Recent Advances in Distributed Garbage Collection

Marc Shapiro Fabrice Le Fessant*

Paulo Ferreira†

INRIA Projet SOR
Domaine de Voluceau, Rocquencourt, B.P. 105
78153 Le Chesnay Cédex, France
http://www-sor.inria.fr/

December 14, 1998

1 Why Distributed Garbage Collection

Dynamically-allocated memory must eventually be reclaimed. But manual reclamation is error-prone, and if an object is de-allocated prematurely, a program that later follows a pointer to it might behave incorrectly. Instead, *garbage collection* (GC) automatically reclaims objects that can no longer be reached by any path of pointers.

In a distributed system, an object might not only be referenced from one program, but also from other programs and other computers. This makes manual reclamation quite intractable. We consider here the problem of *distributed garbage collection* (DGC). DGC is particularly important in a large-scale distributed system, since a de-allocation error might cause a completely unrelated program to fail, possibly far away and an arbitrarily long time later.

DGC has been a subject of academic research since at least 1977 [3]. With the growth of the Internet, DGC is now receiving its share of commercial attention. For instance, both Java RMI [27] and DCOM [22] come with some form of DGC.

The next section contains a quick review of DGC techniques; for a more in-depth treatment, we refer to an earlier survey [20]. Then we detail two recent advances in DGC developed in the context of the Broadcast project. The first

*Ph.D student at École Polytechnique. Also with INRIA projet PARA.

†INESC, Rua Alves Redol 9-6, 1000 Lisboa Cedex, Portugal.

is an algorithm for collecting distributed cycles of garbage in a message-passing system; the second is an algorithm for DGC in a system with caching and/or replication. Both have been implemented in real systems and are in actual use.

This article assumes some familiarity with centralised GC, now a mature area [10, 26]. We use the following vocabulary. Objects connected by references form a graph; a remote reference may point into another process. An application or mutator enters the graph via roots, allocates objects, and assigns references. The system or collector reclaims garbage, i.e., objects that are reachable by no path from any root.

2 Distributed Garbage Collection approaches

We are interested in scalable DGC algorithms, which implies local execution, absence of remote synchronisation, low complexity and costs, and fault tolerance. Scalability is often achieved at the cost of incompleteness, i.e., such algorithms reclaim only a safe subset of actual garbage. Scalability excludes the apparently straightforward approach of running one of the existing centralised algorithms in a consistent view of the object graph, e.g., on a snapshot [4] or in a transaction. Our approach instead is to use the unique properties of DGC that enable specific, efficient solutions.

2.1 Two models of distributed systems

Before going into more detail, it is appropriate to explain our two models of distributed system. Classically a distributed system is defined as disjoint processes, sharing no memory, and communicating by *message passing*; this is our first model. The system is asynchronous, in that we do not assume any upper bound on message delivery time. An object exists at a single process at any time.

The alternative *shared-memory model* lets processes interact by sharing memory. The system simulates a shared memory by sending (in a message) the value of a chunk of memory to a remote site, where it is cached and mapped into application space. We call the different copies of a given chunk of memory its replicas. The system keeps replicas consistent, detecting updates and sending messages invalidating or updating out-of-date replicas. Messages are not visible to applications. This model includes Distributed Shared Memories (DSMs) such as Ivy [17], distributed shared object systems such as Orca [2], and cached distributed file systems or databases.

We use the word *bunch* for the smallest subdivision of the shared memory that can be replicated, often a page or segment. An object resides in a single bunch, but at any point in time the bunch (and the objects it contains) might be replicated to multiple processes.

2.2 Basic DGC techniques

The simplest DGC algorithm uses time-outs, as was first proposed in the Bullet file system [25]. Time-outs are simple and effective but unsafe, since a transient error can cause a reachable object to disappear.

Other DGC algorithms derive from the two well-known families of GC algorithms, namely *counting* and *tracing*. A counting algorithm maintains a count of references to every object. Counting is a local operation, hence the algorithm scales well. It has the disadvantage that it does not collect cycles of garbage. *Reference listing* maintains the list (and not just the number) of pointers to an object; this allows a reference originating from a site declared crashed to be reclaimed.

A tracing algorithm determines garbage by direct examination of the graph. A tracing collector collects all garbage, including cycles. However distributed tracing is not scalable, because it is a global algorithm, and because it runs in phases between which distributed synchronisation is necessary.

Some authors advocate *back-tracing* [19]: if a backwards graph walk returns to the start object and does not encounter a root, then that object is part of a garbage cycle. The backwards walk is expensive; therefore we do not considere back-tracing any further.

2.3 Hybrid collection algorithms

Practical DGC algorithms are a hybrid of tracing, counting, and time-out. For instance a hybrid DGC is used in the commercial remote-object system Java RMI [27]. Two systems that we developed, SSP Chains [23] and PerDiS [8], are also based on hybrid DGC; they will be examined in more detail in later sections.

A distributed system is typically partitioned for locality. In the DGC context, we call each part a *space*. An object exists and is located within a single space. DGC algorithms typically focus on collecting references that cross the boundaries between spaces. Such remote references are managed with reference lists, supplemented with time-outs to guard against crashes.

Each space independently takes care of its local garbage with a tracing algorithm.[1] An incoming remote reference is (conservatively) considered part of the local root set. When a local collector determines that an outgoing remote reference has become unreachable, the DGC sends a message to the target space, causing it to decrement its reference count. Time-outs are used to detect that a space has crashed; reference listing allows to selectively ignore references from a crashed space.

A hybrid DGC combines the best of all worlds. Within a space, the local collector will reclaim all garbage, including cycles, without imposing a global algorithm. Across spaces, counting or listing provides a scalable solution.

[1] Local tracing is not an obligation; local counting or even manual collection is possible too.

However, a hybrid algorithm does not collect cycles of garbage that span spaces. Furthermore, time-outs must be used with utmost care.

2.3.1 Hybrid collection in the message-passing model

In the message-passing model, each process (or processor) is a space. It is easy to detect reference mutations, because assigning a remote pointer requires a message, containing the reference, between processes. Then, sending or receiving the message causes the incrementation of the corresponding count.

This algorithm model is well understood; remaining open questions have to do with fault tolerance and with collecting distributed cycles of garbage. Section 3 presents a new algorithm for collecting distributed cycles in a large-scale distributed system.

2.3.2 Hybrid collection in the shared-memory model

In the shared-memory model, each bunch replica constitutes a separate space. At some arbitrary point in time, replicas of a same bunch may be mutually inconsistent. Multiple spaces may coexist in a single process. A pointer to an object designates any of its replicas, independently of process. Pointer assignments occur directly in memory. The collector must somehow be notified when the application assigns a pointer across a bunch boundary. This all makes it hard to determine which objects are reachable. The advance described in Section 4 is the Larchant algorithm, an efficient, scalable DGC algorithm for a distributed shared memory, which takes these issues into account.

3 Collecting distributed garbage cycles in the message-passing model

In this section, we present our recent work on collecting distributed cycles of garbage in message-passing systems. This work is mainly inspired from Hughes' algorithm [9], but our detector [14] extends Hughes to asynchronous distributed systems.

