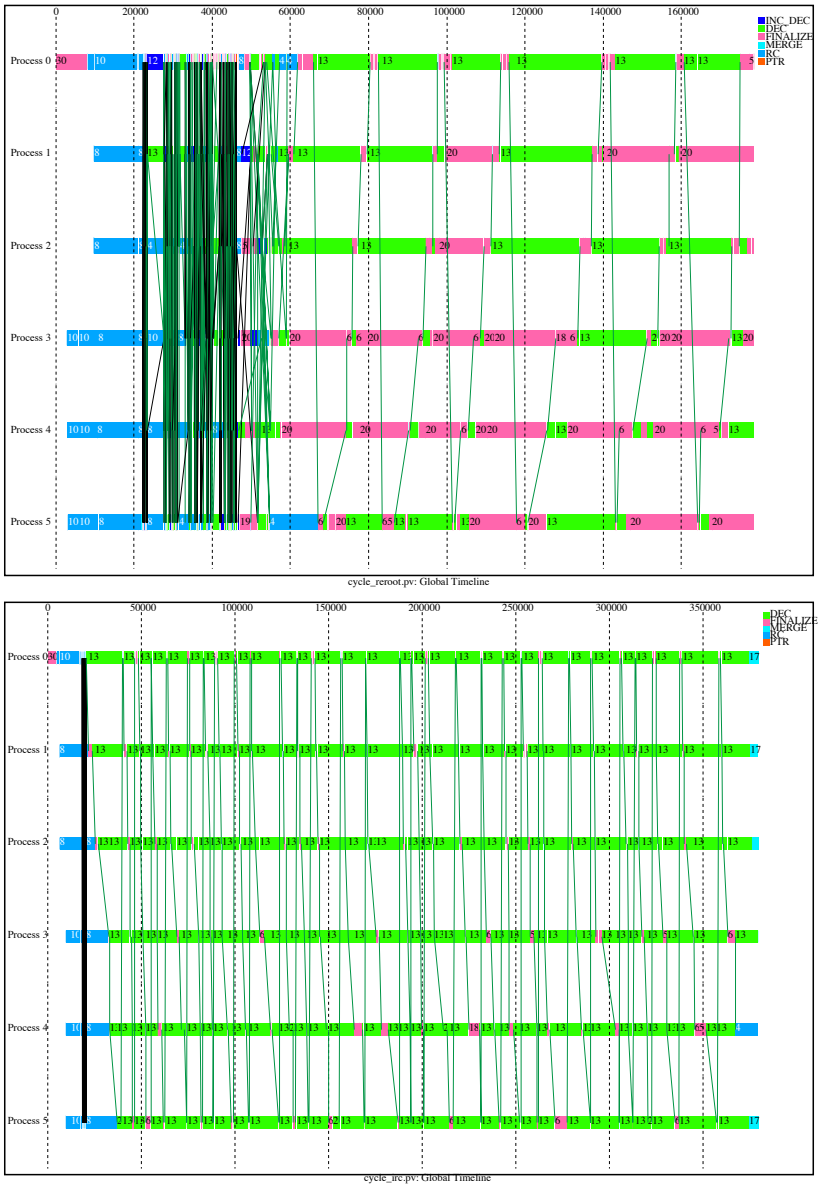

Figure 4: The cycle benchmark with (a) INC_DEC messages (b) IRC

Let us consider the situation where n iterations of the circle have been executed, and let us imagine that all INC_DEC and subsequent DEC messages have been propagated. We then obtain a situation such that for each circle there remains a single instance of a DGC pointer that is live. Consequently, the number of DEC messages that remains to be propagated is given by the number of circles that were completed, as opposed to IRC where the number of DEC messages is given by the number of circles multiplied by the number of nodes in a circle.

The benchmark was designed so as to model mobile computations, hopping from nodes to nodes. In theory, we should therefore consider a very high number of nodes. In practice, we consider a smaller number, but artificially create new pointers that we propagate in a circular manner. Therefore, if we push this example to the limit of a very high number N and a single circle, in case (a), after all INC_DEC and DEC messages were propagated, there remains only 1 DEC message to send, whereas in case (b), N messages still have to be defined.

Figure 4 also shows that using INC_DEC message potentially reduces the total duration of the benchmark; in this figure, 1 unit is $100\mu s$ and execution duration is halved when using INC_DEC messages. It should be noted that in case (a), about three times more messages were exchanged than in (b), but messages could be propagated in parallel.

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 INC_DEC messages. Causality[12] 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:

1. on a given node, 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' on the same host.

