Modelling a Distributed Cached Store for Garbage Collection: the algorithm and its correctness proof*

Paulo Ferreira[1] and Marc Shapiro[2]

[1] INESC/IST, R. Alves Redol N° 9, Lisboa, Portugal
paulo.ferreira@inesc.pt
[2] INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France
marc.shapiro@inria.fr

Abstract. Caching and persistence support efficient, convenient and transparent distributed data sharing. The most natural model of persistence is persistence by reachability, managed automatically by a garbage collector (GC). We propose a very general model of such a system (based on distributed shared memory) and a scalable, asynchronous distributed GC algorithm. Within this model, we show sufficient and widely applicable correctness conditions for the interactions between applications, store, memory, coherence, and GC.
The GC runs as a set of processes (local to each participating machine) communicating by asynchronous messages. Collection does not interfere with applications by setting locks, polluting caches, or causing I/O; this requirement raised some novel and interesting challenges which we address in this article. The algorithm is safe and live; it is not complete, *i.e.* it collects some distributed cycles of garbage but not necessarily all.

1 Introduction

We present a system, Larchant, which provides a distributed and persistent store, intended for interactive cooperative tasks. A program shares data with others, possibly running at different sites and at different times, by mapping the Larchant store in memory via a Distributed Shared Memory (DSM) mechanism [18]. Programmers may concentrate on application development; low-level issues related to distribution, replication, coherence, input/output, and memory management are handled automatically. Thus, we call Larchant a Persistent Distributed Store; it consists essentially of a large-scale DSM that is persistently backed to disk and garbage collected.

1.1 Motivation

In a centralized program, sharing consists simply of using a pointer. So-called Single Address Space Operating Systems (SASOS) such as Monads [15], Opal

* This work was supported in part by the Esprit Project PerDiS N° 22533.

[8] or Grasshopper [10] extend this simple model elegantly to distribution and persistence. In a SASOS, an object is mapped at the same address in every process ever accessing it, ensuring that pointers remain valid across address spaces and time. It uses DSM techniques to ensure consistency of distributed replicas, and memory is mapped to backup storage for persistence. However the SASOS design has two flaws. First, since every object has a fixed address for all eternity, fragmentation of the store is a serious risk. Second, it relies on programmer discipline to deallocate objects properly.

Relying on programmer discipline to deallocate objects may lead to the deletion of an object that is still referenced. This would make the store *unsafe:* some other program may fail mysteriously when using the remaining reference, possibly much later in time. Such errors are very hard to detect, and when they are, it is too late. Furthermore, failure to delete unreachable objects causes memory leaks, which clog up the store persistently.

The deallocation problem is fixed by the model of *Persistence By Reachability* [3]. Programs have access to a *persistent root* (*e.g.*, a name server), from which they can *navigate* the pointer graph. Those objects that are transitively reachable from the persistent root must remain in persistent memory; any others are garbage and must be reclaimed. This is the task of the GC algorithm. Then, the fragmentation problem is solved by recycling storage and using a compacting GC.

GC techniques are well known in centralized systems [29]. Many researchers have proposed GC extensions to message-passing distributed systems [25]. In contrast, there is little previous work applicable to the problem of supporting PBR in a distributed cached store [1, 17, 19, 30] such as Larchant. This is a hard problem because:

- Applications modify the pointer graph concurrently by simply performing a pointer assignment. This is a very frequent operation, which should not be slowed down (by inserting reference counting code, for instance).
- 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 blow is unfeasible.
- A localized change to the pointer graph can affect remote portions of the graph. This has consequences on the global ordering of operations.
- The GC should not compete with applications. For instance, it should not take locks, cause coherence operations, or cause I/O.

GC in a large-scale distributed system is hard, especially with replication. It is tempting to apply standard consistency algorithms to the GC problem. For instance, one could layer a centralized GC algorithm above a coherent DSM, but this approach ignores the scalability and non-competition issues.

Some object-oriented databases run their collector as a transaction; this essentially blocks all useful work for the duration of the collection, and ignores the scalability issue. Another possible approach would be to collect a consistent

snapshot [7] off-line; this is correct because being garbage is a stable property; unfortunately it is an expensive and non-scalable solution.

1.2 Overview

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 (each partition is called a bunch; more details in Section 2.2), and partitions are replicated. GC is a hybrid of tracing within a partition and counting across partitions.
- Each site runs a collector with a standard tracing algorithm [29] that works in one or more partitions (on that site) at the same time.
- The cooperation protocol between collectors enables them to run without any mutual synchronization.
- 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.

This paper presents a distributed GC algorithm and a set of five simple rules ensuring its correctness. In particular, we show that, in this context, GC is safe if it conforms to the following rules (presented here informally):

- No collector may reclaim data until it has been declared unreachable at all replicas.
- A collector sends constructive (reachability) information before destructive (reclamation) information.
- All constructive information is sent.
- This information is received in the same order by remote collectors.
- The coherence protocol may propagate modified data only after it has been scanned by the local collector.

We prove our algorithm is safe, *i.e.* no reachable data is reclaimed; it is also live, *i.e.* some garbage is eventually reclaimed. Unfortunately, it is not *complete*, *i.e.* not all garbage is reclaimed (in particular, some distributed cycles) because completeness is at odds with scalability. An evaluation of the amount of unreclaimed garbage is the subject of on-going research [24]).

The contributions of this paper are the following. (i) A very simple, general model of a cached distributed shared store. (ii) Sufficient safety rules for GC in this context, in particular, for the interactions between coherence and GC. (iii) A distributed GC algorithm which is adapted to the model, avoids compiler modifications, is widely applicable, correct, scalable and efficient.

The outline of this paper is as follows. Section 2 presents our model of a distributed cached store. Section 3 describes the distributed GC algorithm and

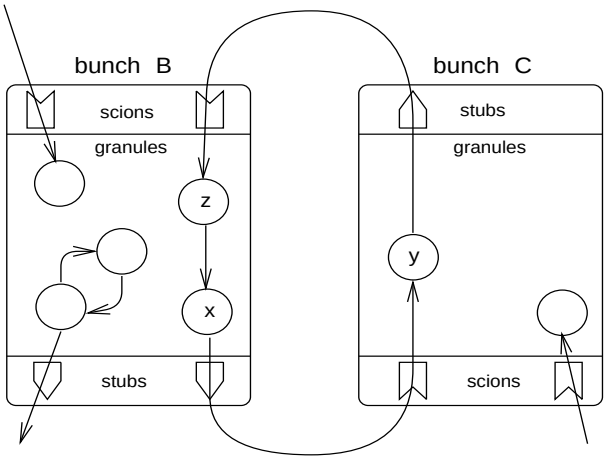

Fig. 1. *Two bunches containing granules, stubs, and scions.*

a set of safety rules for tracing garbage collection in the presence of replication. Section 4 compares our solution with related work. We summarize our contributions and future work in Section 5. Appendix A complements the main text; it provides a proof for the safety and liveness of the distributed GC algorithm.