After introducing the main properties of our detector, we will present the algorithm and argue for its scalability and fault-tolerance. More implementation details can be found in le Fessant et al. [14].

3.1 Introduction

3.1.1 Terminology

In this paper, we use almost the terminology of the Stub-Scion Pair Chains system (SSPC).

Stubs and scions Each remote reference R from object A in space X to object B in space Y is represented by a local pointer in X from object A to a special object $stub_X(R)$, called a *stub* and used as a proxy in X for object B, and a local pointer in Y from another special object $scion_Y(R)$, called a *scion* and used as a local root in Y, to the object B. Stubs and scions come in pairs: each stub has exactly one *matching* scion, and each scion has at most one matching stub. A remote reference R is created by first creating the scion, then sending the reference (in fact, the scion identifier, called the *locator*) in a message, and finally by creating the associated stub in the remote space.

Timestamps and dates In the SSPC acyclic garbage collector, all messages are stamped to prevent race conditions. In our algorithm, we use both the original SSPC timestamps, and other timestamps (starting times of global traces, generated by a distributed Lamport clock) on stubs and scions to distinguish multiple concurrent traces. To avoid confusion, the former are called *timestamps*, while the latter ones are called *dates*.

3.1.2 Overview

Like Hughes' algorithm [9], our algorithm is based on multiple global traces progressing concurrently in the system. Each global trace starts at one space with a local garbage collection, is characterised by the current date from the Lamport clock at its starting space, and propagates that date from local roots to stubs reachable from those roots. Dates on stubs are then propagated along chains of remote pointers, from stubs to scions by messages and from scions to reachable stubs by local garbage collections. A stub is always marked with the highest date of all scions or local roots it is reachable from.

As a consequence, reachable stubs are marked with increasing dates, remotely-propagated from their original roots. On the contrary, unreachable stubs eventually stabilise at the starting date of the latest global trace when they were found to be reachable.

A global trace is *terminated* when its starting date does no longer appear on reachable stubs and scions. From the analysis of increasing dates on its stubs, each space computes the date, called localmin, of the oldest global trace which has not yet terminated for that space. All localmins are then gathered on a central space, called the *detection server*, which computes the oldest date, called globalmin among traces which have not terminated in one or more spaces. A scion marked with a date earlier than globalmin is known to be unreachable, since it is marked with the date of a terminated trace.

3.1.3 Main properties

This algorithm presents some interesting properties for distributed systems:

- *It is centralised*: centralisation is often seen as a drawback in distributed systems. However, for this particular problem, it enables a simple, lightweight and fault-tolerant solution. We will show that, in our case, the load of the central server is light and not a bottleneck.

- *It is clustering aware*: Networks can be organised as real clusters around a LAN loosely coupled to the rest of the world through a WAN connection. In our model, there is one server per cluster, and only intra-cluster cycles are collected. Inter-cluster cycles are not detected.

- *It is asynchronous:* all messages are sent asynchronously, and computations, either from mutators or from garbage collectors, are never blocked.

- *It is cheap in resources:* our algorithm relies on small local data structures, it sends few messages and only to spaces in the immediate vicinity, and its computation overhead is negligeable.

- *It has a low implementation cost:* our algorithm has been implemented for the Objective-CAML [16] platform of Stub-Scion Pair Chains [12]. Only minor modifications of the runtime were necessary to propagate timestamps as a side-effect of local garbage collection. Other parts of the algorithm were implemented over the runtime as an optional library.

- *It is fault-tolerant:* the algorithm is safe in the presence of unreliable communications (messages can be lost, re-ordered or duplicated) and crashes of participating spaces.

3.1.4 Comparison with Hughes' algorithm

Hughes' requirements are much stronger than ours: his algorithm relies on a global clock, on instantaneous and reliable communications, and doesn't support space crashes. Thus, important aspects of large-scale distributed systems, such as messages in transit or failures, are not addressed. All these strong requirements allow termination detection using an algorithm from Rana [21], based on a snapshot of local states.

Our assumptions are more in line with real distributed systems. Communication is asynchronous and unreliable. Hughes' global clock is replaced by a Lamport logical clock [11]. Spaces can crash without endangering the safety and liveness of our algorithm. Moreover, most computation is delayed until local garbage collection time, leading to a more conservative but cheaper algorithm. We replace Rana's termination algorithm by a minimum computation on a central server.

3.2 The algorithm: propagation of dates

3.2.1 Local garbage collection

At the beginning of each local garbage collection, local roots are marked with the current date of the local Lamport clock. Local roots and scions[2] are then sorted, and traced in decreasing order of their dates. During the trace of a root, each reachable stub is marked with the date of the traced root. Since objects are only marked once during a complete local garbage collection, each reachable stub is finally marked with the greatest date on roots (local roots and scions) from which it is reachable.

3.2.2 Remote propagation

After each local garbage collection, dates are propagated from marked stubs to their associated scions by Stubdates messages. Only messages carrying a increased date are taken into account; decreases are ignored. Moreover, the SSPC timestamps are used to test if the scion locator has been sent but not yet received at the local garbage collection that generated the Stubdates message. In such a case the scion date, set to *now* when its locator was sent, is not updated. Otherwise, the scion date is set to the current local date. Stubdates messages are only sent to participating spaces in the *immediate vicinity*, i.e., remote spaces holding scions associated with local stubs.

3.2.3 Characterisations of cycles of garbage

At each local garbage collection, local roots propagate a new larger date. Thus, the date of any reachable stub will eventually increase. In contrast, dates belonging to unreachable cycles evolve in two phases: the greatest date is first propagated to all stubs and scions in the cycle. Then, since there is no local root leading to the cycle, no new date can enter the cycle, thus the dates remain constant forever.

In the following, we interpret the behavior of stub dates in terms of concurrent traces: dates are markers of concurrent traces. Thus, reachable stubs are always marked by new traces, i.e., with new markers, whereas unreachable stubs remain marked with the marker – the starting date – of the last trace when they were found to be reachable.

This is illustrated by Figure 1. After local garbage collections on the three participating spaces, dates have been propagated along two chains of remote pointers. One is reachable, whereas the other forms a cycle of garbage. Observe that:

- Reachable stubs are marked with increasing dates (from 5 to 7).

[2] Scions are marked with the date received from their associated stub. Initially, they are marked with a special date, called *now*, which is always replaced by the current date in computations.