The causality relationship is a partial order from which we can derive a directed acyclic graph, starting in t_0 and converging to the last event of the computation. We can determine the longest path from t_0 to the end of the computation, which we call the *longest causality chain*. The longest causality

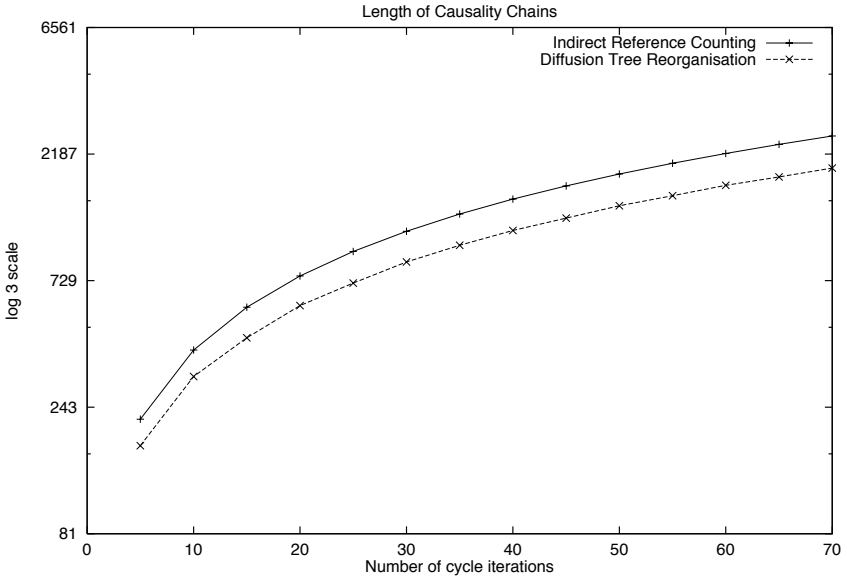

Figure 5: Length of the longest causality chain for circle with 3 nodes

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 5 displays the length of such chains for different runs of the cycle program.

The measure that is relevant to us is the difference between the length of the longest causality chains in the two algorithms. In order to show that this difference is proportional to the number of nodes involved in the computation, the y axis scale is expressed in the logarithm of the number of hosts. We can see that the space between the curves remains constant as the number of iterations increases.

Figure 6 displays the messages that were sent for two runs[e] of the diffuse program with (a) a fifo message queue or (b) with a sorted message queue. We

[e]The DGC protocol used INC_DEC messages but an exactly similar result is obtained with IRC. The reason being that in the presence of only three nodes, we are never in the situation of sending an INC_DEC message.