For brevity, this paper omits some material without compromising its readability and correctness. For a more extensive treatment see Ferreira[12].

2 System model

In this section we present a general model for a garbage-collected distributed cached store, with sub-models for: network and processes, memory, coherence, mutator (application), and garbage collection. It incorporates only those elements that are relevant to the distributed garbage collection problem. It is extremely stylized with respect to any actual implementation.

Our model is based on a minimal set of coherence operations that are the same for any replicated memory. It does not dictate when such operations take place. (In a practical system, these operations are related to mutator activity.)

The model is also independent of the local garbage collection algorithm, pointer representation, or secondary storage technology. It applies to a large number of distributed sharing architectures, for instance a DSM, a SASOS, a client-server or peer-to-peer object-oriented database, or a client-server CORBA system.

2.1 Network and process model

The distributed system is composed of a set of sequential processes, communicating only by messages.

An event E that is atomic at some process i is noted $<E>_i$. For some message M, we note $<send.M>_i$ the sending event at process i, and $<deliver.M>_j$ the delivery of M at receiver process j. For GC safety, we assume causally-ordered delivery [5] of some messages; this will be justified in Section 3.2.

A process is composed of a mutator (application code), a collector, and a coherence engine. It's important to note that messages between processes flow only on behalf of collectors or coherence engines. Mutators do not send messages directly, *i.e.* they communicate only via updates to the shared memory. [3]

2.2 Memory model

The memory is structured at two levels of granularity (see Figure 1). (i) It is partitioned into (relatively large) *bunches*. A bunch is the unit of caching and tracing; it contains any number of granules. Each bunch may be traced independently of the others, which is what makes garbage collection scalable. (ii) The (rather small) *granule* is the unit of allocation, deallocation, identification, and coherence.[4] A granule resides entirely within a single bunch and allows fine-grained coherence.

A granule may contain any number of references, pointing to other granules. A reference may also be null (represented here as value zero), *i.e.* not pointing to anything. The model does not constrain the representation of references; for instance a raw pointer is supported.Hereafter, we indifferently use the words reference or pointer.

Bunches are noted X, Y, etc.; granules are noted x, y, etc. When x contains the address of y, x is said to point to y. To simplify the presentation, and without loss of generality, this article only considers pointers that cross bunch boundaries; the mechanisms for intra-bunch pointers are similar.

2.3 Coherence model

Bunches and granules are replicated in processes. The image of X (resp. x) in process i, noted X_i (resp. x_i), is called i's replica of X (resp. x). An invalidated replica is modeled by the null value.

In each process, a *coherence engine* provides the shared memory abstraction, by managing replicas of granules. The coherence engine sends and receives messages according to some coherence algorithm.

A process may disseminate the value of x to other processes, by sending propagate messages. Event $<send.propagate(x)>_i$ puts the current value of x_i in the message; event $<deliver.propagate(x)>_j$ assigns x_j with the value from the message. There is no assumption to which process, if any, or in what order propagate messages are delivered.

[3] If mutators were allowed to exchange messages, the pointers they contain must be taken into account by the GC algorithm. This is probably straightforward, using techniques such as SSP Chains [26], but has not been considered yet.

[4] It is convenient to think of a granule as an object, but note that granules are not necessarily the same as language-level objects, which can be larger or smaller.

After a granule replica changes value, either by assignment or by being the target of a propagate, it is said *GC-dirty*. A replica remains GC-dirty until it is subjected to a scan operation (see Section 2.5).

Many coherence algorithms define a *owner* process for a granule. Ownership does not appear in our model; however, in such coherence algorithms, the "Union Rule" (presented in Section 3.1) can make use of the properties of owners for a more efficient implementation.

The coherence model presented above is unconstrained enough to apply to any cached or replicated architecture. For concreteness, we show the mapping of entry consistency [4], the coherence protocol used in the current implementation of Larchant, to the model just described.

Mapping entry consistency to the coherence model Entry consistency uses *tokens* to schedule access to shared data and to ensure its consistency. A program precedes every use of a shared variable with an acquire primitive, and follows it with the corresponding release. Acquire asks for a token, and is parameterized with the type of access, either read or write; the protocol maintains a single writer, multiple readers semantics.

At any point in time, each granule has a single *owner* which is defined as the process that holds the write token or was the last one to hold it.

Sending a token also sends the most current version of the granule. Only the owner may send the write token. Sending a write token (i) invalidates the sender's replica and any reader tokens and replicas, (ii) sends the current value, and (iii) transfers ownership to the receiver. The owner may transform its write token into a read token. Any holder of a read token may send another read token (along with the granule value) to another process.

In this protocol, acquire messages, invalidation messages, and their replies are all modeled by a propagate. More precisely, the sending of an acquire message and the corresponding reply, is modeled as a single propagate (from the owner to the acquiring process) in which the granule's data is sent within the message. Invalidation of a replica is equivalent to a propagate message in which the granule's contents is null; thus, once a granule replica becomes invalid it contains no pointer (it is equivalent to a spontaneous, local assignment of the whole granule's data to the value zero).

2.4 Mutator model

For the purpose of GC, the only relevant action of the mutator is *pointer assignment*, which modifies the reference graph, possibly causing some granules to become unreachable. An application may arbitrarily assign a pointer with any legal reference value.

Suppose x points to t and y points to z. The result of assignment $\langle x := y \rangle_i$ is that x now also points to z, and the previous pointer to t is lost. (The replica that appears on the left-hand side of an assignment thereafter becomes GC-dirty.) This operation is atomic at process i, which only means that the model