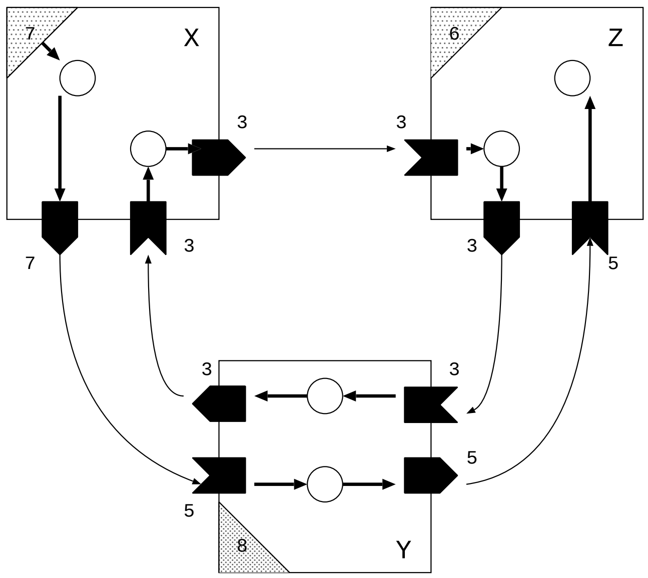

Figure 1: As a consequence of date propagation, unreachable stubs in a cycle are eventually marked with the date of a terminated trace, whereas dates on reachable stubs continue to increase with successive global traces.

- Stubs on unreachable cycles are eventually marked with the same, constant, date which is a marker for a terminated trace (trace 3 is terminated).

The goal of our algorithm is therefore to compute an increasing threshold, called globalmin, which is a minimum on dates found on stubs marked by successive global traces. Globalmin is a conservative approximation of the date of the oldest concurrent trace which has not terminated yet in one or more spaces. A trace is said to be *terminated* when its marker (its starting date) can not appear anymore on any reachable stubs or scions in any participating space, i.e., when its marker cannot be replaced by any greater marker during subsequent garbage collections.

3.3 The algorithm: computation of globalmin

3.3.1 Computation of globalmin by the detection server

The computation of globalmin requires a consensus involving all participating spaces. A fully distributed and fault-tolerant consensus would be extremely complex. However, for our particular case, the use of a centralised computation will be shown to be scalable and fault-tolerant.

In our system, globalmin is computated by a dedicated process, called the *detection server*. The value of globalmin is a conservative approximation of the date of the oldest non-terminated trace. A participating space computes a value, called localmin, representing the date of its oldest non-terminated trace. It periodically sends localmin to the detection server in a Localmin message. Since a space can believe that a trace is locally terminated, and then, receive the marker of that date from another space, the value of localmin varies non-monotonically.

However, globalmin, the conservative minimum of all localmins, is monotonically increasing, since a trace is terminated only when terminated for all spaces. Unreachable stubs and scions are marked with the dates of terminated traces, which eventually become smaller than globalmin.

The value of globalmin on the server does not need to be always up-to-date. Keeping an old value of globalmin means that some terminated traces have not been detected as terminated by the detection server, and therefore that some garbage cycles with those dates will not be detected immediately. However, when globalmin is updated, those cycles becomes detected as soon as the value of globalmin will be greater than their dates. Consequently the server can compute globalmin with any arbitrary period, trading off load on the detection server against promptness of cycle collection.

The only information maintained by the detection server is the list of participating spaces and, for each participating space, the last value of localmin received. This information is simple to recover in the case of a crash of the detection server. A new detection server can be started promptly in the same computation state, without aborting any global trace.

3.3.2 Computation of localmin by a participating space

For any participating space, localmin is the oldest date of the traces that have not terminated yet for that space. In the following, we note that a space *protects* a date when the trace associated with that date has not terminated in that space, and therefore the localmin for that space is smaller than that date.

By definition, a trace is terminated when its marker can not appear on reachable stubs or scions, i.e., when no new trace can mark stubs or scions marked with the date of the terminated trace. Thus, two criteria must be satisfied for a particular trace to be globally terminated:

- *Local criterion:* there is no stub that is marked with the date of the trace and locally reachable from a scion or root marked with a greater date.

- *Remote criterion:* there is no scion that is marked with the date of the trace and associated with a stub marked with a greater date.

If a trace does not satisfy either of these two criteria, its date must be protected by at least one of the participating spaces, since it is not terminated yet.

Localmin values are only computed after each local garbage collection. Therefore, a participating space must compute a conservative approximation of these two criteria, taking the asynchronous nature of messages and the delays in the computation of localmin in other spaces into account. We refine the two criteria into the following two rules:

- *Local rule:* When the local collector increases the date of a stub, the stub's previous date must be protected by the next value of localmin. Indeed, the downstream scion is still marked with the stub's previous date, and from the remote criterion, the trace started at the stub's previous date is not terminated yet.

- *Remote rule:* when a Stubdates message increases the date of a scion, the scion's previous date must be protected until the stub's new date has been propagated from the scion by a local garbage collection. Indeed, when the scion receives a new date, there might be stubs reachable from that scion and still marked with the scion's previous date and, from the local criterion, the trace started at the scion's previous date is not terminated yet.

Observe that the local rule is derived from the remote criterion, and the remote rule from the local criterion. This is a consequence of asynchrony: a trace can be terminated at one space and appear to be terminated on another space, while in fact a message containing its date is in transit between these spaces. Thus, the sender's locamin must protect dates whose traces are locally terminated but not remotely terminated.

We can now combine both rules in a single one:

- *Protection rule:* when the local garbage collector of some space increases the date of a stub, the stub's previous date must be protected by the next values of localmin (local rule). Moreover, since the new date will be propagated by a message to the stub's associated scion, also marked with the stub's previous date, the stub's previous date must be protected by next values of localmin until the new date has been propagated by a local garbage collection from the associated scion (remote rule).

In what precedes, we did not take into account the fact that globalmin is computed on the detection server, and that dates protected by one localmin value are really protected on all spaces only when the Localmin message has been received by the detection server. Consequently, the protection rule must be refined:

- *Global protection rule:* when the local garbage collector of some space increases the date of a stub, that stub's previous date must remain protected by localmin until the collector of the downstream space has itself

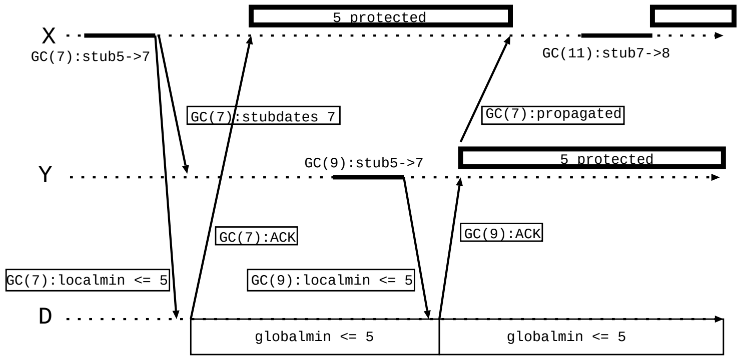

Figure 2: The global protection rule is safe: space X stops protecting date 5 when Y has started protecting it.

propagated the new date from the stub associated scion, and the resulting localmin has been received by the detection server (see Figure 2).

The global protection rule is safe: if a trace is not terminated, a new date will be propagated to reachable stubs marked with the trace date, and the trace date will be protected. It is also live: the date of a terminated trace is not protected, as soon as stubs marked with that trace are not reachable for newer traces.