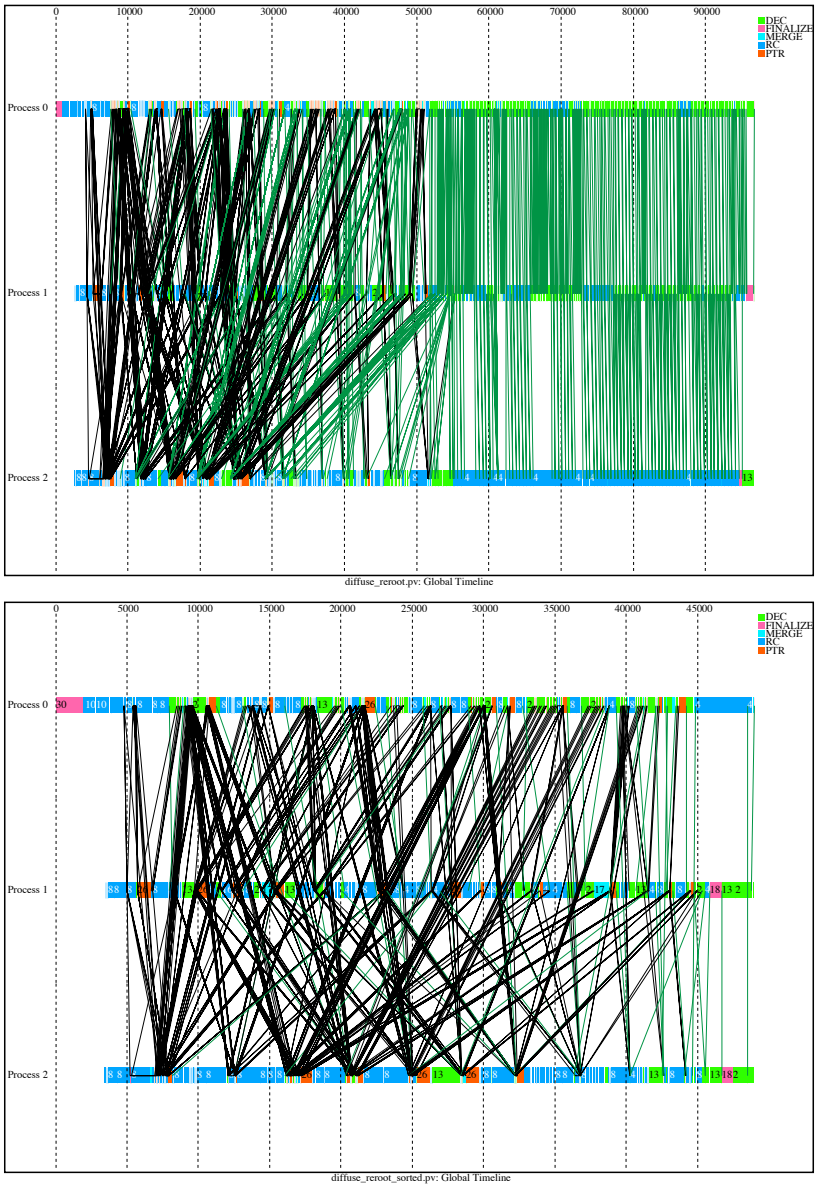

Figure 6: The diffuse benchmark with (a) fifo and (b) sorted message queues

can observe a dramatic reduction of the number of DGC message exchanges when sorted message queues are used. In addition, overall execution time decreases as less time is required for communications.

The following table gives a small sample of the number of messages exchanged during some runs of the diffuse program. 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, the frequency at which pointers are sent to nodes where they were previously sent also increases. Consequently, the sorted queue has got more opportunities to optimise the grouping of messages to a same destination.

5 Related Work and Conclusion

Literature on distributed garbage collection is abundant; we refer to Jone's book[10] for a complete chapter on the subject and the garbage collection page[9] with more than 1600 references. In previous papers[13,15], we have covered the differences between our algorithm and other reference counting or reference listing approaches such as Birrel's[2,18], Lermen and Maurer's[11,21], Piquer's[17], or Bevan's[1].

In this paper, we have focused on the impact of our algorithm on communications. The distributed version of the train GC[8] uses an algorithm to keep track of live pointers. Their algorithm requires less messages than ours; however, the information that is made available is not the same because, for instance, the number of remote live copies or their location (in the case of reference listing) is not made available to the owner of a pointer. A similar discussion also hold for Weighted Reference Counting[1].

The garbage collector we present in this paper is based on reference counting. As other reference-counting algorithms, ours is unable to reclaim distributed cycles. However, we should observe that there is a range of applications that do not create distributed cycles. In particular, Tel and Mattern[21] have shown that the problem of termination in distributed systems is equivalent to distributed GC. Reference counting can be used because processes form a hierarchy. Groups[16] also have a hierarchical organisation and can be reference counted.

This paper concludes an investigation about 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. A complete performance evaluation would require real-world application using distributed GC, but we currently lack such applications. Future work concern 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 [3].

Acknowledgements

Thanks to Christian Queinnec and Danius Michaelides for their comments. This research was partially funded by EPSRC grant GR/M84077.

1. David I. Bevan. Distributed Garbage Collection using Reference Counting. In *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer-Verlag, June 1987.
2. 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.
3. Fabian Breg and Dennis Gannon. Compiler support for an RMI implementation using NexusJava. Technical report, Indiana University, 1997.
4. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.
5. 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.
6. Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. Series in Computer Science. MIT Press, Cambridge, Massachusetts, 1985.
7. H.-J.Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software – Practice and Experience*, 18(9):807–820, 1988.
8. 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.
9. Richard Jones. The Garbage Collection Page. `http://stork.ukc.ac.uk/computer_science/Html/Jones/gc.html`.

10. Richard Jones and Rafael Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management.* Wiley, 1996.
11. C.-W. Lermen and D. Maurer. A Protocol for Distributed Reference Counting. In *Lisp and Functional Programming*, pages 343–354, 1986.
12. 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.
13. 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.
14. 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.
15. Luc Moreau and Jean Duprat. A construction of distributed reference counting. Technical Report RR1999-18, Ecole Normale Supérieure, Lyon, March 1999.
16. 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.
17. José M. Piquer. Indirect Distributed Garbage Collection: Handling Object Migration. *ACM Transactions on Programming Languages and Systems*, 18(5):615–647, September 1996.
18. 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.
19. Manuel Serrano. *Vers une compilation portable et performante des langages fonctionnels.* PhD thesis, Université Paris VI, December 1994.
20. Sun MicroSystems. Java Remote Method Invocation Specification, November 1996.
21. 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.