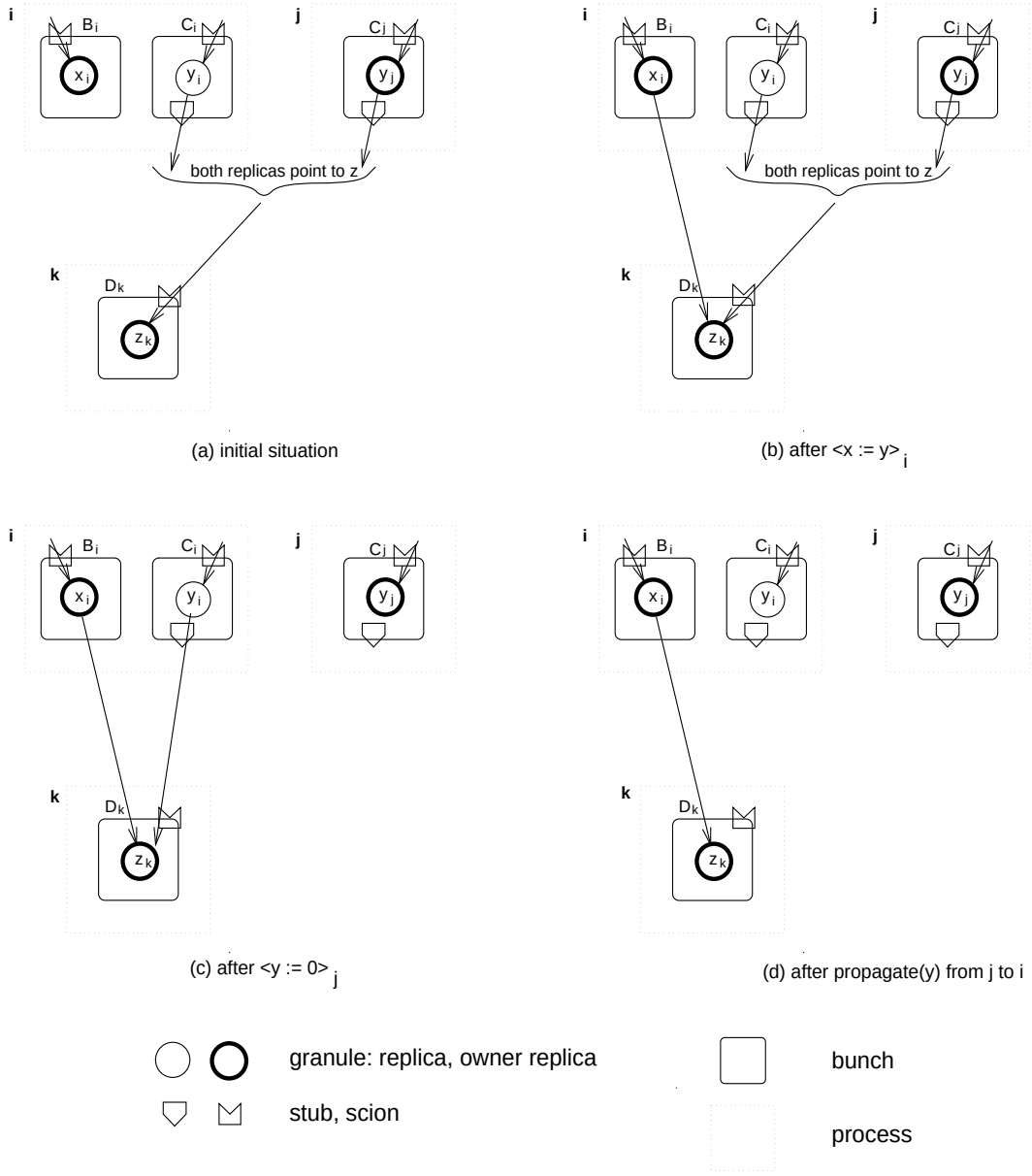

Fig. 2. *Prototypical example of mutator execution. Note that the stubs and scions become temporarily inconsistent with the pointers. However, as described in the paper, this does not compromise safety.*

does not allow hidden pointers. Granule creation (by some primitive similar to Unix malloc) is taken as a special case of pointer assignment.

Our reasoning is based on the following *prototypical example* (see Figure 2). Consider two granules x and y located in bunches X and Y, respectively. Initially x is null and y points to granule z located in bunch Z. Now, mutators within processes i and j execute the following operations: $<x := y>_i$, $<y := 0>_j$, such that at the end x points to z and all the replicas of y are null.

2.5 GC model

There are two well-known families of GC algorithms [29]. The first, counting, is invoked each time a reference is assigned; it scales well because it only involves the process performing the assignment, and the granule(s) referred to. It is considered expensive and non-portable because the counting operation is inserted inline by the compiler; furthermore it does not reclaim cycles of garbage. The second, tracing, is a global operation on the whole memory, and therefore scales poorly.

Many distributed GCs [25] are hybrids, in an attempt to combine the best of both worlds. Hybrid GCs partition the memory: they trace references internal to a partition, and count references that cross partition boundaries. This replaces the unfeasible global trace with the weaker problem of tracing each partition.[5]

In previous distributed GCs, a partition was often identified with a process [26]. This is a natural design option as those GCs were conceived for distributed systems based on RPC (Remote Procedure Call) in which cross-partition pointers were tracked at the process border. However, Larchant is based on DSM, therefore, memory partitioning is different: (i) a bunch may be replicated in multiple processes; (ii) a trace partition contains multiple bunches, in order to reclaim cross-bunch cycles of garbage; (iii) it will do so opportunistically in order to avoid input/output. This article focuses on the issues associated with point (i) with emphasis on the algorithm and its safety; points (ii) and (iii) have been studied in previous articles [13, 27] and are out of the scope of this paper. Fault-tolerance is also out of the scope; however, there are well known solutions that can be applied to our system [20].

In order to make the tracing of a bunch independent from other bunches, each bunch replica is provided with data structures that describe the references that cross its boundaries (see Figure 1): a *stub* describes an outgoing reference, a *scion* an incoming one; scions have an associated counter that counts references to some granule.[6] Note that Larchant's stubs and scions are not indirections participating in the mutator computation, but auxiliary data structures of the collector.

[5] Of course, the weaker formulation does not allow to collect cross-partition cycles of garbage.

[6] For simplicity, we speak of reference counting. In reality, we use the "reference listing" variant: instead of just a count, there is a list of scions, one per bunch containing a pointer to this granule, that records the identity of the originating bunch. This makes the increment messages idempotent, *i.e.* redundant increments have no effect.

A granule x in bunch X is said *protected* if and only if x is pointed directly or indirectly by a scion in X.

In the case of a reference from x contained in X, to z contained in Z, the stub is noted stub(Xx, Zz) and the corresponding scion scion(Xx, Zz).

The elementary collection operation is scan. At times a single granule replica x_i is scanned. Operation $scan_i(x)$ returns the list of granules z, t, ..., that are pointed to by x_i. At other times, a whole bunch replica X_i is traced, noted $trace_i(X)$, scanning all reachable granules that it contains. The roots of the trace are the scions in X_i.

A trace of X_i produces: (i) a set of granules contained in X_i, transitively reachable from its scions, and (ii) a set of stubs describing the outgoing pointers of the set of granules (mentioned in the previous point). Then, this generated set of stubs can be compared to the previous set (before the trace) in order to find which scions should have its counter incremented or decremented. This will be addressed with more detail in Section 3.2.

A GC-dirty replica remains GC-dirty until scanned. Then, *i.e.* after being scanned, it becomes GC-clean.

For concreteness, now we show the mapping of a mark-and-sweep collector [22], one of the GC algorithms used in the current implementation of Larchant, to the model just described.