To implement the global protection, a local garbage collector computes, for each remote space, the minimum date which must be protected, called ProtectNow. ProtectNow is entered into the ProtectedSet of all dates protected on the space. Localmin is then computed as the minimum of all entries in the ProtectedSet. Localmin is sent to the detection server, which returns an Ack message. Once the Ack is received, this space knows that all dates received before the local garbage collection (from Stubdates messages) have been propagated and that the new localmin has been received by the server. This information is communicated to other spaces by a Propagated message, containing the dates of remote garbage collections associated with the Stubdates messages, which are stored in a special set, called the PropagatedSet.

3.3.3 Coping with mutator activity

The mutator can concurrently with the distributed garbage collector send remote references (either by creating a new stub-scion pair or by using existing stub-scion pairs) or invoke methods of objects on remote spaces.

When a new stub-scion pair is created, the scion date is initialized to *now* and the stub date to the current date of its space. When re-using an existing stub-scion pair, the scion owner does not know if the stub associated with its scion is still live. Thus, it must behave as if a new pair were created: the scion date is set to *now*. The *now* date on a scion is only replaced when a new date is propagated from its matching stub. However, to avoid race conditions, the Stubdates message propagating the new date must have been sent after the last message containing the scion locator has been received by the sender space. Observe that this same mechanism allows a space to communicate with a space not participating in cyclic collection, since scions reachable from them remain marked with *now* and thus, are never collected.

When the mutator involves an object through a stub-scion pair, this is sufficient proof that the stub is reachable. Therefore, an invocation increases the stub's date to the current date. As a consequence of the global protection rule, its previous date must therefore be protected by subsequent localmin values.

3.4 Complexity

We can now examine the complexity of our algorithm, in memory, computation time and communication requirements.

3.4.1 Participating spaces

Memory consumption: Compared to the SSPC algorithm, each stub contains two extra date fields and each scion one extra date field. Furthermore, our algorithm needs two data structures at each participating space: a ProtectedSet, containing the dates to be protected for each remote space by the value of localmin, and a PropagatedSet, containing the timestamps of Stubdates messages received before a local garbage collection for which Localmin message is awaiting acknowledgement. Both can be implemented as FIFO queues, one for each remote participating space, since their entries are removed in FIFO order.

For the ProtectedSet, one entry is put in each queue for each local garbage collection. Therefore size of a queue depends on the frequency of local garbage collections between remote collections. If the local garbage collection frequencies are similar, each queue should contain a small number of entries. However, if one space requires too many garbage collections, and its set becomes too large, date propagation and computation of localmin should be avoided until sufficient entries have been removed from its ProtectedSet.[3]

The size of the PropagatedSet depends on the delay between transmitting a Localmin message and receiving the Ack. Thus, it should almost always be very small unless communication is very unreliable.

[3]Notice than one remote garbage collection can remove several entries at once, i.e., all the entries for local garbage collections whose Stubdates messages were received before the date of the remote garbage collection.

Computation time: The main computation on a participating space is done at local garbage collection time.

The local trace requires that scions are sorted by decreasing order of their dates. This sort can be done from scratch (cost $O(N.log(N))$ in the number of scions) at each local garbage collection, or by inserting a scion into a sorted table each time its scion date is modified. Assuming there are fewer dates than scions, another approach could be to put all scions having the same scion date into the same set, and then sort those sets.

Localmin is computed as the minimum of all increasing stub old dates, and of all the entries in the ProtectedSet. Thus, there is one comparison for each stub, to compute the current date to be protected for the current local garbage collection for each remote space, and one comparison for each entry in the ProtectedSet, i.e., $O(n_{stubs}) + O(n_{spaces})$.

3.4.2 Detection server

The detection server, although centralised, is not a bottleneck. Indeed, its computation and memory requirements are low, and linear in the number of participating spaces:

- *Memory consumption:*

  The detection server maintains a vector identifying participating spaces and, for each one, its last localmin received.

- *Computation time*:

  Globalmin is computed periodically as the minimum of the localmins of the participating spaces. To trigger this computation, a good stimulus is that all participating spaces whose previous localmins were equal to globalmin have increased their localmin. The number of such spaces can easily be computed during the computation of globalmin.

3.4.3 Messages

For each local garbage collection, two messages are sent to each remote space in the vicinity: Stubdates and Propagated. There are also two messages exchanged with the detection server: Localmin and Ack. While Propagated, Localmin and Ack are small messages (containing at most three values), the size of the Stubdates message depends on the number of live stubs, since it contains one date for each stub.

3.5 Fault-Tolerance

This algorithm is tolerant to both unreliable communication and space failures. Although building reliable communication above unreliable communication is often not a problem, space failures cannot be avoided in a real distributed system.

Unreliable communication is tolerated because of the conservative approach taken in the communication of globalmin and localmin. Localmins are computed not only from the protected dates of the current garbage collection, but also for previous collections (ProtectedSet). Globalmin is computed from the last localmins received, and its value is always safe, even if not up-to-date.

The detection server can crash, and some new process can become the new detection server without aborting the current detection process nor the current global traces.

Participating spaces can also crash: in such a case, the detection server propagates the new list of participants to all remaining participating spaces. Each space sets its scion dates from suspected spaces to the special value *now*. Then, the detection server waits for the new localmin value computed after this update, before computing the next value of globalmin.

3.6 Scalability

Our algorithm has very few requirements on participating spaces: low computation and memory overhead, and few messages sent for each local garbage collection. However, it has two main drawbacks: the computation is centralised and the detection can be slowed down by any slow participating space.

However, we consider this algorithm interesting in large scale networks. Indeed, the limitation introduced by these two drawbacks is on the *number* of participating spaces, and not the *distance* between them. Since cycle detection is most useful between long-lived (and/or persistent) servers, such limitations in are not important. Short-lived clients are simply considered as non-participating spaces, and their cycles will be collected at their death, whereas long-lived clients are inserted into the set of participating spaces with the long-lived servers.

3.7 Discussion of cyclic distributed garbage collection

We have described a detector of distributed cycles of garbage. Our cycle detector presents some interesting properties for large-scale systems: asynchrony between participating spaces, handling of non-participating spaces, tolerance to communication faults and crashes, low resource requirements, and ease of implementation (no modifications to local objects, only a few to the local garbage collector). This algorithm has been implemented in a distributed system; details are available elsewhere [14].

We are now working on a new version of this cycle detector using both propagation of marks and back-tracing. This new algorithm will have the same properties as the one described here, plus the ability to detect cycles spanning any spaces in the whole system. This new algorithm is currently being implemented for our mobile agents platform [13].

4 The Larchant DGC algorithm for distributed shared stores

4.1 Introduction

Modern distributed systems rely on caching and replication to speed up access to remote objects. A reference can then be resolved directly in the local memory. The purest example is provided by a Distributed Shared Memory (DSM) [17], which enables processes on different computers to share data simply by using pointers, just like in a centralised program. Distributed file systems and object-oriented databases provide similar facilities. In what follows we will refer to all such cached or replicated memories as DSMs.