Mapping mark-and-sweep to the GC model A mark-and-sweep collector has two phases: *(i)* trace the pointer graph starting from the root and mark every granule found, and *(ii)* sweep (*i.e.*, examine) all the heap reclaiming unmarked granules.

During the mark phase every reachable granule is marked (setting a bit in the granule's header, for example) and scanned for pointers. This phase ends when there are no more reachable granules to mark.

During the sweep phase the collector detects which granules are not marked and inserts their memory space in the free-list. When the collector finds a marked granule it unmarks it in order to make it ready for the next collection. This phase ends when there is no more memory to be swept.

Our model describes this algorithm through the operations scan and trace. The first operation models the scanning for pointers on each reachable granule, during the mark phase. The second operation, trace, models the entire mark phase in which all reachable granules are found (*i.e.*, marked and scanned).

3 GC algorithm and safety rules

Having exposed our model, we now turn to the solution of the main issues. This section describes the distributed GC algorithm, *i.e.* tracing a bunch and counting references that cross bunch boundaries, and the corresponding safety rules.

Tracing a bunch causes stubs to be created or deleted; the purpose of the counting algorithm is to adjust scions accordingly. As a simplification, we address

the tracing of a single bunch, ignoring the fact that a partition may in fact contain any number of bunches in order to collect cross-bunch cycles of garbage [27].

When a mutator performs an assignment such as $<x := y>_i$, up to three processes are involved in the corresponding counting. Say granules 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 $<\mathsf{create.stub}(Xx, Tt)>_i$ and sending message $\mathsf{increment.scion}(Xx, Tt)$ to the collector in some process j, managing T. In addition, it decrements the reference count for z by performing the local operation $\mathsf{delete.stub}(Xx, Zz)$, and sending message $\mathsf{decrement.scion}(Xx, Zz)$ to the collector in some process k, managing Z.

An obvious solution to perform reference counting would be to instrument applications in order to track every pointer assignment. However, this solution would be extremely expensive in terms of performance because pointer assignment is a very frequent operation and counting might imply remote communication. This leads us to a fundamental insight of our design: instead of instrumenting the application, as suggested above, we observe that *counting may be deferred to a later tracing*. This removes counting from the application path, avoids reliance on compiler support, and enables batching optimizations. It in turn puts requirements on tracing, which will be addressed in the following sections.

3.1 Tracing in the presence of replicas: the union rule

Each process runs a standard centralized 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 probably 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 a granule until it is observed unreachable in the union of all replicas. This is captured by the following rule.

> **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 granule pointed only by an unreachable replica x_i, if some other replica x_j is reachable. This very conservative formulation is necessary in the absence of knowledge of the coherence algorithm.

It's worthy to note that, instead of adopting the Union Rule, we could consider each replica as a separate granule, implicitly connected to its peer replicas.

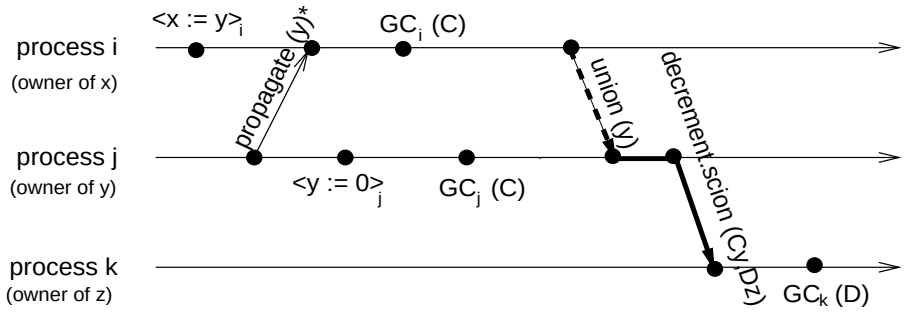

Fig. 3. *Prototypical example of Figure 2 restrained to the case of entry consistency.*

Pointers from different replicas would be counted separately. For instance, suppose x has three replicas, two of which point to z; then the reference count of z is at least 2. However, this solution has the drawback that every replica of x must send a number of counting messages.

A simpler and more efficient solution applies to coherence protocols where a granule has a single owner process (such as entry consistency, for example). The collectors centralize 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 0). 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 granules where references originate.

For concreteness, in the next section we show how the Union Rule can be enforced in a system supporting a specific coherence protocol, entry consistency.

Adapting the union rule to entry consistency In some DSM coherence protocols (for instance, sequential consistency and entry consistency) only the current owner of granule x is allowed to assign to x. In this case, a non-owner replica x_i cannot cause a granule unreachable from x_i to become reachable (because to do so would require assigning to x_i). Then, the implementation of the Union Rule is straightforward as explained now.

Consider Figure 3; it illustrates the prototypical example of Figure 2 re-

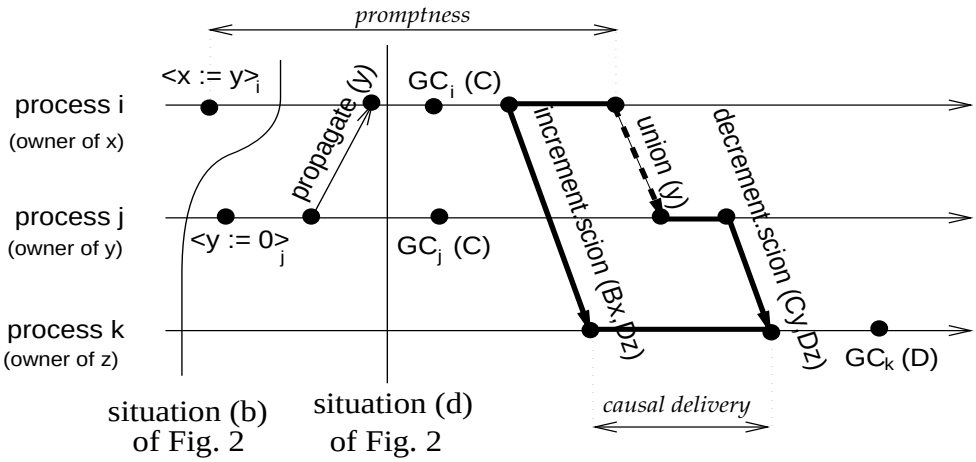

Fig. 4. *Timeline showing the effect of some of the safety rules for the prototypical example of Figure 2. On site* i, *the sending of* increment.scion(Bx, Dz) *may be delayed at most until sending* union(y). *Note the causal dependence (indicated by the thick lines) between the* increment.scion *and* decrement.scion *messages, carried by the* union *message.*

strained to a system with the entry consistency coherence protocol. The owner of y (site j) maintains a *copy-count* for each stub corresponding to a pointer originating in y. Let us consider the stub for a pointer from y to z; the stub's copy-count is equal to the number of replicas of y. Now, suppose the mutator at j performs $\langle y := 0 \rangle_j$. Note that, given the entry consistency protocol, previous to this assignment all replicas of y have been invalidated (in particular, y_i). Later, on site i, when the collector discovers that y_i does not point to z (because the invalidation has nulled the whole data of y_i), it sends a union message to j. This decrements the corresponding stub's copy-count. After all the collectors (whose processes cache a replica of y) have sent their union message to j, the stub's copy-count at j is zero; therefore, a decrement.scion(Cy, Dz) is sent from j to the owner of z (site k) in order to decrement the reference count of scion(Cy, Dz).

An attentive reader could argue that there is a simpler solution, in which the Union Rule is not needed, because as soon as a granule replica is invalidated we safely know that such a replica does not point to any other granule. We now describe this (apparent) solution and show why it is wrong.

When the mutator at j performs $\langle y := 0 \rangle_j$ all the other replicas of y are invalid, thus they contain no pointer to any granule, in particular y_i does not point to z anymore. For this reason, as soon as $\langle y := 0 \rangle_j$ is performed, the collector at j could safely reach the conclusion that there is no replica of y that still points to z. This is in fact true. However, note that this does not mean that decrement.scion(Cy, Dz) can be safely sent from j to k. The reason is that on site i, before y_i has became invalid, the mutator might have performed $\langle x := y \rangle_i$

(as indicated in Figure 3) therefore creating a new pointer to z. In this case, performing decrement.scion(Cy, Dz) could lead to the deletion of the last scion to z and later to its reclamation. This situation would be an error because there is still a reference from x_i to z.

Note that this scenario shows the Union Rule is effectively needed. In addition, it raises a safety problem which is the following: how to guarantee that increment.scion(Bx, Dz) reaches site k before decrement.scion(Cy, Dz). In the next section we provide a solution to this problem, and present a few more safety rules which are valid for any coherence protocol.

3.2 Cross-bunch counting and more safety rules

As already mentioned, 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 assignment $<x := y>_i$, 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 prototypical example of Figure 2. At the time of $<x := y>_i$, granule 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, that causes the corresponding scions count to be adjusted.)

There is a problem with this approach, however. In the prototypical 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 4: the interval named *promptness*, shows how much the message increment.scion(Bx, Dz) can be delayed w.r.t. the moment when the corresponding assignment operation ($<x := y>_i$) has been performed.

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

> **Safety Condition II: Increment Before Decrement Rule.** *Scanning a granule (i.e., making it GC-clean) 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 be GC-clean.*

Safety Condition IV: Clean Propagation Rule. *When process* i *sends* propagate(x), x_i *must be GC-clean.*

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

Rule II allows a granule replica to be scanned at any time; scanning a granule that contains a new pointer immediately sends an increment.scion message to the referent. It's important to mention that any message is asynchronous, so its actual transmission might take place later, as long as messages are delivered in the order sent (more on this later).[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 granule 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 2 and 4 after mutators have executed $<x := y>_i$, $<y := 0>_j$, and y propagated to site i, *i.e.* once situation (d) has been reached. (Note that scion(Bx, Dz) has not been created yet.) Suppose that $\text{trace}_i(C)$ 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 stub(Cy, Dz) has disappeared in process i, but $\text{scan}_i(x)$ is not performed; therefore increment.scion(Bx, Dz) is not sent. As a result, j applies the Union Rule and $<$send.decrement.scion(Cy, Dz)$>_j$ is executed. Thus, scion(Cy, Dz) is deleted in k. Then, if $\text{trace}_k(D)$ runs, granule z is unsafely reclaimed.

[7] In fact, sending a message immediately means put it in the sending queue.

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

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 2 and 4 after the mutator has executed <x := y>$_i$ and before <y := 0>$_j$, *i.e.* once situation (b) has been reached. Now, suppose that granule x_i is propagated to some process w in the absence of the Clean Propagation Rule, *i.e.* without performing scan$_i$(x). Then, x_i is assigned in such a way that it no longer points to z (*e.g.*, <x := 0>$_i$). At this moment, the only scion that protects z is 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, scion(Cy, Dz) is deleted. Thus, z may be unsafely reclaimed by trace$_k$(D) (x_w still points to z).

The Clean Propagation Rule prevents the above scenario as it forces x_i to be GC-cleaned. Thus, by Rule II, <send.increment.scion(Bx, Dz)>$_i$ is performed immediately, *i.e.* before 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 prototypical example of Figure 2 after mutators have executed <x := y>$_i$, <y := 0>$_j$, and y propagated to site i, *i.e.* once situation (d) has been reached. (Note that scion(Bx, Dz) has not been created yet.) Then, the collectors on sites i and j perform as follows: i executes <send.increment.scion(Bx, Dz)>$_i$, while j executes <send.decrement.scion(Cy, Dz)>$_j$. 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 4. Thus, given the Causal Delivery Rule, there is at all times at least a scion protecting z from being reclaimed.

4 Related work

The concept of persistence by reachability (PBR) was first proposed by Atkinson [3] in the early 1980's. EOS [14] is an early example of a DSM providing PBR. It has a copying GC that takes into account user placement hints to improve locality. However, its GC is quite complex and has not been implemented.

Much previous work in distributed garbage collection, such as SSP Chains [26] or Network Objects [6], considers processes communicating via RPC, and

uses a hybrid of tracing and counting. Each process traces its internal pointers; references across process boundaries are counted as they are sent/received in messages.

Some object-oriented databases (OODB) use a similar approach [1, 9, 23, 30] *i.e.* a partition can be collected independently from the rest of the database. In particular, Thor [20, 21] is a research OODB with PBR. In Thor, data resides at a small number of servers and is cached at workstations for processing. A Thor server counts references contained in objects cached at a client. Thor defers counting references originating from some object x cached at a client, until x is modified at the server. However, their work does not address the issue of GC interaction with coherence and replication in a DSM based system.

The work most directly related to ours is Skubiszewski and Porteix's GC-consistent cuts [28]. They consider asynchronous tracing of an OODB; however they consider neither distribution nor replication. The collector is allowed to trace an arbitrary database page at any time, subject to the following ordering rule. For every transaction accessing a page traced by the collector, if the transaction copies a pointer from one page to another, the collector either traces the source page before the write, or traces both the source and the destination page after the write. The authors prove that this is a sufficient condition for safety and liveness.

It's worthy to note that the solutions mentioned above do not consider a DSM based system, such as Larchant, in which cross-partition pointers are created by a simple assignment operation.

Most previous work on garbage collection in shared memory deals either with multiprocessors [2, 11] or with a small-scale DSM [17]. These authors make strong coherence assumptions, and they ignore the issues of scale and of persistence.