DGC in a distributed shared memory is a harder problem than DGC in the message-passing model studied in Section 3 because:

- Applications modify the graph concurrently, not by sending messages but simply by pointer assignment. This is a very frequent operation, which should not be slowed down (for instance by inserting reference counting code in the pointer assignment path).
- Replicas are not instantly coherent. Observing a consistent image of the graph is difficult and costly.
- The pointer graph may be very large and distributed. Much of it resides on disk. Tracing the whole graph in one go is unfeasible.
- A localised change to the pointer graph can affect remote portions of the graph. This has consequences on the global ordering of operations.
- GC should not compete with applications. For instance, it should not take locks, cause coherence operations, or cause I/O.

We consider previous results on collecting a DSM [1, 15, 18, 28] inapplicable because it does not take the above issues into account. We describe now work done in the context of Larchant, a distributed and persistent shared store, intended for interactive cooperative tasks. Larchant consists essentially of a large-scale DSM; only objects reachable from a *persistent root* (e.g., a name server) persist.

The main goals of our distributed GC algorithm are correctness, scalability, low overhead, and independence from a particular coherence algorithm. Secondary goals are avoiding source code and compiler changes.

Our approach divides the global GC into small, local, independent pieces, that run asynchronously, hence can be deferred and run in the background:

- The store is partitioned into bunches; a bunch may be replicated. GC is a hybrid of tracing within a bunch and counting across bunches.

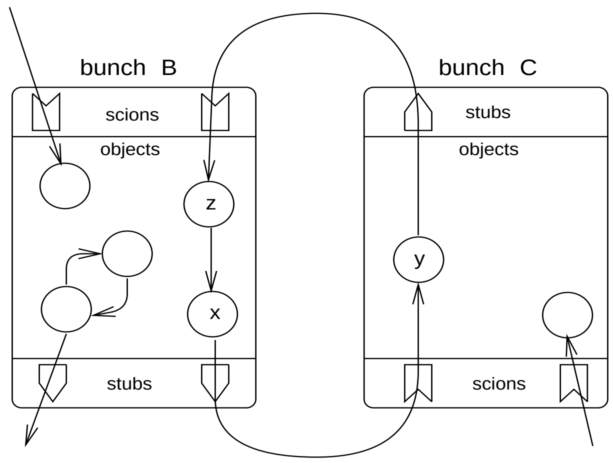

Figure 3: *Two bunches containing objects, stubs, and scions.*

- Each site runs a collector with a standard tracing algorithm [26] that works in one or more bunch replicas (on that site) at the same time.
- The cooperation protocol between collectors does not entail mutual synchronisation.
- A collector examines only the local portion of the graph, without causing any I/O or taking locks.
- A collector may run even when local replicas are not known to be coherent.

The Larchant DGC algorithm satisfies five simple correctness rules. The algorithm is safe (no reachable data is reclaimed) and live (garbage is eventually reclaimed). Sadly but unavoidably, it is not complete.

The underlying theoretical model and a proof of correctness can be found in Ferreira and Shapiro [7]. More information about the Larchant algorithm can be found in previous publications [5, 6, 24].

4.2 GC algorithm and safety rules

The Larchant memory, as shown in Figure 3, is subdivided into coarse-grain *bunches* (typically, a set of continguous pages) in which objects are allocated. A bunch can be replicated (e.g., cached) in multiple processes. An object can point to another object, either in the same bunch or across bunches. In both cases the reference is a raw pointer, but in the second case the collector maintains additional data structures, a *stub* for an outgoing pointer and a *scion* for an

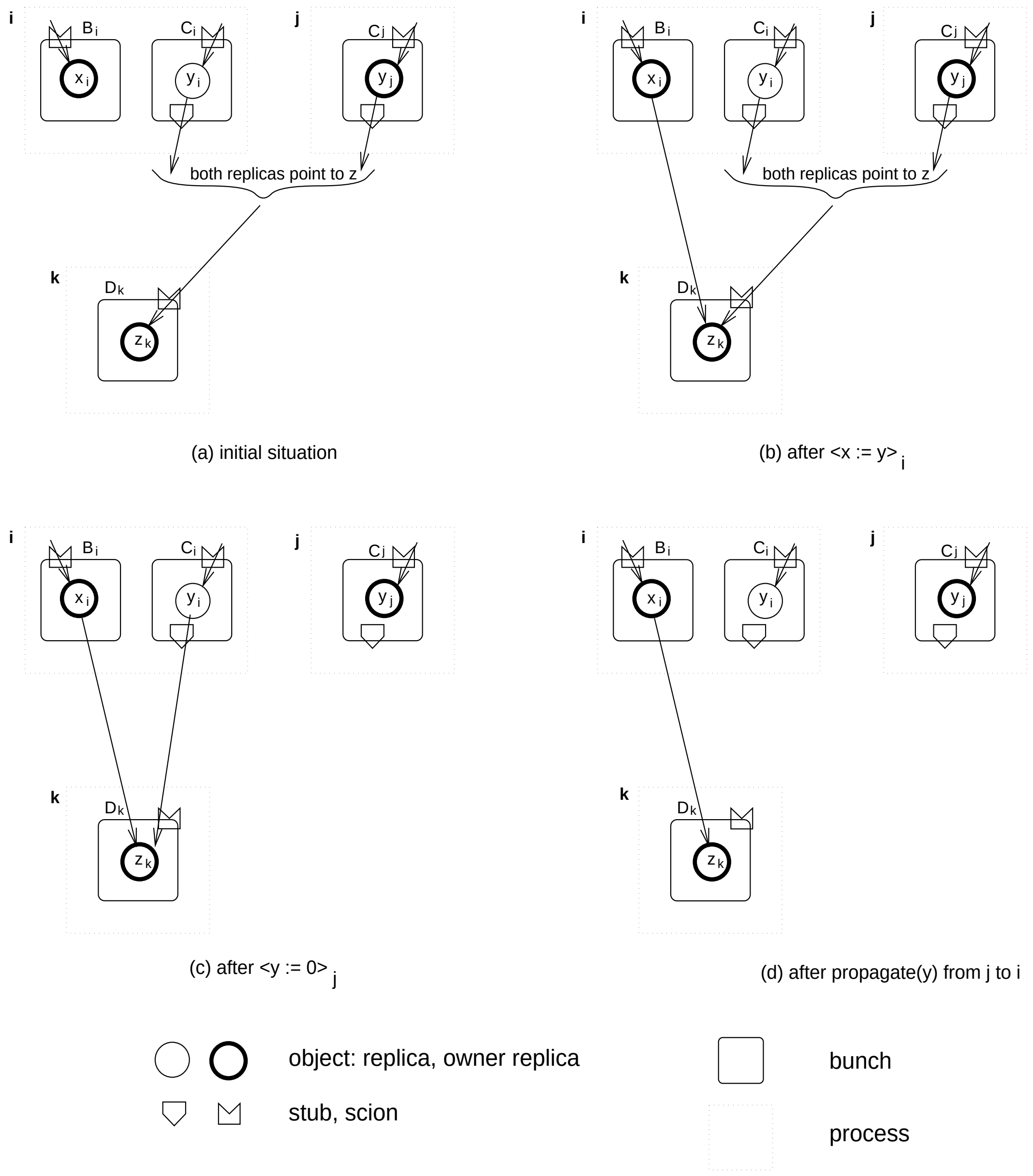

Figure 4: *Prototypical example of mutator execution. Note that the stubs and scions become temporarily inconsistent with the pointers. However, as described in the paper, this does not compromise safety.*

incoming one.[4] A typical execution example is shown in Figure 4.