Yu and Cox [31] describe a conservative collector for the TreadMarks [16] DSM system. It uses partitioned GC on a process basis; all messages are scanned for possible contained pointers. Like Larchant, their GC does not rely on coherence. However, their solution is specific to TreadMarks, *i.e.* it is not widely applicable.

5 Conclusion

Larchant is a Cached Distributed Shared Store based on the model of a DSM with Persistence By Reachability. Data is replicated in multiple sites for performance and availability. Reachability is assessed by tracing the pointer graph, starting from the persistent root, and reclaiming unreachable data. This is the task of Garbage Collection.

This paper focused on the interactions between garbage collection on the one hand, and caching and replication on the other. We show that both the tracing and the distributed counting garbage collector run 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.

Using an extremely stylized model of an application (reduced to unconstrained pointer assignments) and of a coherence protocol (reduced to unconstrained propagation messages), we give rules for the safe coexistence of garbage collection with replication.

Our GC is a hybrid (or partitioned) algorithm. It combines tracing within a partition, with reference-counting across partition boundaries. Each process may trace its own replicas, independently of one another and of other replicas. Counting at some process is asynchronous to other processes, and asynchronous 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. Along with these rules, we provided a proof that the algorithm is safe and live (see Appendix A). These safety rules are minimal and generally applicable given the asynchrony to applications, and the minimum assumptions we made concerning coherence:

- Union Rule: a granule may be reclaimed only if it is unreachable from the union of all replicas (of the pointing granules);
- Increment before Decrement Rule: when a granule is scanned, the corresponding increment.scion messages must be sent immediately (*i.e.*, put them 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 GC-clean;
- Clean Propagation Rule: a granule 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, which can be found in Ferreira[12], 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.

We are currently working on a follow-up of Larchant in Esprit Project PerDiS [24], where it will be used for 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. A first prototype of the PerDiS implementation is freely available in http://www.perdis.esprit.ec.org.

References

1. L. Amsaleg, M. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Client-Server Object Database Systems. In *Proc. of the 21th VLDB Int. Conf.*, Zürich, Switzerland, September 1995.

2. Andrew W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19(2):171–183, February 1989.
3. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.
4. B. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 1993 CompCon Conf.*, 1993.
5. Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
6. Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software Practice and Experience*, S4(25):87–130, December 1995. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-115.html.
7. 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.
8. J. S. Chase, H. E. Levy, M. J. Feely, and E. D. Lazowska. Sharing and adressing in a single address space system. *ACM Transactions on Computer Systems*, 12(3), November 1994.
9. Jonathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition selection policies in object database garbage collection. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 371–382, Minneapolis MN (USA), May 1994. ACM SIGMOD.
10. Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.
11. Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, pages 113–123, Charleston SC (USA), January 1993.
12. 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/SOR/docs/ferreira_thesis96.html.
13. 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/SOR/docs/GC-DSM-CONSIS_OSDI94.html.
14. Olivier Gruber and Laurent Amsaleg. Object grouping in EOS. In *Proc. Int. Workshop on Distributed Object Management*, pages 184–201, Edmonton (Canada), August 1992.
15. James Leslie Keedy. Support for objects in the MONADS architecture. In J. Rosenberg, editor, *Proc. Workshop on persistent object systems*, pages 202–213, Newcastle NSW (Australia), January 1989.
16. P. Keleher, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. *Proceedings of the 1994 Winter USENIX Conference*, January 1994.
17. 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.

18. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
19. 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.
20. Umesh Maheshwari and Barbara Liskov. Fault-tolerant distributed garbage collection in a client-server, object database. In *Proc. Parallel and Dist. Info. Sys.*, pages 239–248, Austin TX (USA), September 1994. ftp://pion.lcs.mit.edu/pub/thor/dgc.ps.gz.
21. Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, Montreal, Canada, 1996.
22. John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960.
23. J. Eliot B. Moss, David S. Munro, and Richard L. Hudson. PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proc. of the 6th Int. Workshop on Persistent Object Systems*, Cape May NJ (USA), May 1996.
24. PerDiS ESPRIT Project - LTR 22533. The PerDiS project: a Persistent Distributed Store, 1997. http://www.perdis.esprit.ec.org.
25. 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/SOR/docs/SDGC_iwmm95.html.
26. Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de Recherche en Informatique et Automatique, Rocquencourt (France), November 1992. http://www-sor.inria.fr/SOR/docs/SSPC_rr1799.html.
27. 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/SOR/docs/LRDSPMGC_wdag95.html.
28. Marcin Skubiszewski and Patrick Valduriez. Concurrent garbage collection in O_2. In *24th International Conference on Very Large Data Bases*, Athens, Greece, 1997.
29. 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.
30. 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.
31. Weimin Yu and Alan Cox. Conservative garbage collection on distributed shared memory systems. In *16th Int. Conf. on Distributed Computing Syst.*, pages 402–410, Hong Kong, May 1996. IEEE Computer Society.

A Proofs of safety and liveness

This appendix provides a proof that the distributed GC algorithm is safe and live.

A.1 Safety

To understand this proof, note that:

- the phrase "scion(Xx, Yy) exists" means that the reference count of the scion is non-zero;
- event <deliver.decrement.scion(Xx, Yy)>$_j$ deletes scion(Xx, Yy) at j if and only if the scion's counter becomes zero as a result of the decrement.scion message;
- event <deliver.increment.scion(Xx, Yy)>$_j$ creates scion(Xx, Yy) at j if and only if that scion does not exist yet; otherwise, it increments that scion's counter;
- a granule is created with an initial scion (this is guaranteed by the granule creation primitive) in order to ensure that the Base Case assumptions, in our proofs, are always verified;
- we assume a coherence protocol in which each granule has a owner as defined in Section 2.3; and
- we represent a pointer from granule x in bunch B to granule z in bunch D as $\mathsf{Bx} \rightarrow \mathsf{Dz}$.

The distributed GC algorithm must satisfy the following obvious safety invariant:

Safety Invariant 1 *No reachable granule is reclaimed.*

Since we are considering only cross-bunch pointers, the above invariant is equivalent to:

Safety Invariant 2 *:*

$$(\forall \mathsf{B}, \forall \mathsf{x} \in \mathsf{B} : \exists \mathsf{Bx} \rightarrow \mathsf{Dz}) \Rightarrow (\exists \mathsf{T}, \exists \mathsf{t} \in \mathsf{T} : \exists \mathsf{scion}(\mathsf{Tt}, \mathsf{Dz}))$$

This reads "if a granule x in B points to z in D, then some scion protects z." Note that this is weaker than the more intuitive "if x points to z, then scion(Bx, Dz) exists"; indeed, the scion that protects z does not need to match the pointer.

We prove the safety of the distributed GC algorithm by showing that it maintains the above safety invariant. We start by proving two lemmas; these will be needed to prove a theorem.