Larchant uses a hybrid DGC, as defined in Section 2.3. Each replica of each bunch forms a space.[5] A reference within a bunch is subject to tracing collection, whereas references that cross bunch boundaries are counted. To avoid interfering with the application, we assume that the collector discovers pointer assignments after the fact, as it traces a bunch. Tracing a bunch causes stubs to be created or deleted; the counting algorithm adjusts scions accordingly.

We now outline the basic algorithm and define some notation. When a mutator performs the assignment noted $<x := y>_i$ (i.e., copy the value of pointer y into pointer x in process i), up to three processes are involved in the corresponding counting. Say objects x, y, z and t are located in bunches X, Y, Z and T respectively. Suppose that prior to the assignment, x pointed to z and y pointed to t. As a consequence of the above assignment operation, the collector of process i increments the reference count for t by performing the local operation $<\text{increment.stub}(Xx, Tt)>_i$ and sending message $\text{increment.scion}(Xx, Tt)$ to the collector in some process j, managing T. In addition, it decrements the reference count for z by performing the local operation $\text{decrement.stub}(Xx, Zz)$, and sending message $\text{decrement.scion}(Xx, Zz)$ to the collector in some process k, managing Z. These adjustments ot reference counts need not occur immediately, as we will see later.

4.2.1 Tracing in the presence of replicas: the Union Rule

Each process runs a standard centralised tracing collector. The issue we raise now is how collectors cooperate, in order to take replication into account. It is desirable that a collector remain independent, both of remote collectors, and of the coherence algorithm. In our design, a collector may scan a local replica, even if it is not known to be coherent, and independently of the actions of remote collectors.

Thus, the collector at process i might observe x_i to be pointing to z, whereas collector at process j concurrently observes x_j to be pointing to t. The coherence protocol will eventually make both replicas equal, but the collector cannot tell which value of x is correct. In the absence of better information, the collector must accept all replicas as equally valid, and never reclaim an object until it is observed unreachable in the union of all replicas. This is captured by the following rule.

[4]Despite the similar vocabulary and role, these stubs and scions are somewhat different from those in SSPC used in Section 3. Larchant's stubs and scions are auxiliary data structures of the distributed collector; the mutator does not see them in any way; increments and decrements occur in the background.

[5]To the first approximation only, because Larchant changes space boundaries dynamically by grouping together a number of bunches [24]. However this is beyond the scope of the current discussion.

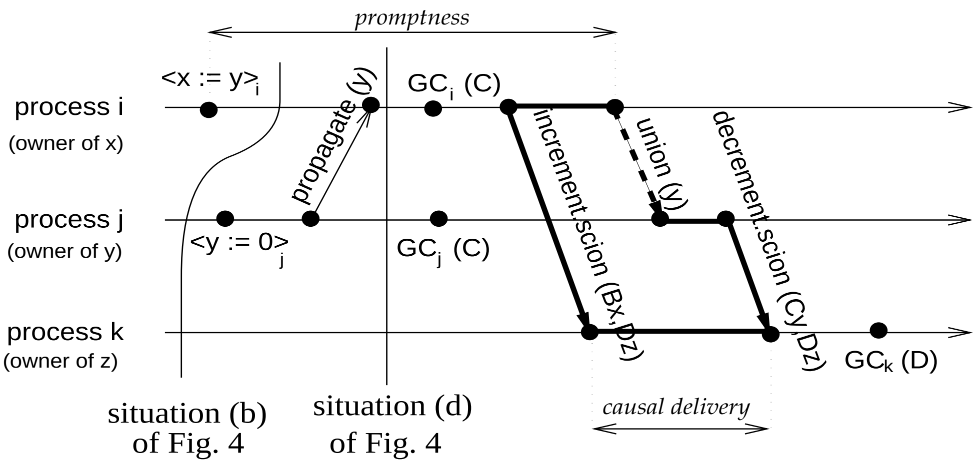

Figure 5: *Timeline showing the effect of some of the safety rules for the example of Figure 4. On site* i*, the sending of* increment.scion(Bx, Dz) *may be delayed at most until sending* union(y)*. Note the causal dependence (indicated by the thick lines) between the* increment.scion *and* decrement.scion *messages, carried by the* union *message.*

> **Safety Condition I: Union Rule.** *If some replica* x_i *points to* z*, and some replica* x_j *is reachable, then* z *is reachable.*

The above rule makes reachable some object z referenced only by an unreachable replica x_i, if some other replica x_j of the same object is reachable. This very conservative formulation is necessary in the absence of knowledge of the coherence algorithm.

An efficient implementation of the Union Rule applies to single-owner coherence protocols such as entry consistency. In such protocols, a single "owner" process centralizes the information about a given object.[6] The collectors centralise the information about pointers from x at the owner of x, using what we call union messages. In other words, a process holding replica x_j sends a union message, to x's owner, after detecting a change in the pointers from x_j. (Note that this detection is achieved by tracing x_j's enclosing bunch.)

Now, suppose that x points to z, and x is assigned a new value (for instance the null pointer). It is only when all the replicas of x have the new value, and the corresponding collectors have informed x's owner (by sending it a union message) that there are no pointers from x to z, that the owner of x sends a message to the owner of z, to decrement the corresponding scion's reference count. This technique moves some of the responsibility for reference counting to the owner of the objects where references originate.

[6] The owner of an object may change over time.

4.2.2 Cross-bunch counting and more safety rules

The standard approach to reference counting is to instrument assignments in order to immediately increment/decrement the corresponding counts. This approach requires compiler modification, and is expensive when assignments are frequent and counting is a remote operation, as is the case in Larchant.

Our solution consists of deferring the counting to a later tracing. In fact, the counts need not be adjusted immediately. Consider an assignment $<\mathsf{x} := \mathsf{y}>_\mathsf{i}$, where y points to z. At the time of the assignment, z is reachable by definition, and is guaranteed to remain reachable as long as y_i is not modified and remains reachable. It is not necessary for (a process managing) x to increment z's reference count, as long as (some process managing) y does not decrement it.

Let us return to the example of Figure 4. At the time of $<\mathsf{x} := \mathsf{y}>_\mathsf{i}$, object z is reachable (from both replicas of y) and is protected by some scion, say scion(Tt, Zz); presumably, but not necessarily, T = Y and t = y. As long as z's scion has a non-zero count, it is safe to delay the increment (possibly, the creation) of scion(Bx, Dz). (Recall that, in our system, it is the trace of bunch X which updates X's set of stubs, which in turn causes the corresponding scion count to be adjusted.)