Lemma 1 *Let n different granules* $\mathsf{x}^1, \ldots, \mathsf{x}^n$ *located respectively in bunches* $\mathsf{B}^1, \ldots, \mathsf{B}^n$*, all cached in process* p*, be the set of all protected granules pointing to* $\mathsf{z} \in \mathsf{D}$ *cached in process* k*. Then,* $\forall i, 1 \leq i \leq n$*:*

- x^i *is GC-clean* $\wedge \exists$ $\mathsf{scion}(\mathsf{B}^i\mathsf{x}^i, \mathsf{Dz})$*, or*
- $\exists j, 1 \leq j \leq n$, x^i *is GC-dirty* $\wedge \exists$ $\mathsf{scion}(\mathsf{B}^j\mathsf{x}^j, \mathsf{Dz})$

This lemma says that as long as there is a granule pointing to z, cached in some process p, there exists at least a scion protecting z. We prove this lemma by induction over the size of the set containing granules pointing to z, for a single process.

Base case:
Initially a single protected granule x^1 points to z: $x^1 \in B^1$ cached in process p, and scion(B^1x^1, Dz) exists. Let $x^2 \in B^2$, also cached in p, initially not pointing to z This initial situation obviously verifies the lemma as x^1 is GC-clean and scion(B^1x^1, Dz) exists.

Now, consider that $<x^2 := x^1>_p$ is performed (now x^2 also points to z); therefore x^2 is GC-dirty. If x^1 is not assigned to thereafter, the lemma remains trivially true since x^2 is GC-dirty and scion(B^1x^1, Dz) exists.

Therefore consider that x^1 is changed by an assignment such as $<x^1 := 0>_p$. (The actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Hence, x^1 is GC-dirty. Until a bunch tracing takes place at p, no <send.increment.scion> or <send.decrement.scion> is performed at p, and the lemma remains trivially true since x^2 is GC-dirty and scion(B^1x^1, Dz) exists.

When a bunch tracing does execute in process p, both x^2 (containing the new pointer) and x^1 (previously containing the pointer to z) are GC-cleaned (they were both GC-dirty); therefore stub(B^2x^2, Dz) is created and stub(B^1x^1, Dz) disappears (*i.e.*, it is no longer in the new set of stubs).

According to Rule II (Increment Before Decrement) and Rule III (Comprehensive Tracing), <send.increment.scion(B^2x^2, Dz)>$_p$ precedes <send.decrement.scion(B^1x^1, Dz)>$_p$.

We assumed that messages are delivered in causal order (Rule V), hence <deliver.increment.scion(B^2x^2, Dz)>$_k$ precedes <deliver.decrement.scion(B^1x^1, Dz)>$_k$. Thus, at any moment, there exists at least a scion protecting z: either scion(B^1x^1, Dz), or scion(B^2x^2, Dz), or both.

Induction case:
Assume a set of different granules $x^1, \ldots, x^j$ located respectively in bunches $B^1, \ldots, B^j$ all cached in process p, pointing to z with the corresponding scions already created. Consequently, granule x^l, $1 \leq l \leq j$, points to z: $x^l \in B^l$ cached in process p, and scion(B^lx^l, Dz) exists. This initial situation obviously verifies the lemma as x^l is GC-clean and scion(B^lx^l, Dz) exists.

Let $x^{j+1} \in B^{j+1}$, also cached in p, initially not pointing to z. Now, consider that $<x^{j+1} := x^l>_p$ is performed (thus, x^{j+1} also points to z) therefore x^{j+1} is GC-dirty. If x^l is not assigned to thereafter, the lemma remains trivially true since x^{j+1} is GC-dirty and scion(B^lx^l, Dz) exists.

Therefore consider that x^l is changed by an assignment such as $<x^l := 0>_p$. (The actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Hence, x^l is GC-dirty. Until a bunch tracing takes place at p, no <send.increment.scion> or <send.decrement.scion> is performed at p, and the lemma remains trivially true since x^{j+1} is GC-dirty and scion(B^lx^l, Dz) exists.

When a bunch tracing does execute in process p, both x^{j+1} (containing the new pointer) and x^{l} (previously containing the pointer to z) are GC-cleaned (they were both GC-dirty); therefore $\mathsf{stub}(\mathsf{B}^{j+1}\mathsf{x}^{j+1}, \mathsf{Dz})$ is created and $\mathsf{stub}(\mathsf{B}^{l}\mathsf{x}^{l}, \mathsf{Dz})$ disappears (*i.e.*, no longer in the new set of stubs).

According to Rule II (Increment Before Decrement) and Rule III (Comprehensive Tracing), event $<\mathsf{send.increment.scion}(\mathsf{B}^{j+1}\mathsf{x}^{j+1}, \mathsf{Dz})>_{\mathsf{p}}$ precedes $<\mathsf{send.decrement.scion}(\mathsf{B}^{l}\mathsf{x}^{l}, \mathsf{Dz})>_{\mathsf{p}}$.

We assumed that messages are delivered in causal order (Rule V), hence $<\mathsf{deliver.increment.scion}(\mathsf{B}^{j+1}\mathsf{x}^{j+1}, \mathsf{Dz})>_{\mathsf{k}}$ precedes $<\mathsf{deliver.decrement.scion}(\mathsf{B}^{l}\mathsf{x}^{l}, \mathsf{Dz})>_{\mathsf{k}}$. Thus, at any moment, there exists at least a scion protecting z: either $\mathsf{scion}(\mathsf{B}^{l}\mathsf{x}^{l}, \mathsf{Dz})$, or $\mathsf{scion}(\mathsf{B}^{j+1}\mathsf{x}^{j+1}, \mathsf{Dz})$, or both.

This terminates the induction over the size of the set containing granules pointing to z, for a single process. □

Now, we present the second lemma.

Lemma 2 *Let n different granules $\mathsf{x}^1, \ldots, \mathsf{x}^n$ be the set of all protected granules, located in bunch B, pointing to $\mathsf{z} \in \mathsf{D}$ owned by k. Let m different processes $\mathsf{p}^1, \ldots, \mathsf{p}^m$ be the set of all processes caching a replica of x^i. Then, $\forall i, 1 \leq i \leq n$:*

- x^i *is GC-clean* $\wedge\ \exists\ \mathsf{scion}(\mathsf{Bx}^i, \mathsf{Dz})$, *or*
- $\exists j, 1 \leq j \leq n$, x^i *is GC-dirty* $\wedge\ \exists\ \mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$

This lemma says that, as long as there is a replica of some granule x located in B, pointing to z, cached in some process, there exists a scion protecting z. We prove this lemma by induction: *(i)* over the size of the set containing granules pointing to z, and *(ii)* over the size of the set containing processes caching replicas of the granules pointing to z.