However, there a problem remains with this approach. In the example, once situation (d) has been reached, it is possible that decrement.scion(Cy, Dz) reaches site k before increment.scion(Bx, Dz); then z could be incorrectly reclaimed. To avoid this unsafe situation, it suffices to give precedence to increment.scion over decrement.scion and union messages. This is illustrated in Figure 5: the interval labeled *promptness*, shows how much the message increment.scion(Bx, Dz) can be delayed with respect to the moment when the corresponding assignment operation ($<\mathsf{x} := \mathsf{y}>_\mathsf{i}$) has been performed.

The following rules say how late counting can be deferred, while still receiving messages in a safe order.

Safety Condition II: Increment Before Decrement Rule. *Scanning an object causes the corresponding* increment.scion *messages to be sent immediately.*

Safety Condition III: Comprehensive Tracing Rule. *When process* i *sends a* union *or* decrement.scion *message, all replicas at* i *must have been scanned since most recently assigned.*

Safety Condition IV: Clean Propagation Rule. *When process* i *sends* propagate(x)*,* x_i *has been scanned since it was most recently assigned.*

Safety Condition V: Causal Delivery Rule. *Garbage-collection messages (*increment.scion*,* union *and* decrement.scion*) are delivered in causal order.*

Rule II allows an object replica to be scanned at any time; scanning an object that contains a new pointer immediately sends an increment.scion message to the referent. It's important to mention that messages are asynchronous, so its actual transmission might take place later, provided messages are delivered in order, which is ensured by Rule V.[7]

Rule III ensures that union and decrement.scion messages are sent after increment.scion messages. In conjunction with Rule II, it ensures all increment.scion messages that the unions and decrement.scions might depend on have indeed been sent.

Rule IV ensures that when a process receives a new object via a propagate operation, any increment.scions corresponding to its new value have already been sent.

If delivery order is no better than FIFO, races can appear between increment.scion and decrement.scion messages. Rule V solves this problem. Note that coherence messages do not need causal delivery, thus limiting the cost.

Rules I through V are sufficient for the safe coexistence of replicated data and a hybrid garbage collector. They are independent of the coherence and tracing algorithms, and impose very few interactions between collection and coherence.

In the following sections we explain these rules in more detail, and provide some examples in which their need is clear and easily understandable.

Comprehensive tracing rule This section provides an explanation of the Comprehensive Tracing Rule. We show the need for this rule by giving an example of what happens when this rule is not enforced.

Consider Figures 4 and 5 after mutators have executed $<\mathsf{x} := \mathsf{y}>_\mathsf{i}$, $<\mathsf{y} := 0>_\mathsf{j}$, and y propagated to site i, i.e., once situation (d) has been reached. (Note that $\mathsf{scion}(\mathsf{Bx}, \mathsf{Dz})$ has not been created yet.) Suppose that $\mathsf{trace}_\mathsf{i}(\mathsf{C})$ runs and the Comprehensive Tracing Rule is not fulfilled. This means that the collector sends a union message to j (owner of y) indicating that $\mathsf{stub}(\mathsf{Cy}, \mathsf{Dz})$ has disappeared in process i, but $\mathsf{scan}_\mathsf{i}(\mathsf{x})$ is not performed; therefore $\mathsf{increment.scion}(\mathsf{Bx}, \mathsf{Dz})$ is not sent. As a result, j applies the Union Rule and $<\mathsf{send.decrement.scion}(\mathsf{Cy}, \mathsf{Dz})>_\mathsf{j}$ is executed. Thus, $\mathsf{scion}(\mathsf{Cy}, \mathsf{Dz})$ is deleted in k. Then, if $\mathsf{trace}_\mathsf{k}(\mathsf{D})$ runs, object z is unsafely reclaimed.

The Comprehensive Tracing Rule prevents the above scenario as it forces x_i to be scanned before i sends the union message to j. Then, according to the Increment Before Decrement Rule, $<\mathsf{send.increment.scion}(\mathsf{Bx}, \mathsf{Dz})>_\mathsf{i}$ is performed before the union message is sent (and j applies the Union Rule and executes $<\mathsf{send.decrement.scion}(\mathsf{Cy}, \mathsf{Dz})>_\mathsf{j}$). Since we assumed causal delivery (Rule V) $\mathsf{scion}(\mathsf{Bx}, \mathsf{Dz})$ is created before $\mathsf{scion}(\mathsf{Cy}, \mathsf{Dz})$ is deleted. Consequently, z is not reclaimed by $\mathsf{trace}_\mathsf{k}(\mathsf{D})$.

[7] In other words, sending a message only puts it on an ordered send queue.

Clean propagation rule This section provides an explanation of the role of the Clean Propagation Rule, by describing an example of what occurs if this rule is not enforced.

Consider Figures 4 and 5 after the mutator has executed $<\mathsf{x} := \mathsf{y}>_\mathsf{i}$ and before $<\mathsf{y} := 0>_\mathsf{j}$, i.e., once situation (b) has been reached. Now, suppose that the coherence algorithm propagates the new value of $\mathsf{x_i}$ to some process w in the absence of the Clean Propagation Rule, i.e., without first performing $\mathsf{scan_i(x)}$. Then, $\mathsf{x_i}$ is assigned in such a way that it no longer points to z (e.g., $<\mathsf{x} := 0>_\mathsf{i}$). At this moment, the only scion that protects z is $\mathsf{scion(Cy, Dz)}$. Suppose that both replicas of y are modified in processes i and j such that they no longer point to z. Hence, by the collection algorithm, $\mathsf{scion(Cy, Dz)}$ is deleted. Thus, z may be unsafely reclaimed by $\mathsf{trace_k(D)}$ ($\mathsf{x_w}$ still points to z).

The Clean Propagation Rule prevents the above scenario as it forces $\mathsf{x_i}$ to be scanned. Thus, by Rule II, $<\mathsf{send.increment.scion(Bx, Dz)}>_\mathsf{i}$ is performed immediately, i.e., before $\mathsf{x_i}$ is propagated to site w.

Causal delivery rule This section shows the need for the Causal Delivery Rule by giving an example of what happens when this rule is not enforced.

Consider the example of Figure 4 after mutators have executed $<\mathsf{x} := \mathsf{y}>_\mathsf{i}$, $<\mathsf{y} := 0>_\mathsf{j}$, and y propagated to site i, i.e., once situation (d) has been reached. (Note that $\mathsf{scion(Bx, Dz)}$ has not been created yet.) Then, the collectors on sites i and j perform as follows: i executes $<\mathsf{send.increment.scion(Bx, Dz)}>_\mathsf{i}$, while j executes $<\mathsf{send.decrement.scion(Cy, Dz)}>_\mathsf{j}$. In an asynchronous system, the former may be delivered after the latter, causing z to be incorrectly reclaimed. In fact, there is a hidden causality relation through the shared variable y. In our algorithm, this causal relation is captured by the union message, as apparent in Figure 5. Thus, given the Causal Delivery Rule, there is at all times at least a scion protecting z from being reclaimed.

4.3 Discussion: a DGC algorithm for cached/shared memory

What precedes focused on the interactions between garbage collection and replication (or caching), applied to a replicated (or cached) shared memory. We showed that both the tracing and the distributed counting garbage collector can execute independently of coherence. Garbage collection does not need coherent data, never causes coherence messages nor input/output, and it does not compete with applications' locks or working sets. However, coherence messages must at times be scanned before sending.

Our GC is a hybrid algorithm for a DSM. It combines tracing within a partition, with reference-counting across partition boundaries. Each process may trace its own replicas, independently of other replicas. Counting (adjusting stubs and scions) at some process happens concurrently to other processes, and in the background with respect to the local mutator. In addition, counting is deferred and batched.

We presented five safety rules that guarantee the correctness of the distributed reference-counting algorithm. These safety rules are minimal and generally applicable:

- Union Rule: an object may be reclaimed only if it is unreachable from the union of all replicas (of the pointing objects);
- Increment before Decrement Rule: when an object is scanned, the corresponding increment.scion messages must be sent immediately (i.e., put in the sending queue);
- Comprehensive Tracing Rule: when a union or a decrement.scion message is sent, all replicas (on the sending site) must be have been scanned since they were most recently assigned;
- Clean Propagation Rule: an object must be scanned before being propagated; and
- Causal Delivery Rule: GC messages must be delivered in causal order.

Measurements of our first (non-optimized) implementation [5] show that the cost of tracing is independent of the number of replicas, and that there is a clear performance benefit in delaying the counting.

Causal delivery, imposed by Rule V, is non-scalable in the general case; however, we do not consider this to be a serious problem in real implementations because causality can be ensured by taking advantage of the specific coherence protocols. For example, in our current implementation (supporting entry consistency) causal delivery is ensured by a mixture of piggy-backing and acknowledgments.

The Larchant algorithm is in use by the Esprit Project PerDiS [8], where it supports a large-scale cooperative engineering CAD application. This will enable us to measure and characterize the behaviour of real persistent applications, to fully study the performance of the distributed GC algorithm and to evaluate its completeness in a real-world environment. The PerDiS implementation is freely available at http://www.perdis.esprit.ec.org.

5 Conclusion

We presented two recent advances in Distributed Garbage Collection in large-scale distributed computing systems. The first one is an algorithm for collecting cycles of garbage in a partitioned message-passing system. It is applicable in a clustered system (e.g., a network of LANs) communicating by RPC, as in Corba, Java RMI or DCOM. This algorithm has been implemented and proved correct. It is used in a prototype extension to ML supporting mobile objects, the JoCAML system [13].

Our second advance is an algorithm to collect garbage in a replicated or cached memory. It is applicable to distributed shared memories, object-oriented databases, and persistent distributed stores. It has been proven safe and live, although by design for scalability, it is not complete. The algorithm provides the basis for the PerDiS platform developed in Esprit project 22.533 and used for cooperative engineering applications.

Acknowledgments

The initial research for both these algorithms was conducted within the Broadcast and Broadcast-WG projects. We are greatful to Xavier Blondel for his improvements of Larchant and his research and implementation of DGC for the PerDiS platform. Many thanks to Ian Piumarta for his constructive comments.

References

[1] Laurent Amsaleg, Olivier Gruber, and Michael Franklin. Efficient incremental garbage collection for workstation-server database systems. In *Proc. 21st Very Large Data Bases (VLDB) Int. Conf.*, Zürich (Switzerland), September 1995.

[2] Henri E. Bal, Raoul Bhoedjang, Rutgwe Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, feb 1998.

[3] P.B. Bishop. Computer systems with a very large address space and garbage collection. Technical Report MIT/LCS/TR-178, Mass. Insitute of Technology, Cambridge MA (USA), 1977.

[4] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[5] Paulo Ferreira. *Larchant: ramasse-miettes dans une mémoire partagée répartie avec persistance par atteignabilité*. Thèse de doctorat, Université Paris 6, Pierre et Marie Curie, Paris (France), May 1996. http://www-sor.inria.fr/publi/ferreira_thesis96.html.

[6] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey CA (USA), November 1994. ACM. http://www-sor.inria.fr/publi/GC-DSM-CONSIS_OSDI94.html.

[7] Paulo Ferreira and Marc Shapiro. Modelling a distributed cached store for garbage collection. In *12th Euro. Conf. on Object-Oriented Prog. (ECOOP)*, Brussels (Belgium), July 1998. http://www-sor.inria.fr/publi/MDCSGC_ecoop98.html.

[8] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, Joâo Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. PerDiS: design, implementation, and use of a PERsistent DIstributed Store. Technical Report QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, QMW, CSTB, INRIA and INESC, October 1998. http://www-sor.inria.fr/publi/PDIUPDS_rr3525.html.

[9] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985. Springer-Verlag.

[10] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester (GB), 1996. ISBN 0-471-94148-4.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] Fabrice Le Fessant. The camlsspc system. 1997.

[13] Fabrice Le Fessant. The jocaml system. Technical report, INRIA, 1998.

[14] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation for complete asynchronous distributed garbage collection. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[15] T. Le Sergent and B. Berthomieu. Incremental multi-threaded garbage collection on virtually shared memory architectures. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 179–199, Saint-Malo (France), September 1992. Springer-Verlag.

[16] Xavier Leroy. The objective-caml system software. Technical report, INRIA, 1996.

[17] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[18] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *Proc. Int. Workshop on Distributed Object Management*, pages 1–15, Edmonton (Canada), August 1992.

[19] U. Maheshwari and B. Liskov. Collecting distributed garbage cycles by back tracing. In *Principles of Distributed Computing*, Santa Barbara CA (USA), aug 1997. ACM.

[20] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), September 1995. http://www-sor.inria.fr/publi/SDGC_iwmm95.html.

[21] S. P. Rana. A distributed solution to the distributed termination problem. *Information Processing Letters*, 17:43–46, July 1983.

[22] Roger Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects.* Wiley, December 1998. ISBN 0-471-19381-X.

[23] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1992. http://www-sor.inria.fr/publi/SSPC_rr1799.html.

[24] Marc Shapiro and Paulo Ferreira. Larchant-RDOSS: a distributed shared persistent memory and its garbage collector. In J.-M. Hélary and M. Raynal, editors, *Workshop on Distributed Algorithms (WDAG)*, number 972 in Springer-Verlag LNCS, pages 198–214, Le Mont Saint-Michel (France), September 1995. http://www-sor.inria.fr/publi/LRDSPMGC_wdag95.html.

[25] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The design of a high-performance file server. In *Proceedings of the 9th Int. Conf. on Distributed Computing Systems*, pages 22–27, Newport Beach CA (USA), June 1989. IEEE.

[26] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag. ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps.

[27] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the java system. In *Conf. on Object-Oriented Technologies*, Toronto Ontario (Canada), 1996. Usenix.

[28] V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proc. Data Engineering Int. Conf.*, pages 120–133, Houston TX (USA), February 1994.