Base case:
Initially there is a single granule $\mathsf{x}^1 \in \mathsf{B}$ pointing to z, x^1 is owned by process p^1, $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$ exists, $\mathsf{x}^2 \in \mathsf{B}$ does not point to z and is also owned by p^1, and neither x^1 nor x^2 are cached in p^2. This initial situation obviously verifies the lemma as x^1 is GC-clean and $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$ exists.

Now, perform $<\mathsf{x}^2 := \mathsf{x}^1>_{\mathsf{p}^1}$; thus, x^2 points to z and is GC-dirty. If x^1 is not assigned to thereafter, the lemma remains trivially true as x^1 is GC-clean and $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$ exists.

Therefore, consider that x^1 is changed by an assignment such as $<\mathsf{x}^1 := 0>_{\mathsf{p}^1}$. (The actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Hence, x^1 is GC-dirty. Until a bunch tracing or a propagate operation takes place at p^1, no $<\mathsf{send.increment.scion}>$ or $<\mathsf{send.decrement.scion}>$ is performed at p^1, and the lemma remains trivially true since x^2 is GC-dirty and $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$ exists. We examine these two cases now: bunch tracing and propagate operation, both performed at p^1.

When a bunch tracing does execute in process p^1, both x^2 (containing the new pointer) and x^1 (previously containing the pointer to z) are GC-cleaned (they were both GC-dirty); therefore $\mathsf{stub}(\mathsf{Bx}^2, \mathsf{Dz})$ is created and $\mathsf{stub}(\mathsf{Bx}^1, \mathsf{Dz})$ disappears (*i.e.*, no longer in the new set of stubs).

According to Rule II (Increment Before Decrement) and Rule III (Comprehensive Tracing), event $<\mathsf{send.increment.scion}(\mathsf{Bx}^2, \mathsf{Dz})>_{\mathsf{p}^1}$ precedes $<\mathsf{send.decrement.scion}(\mathsf{Bx}^1, \mathsf{Dz})>_{\mathsf{p}^1}$.

We assumed that messages are delivered in causal order (Rule V), hence $<\mathsf{deliver.increment.scion}(\mathsf{Bx}^2, \mathsf{Dz})>_{\mathsf{k}}$ precedes $<\mathsf{deliver.decrement.scion}(\mathsf{Bx}^1, \mathsf{Dz})>_{\mathsf{k}}$. Thus, at any moment, there exists at least a scion protecting z: either $\mathsf{scion}(\mathsf{Bx}^1, \mathsf{Dz})$, or $\mathsf{scion}(\mathsf{Bx}^2, \mathsf{Dz})$, or both. Therefore, the lemma remains true when a bunch collection occurs in process p^1.

Now, we consider a propagate operation. Before a $<\mathsf{send.propagate}(\mathsf{x}^2)>_{\mathsf{p}^1}$ (possibly sent to p^2) takes place,[8] according to Rule IV (Clean Propagation), x^2 is GC-cleaned. Therefore x^2 is scanned, $\mathsf{stub}(\mathsf{Bx}^2, \mathsf{Dz})$ is created and $<\mathsf{send.increment.scion}(\mathsf{Bx}^2, \mathsf{Dz})>_{\mathsf{p}^1}$ is performed. Therefore, the lemma remains true as x^2 is GC-clean and $\mathsf{scion}(\mathsf{Bx}^2, \mathsf{Dz})$ exists.

Induction case:
Let h different process $\mathsf{p}^1, \ldots, \mathsf{p}^h, 1 \leq h \leq m$ be the only processes caching granules $\mathsf{x}^1, \ldots, \mathsf{x}^j, 1 \leq j \leq n$, all $\in \mathsf{B}$, all pointing to $\mathsf{z} \in \mathsf{D}$ owned by k. The scions for the pointers from $\mathsf{x}^1, \ldots, \mathsf{x}^j$ to z do exist. Granule x^j is owned by p^w, $1 \leq w \leq m$. Let granule x^{j+1} owned by p^w, initially not pointing to z, and process p^{h+1} initially not caching any granule pointing to z. This initial situation obviously verifies the lemma as x^j is GC-clean and $\mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$ exists.

Now, perform $<\mathsf{x}^{j+1} := \mathsf{x}^j>_{\mathsf{p}^w}$; thus, x^{j+1} points to z and is GC-dirty. If x^j is not assigned to thereafter, the lemma remains trivially true as x^j is GC-clean and $\mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$ exists.

Therefore, consider that x^j is changed by an assignment such as $<\mathsf{x}^j := 0>_{\mathsf{p}^w}$. (The actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Hence, x^j is GC-dirty. Until a bunch tracing or a propagate operation takes place at p^w, no $<\mathsf{send.increment.scion}>$ or $<\mathsf{send.decrement.scion}>$ is performed at p^w, and the lemma remains trivially true since x^{j+1} is GC-dirty and $\mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$ exists. We examine these two cases now: bunch tracing and propagate operation, both performed at p^w.

When a bunch tracing does execute in process p^w, x^{j+1} (containing the new pointer) and x^j (previously containing the pointer to z) are GC-cleaned (they were both GC-dirty); therefore $\mathsf{stub}(\mathsf{Bx}^{j+1}, \mathsf{Dz})$ is created and $\mathsf{stub}(\mathsf{Bx}^j, \mathsf{Dz})$ disappears (*i.e.*, no longer in the new set of stubs).

According to Rule II (Increment Before Decrement) and Rule III (Comprehensive Tracing), event $<\mathsf{send.increment.scion}(\mathsf{Bx}^{j+1}, \mathsf{Dz})>_{\mathsf{p}^1}$ precedes $<\mathsf{send.decrement.scion}(\mathsf{Bx}^j, \mathsf{Dz})>_{\mathsf{p}^1}$.

We assumed that messages are delivered in causal order (Rule V), hence $<\mathsf{deliver.increment.scion}(\mathsf{Bx}^{j+1}, \mathsf{Dz})>_{\mathsf{k}}$ precedes $<\mathsf{deliver.decrement.scion}(\mathsf{Bx}^j, \mathsf{Dz})>_{\mathsf{k}}$. Thus, at any moment, there exists at least a scion protecting z: either $\mathsf{scion}(\mathsf{Bx}^j, \mathsf{Dz})$, or $\mathsf{scion}(\mathsf{Bx}^{j+1}, \mathsf{Dz})$, or both. Therefore, the lemma remains true when a bunch tracing occurs at process p^w.

[8] Note that the propagation of x^1 is not relevant because it does not lead to the sending of any GC specific message, in particular no decrement.scion is performed.

Now, we consider a propagate operation. Before a $<\mathsf{send.propagate}(\mathsf{x}^{j+1})>_{\mathsf{p}^w}$ takes place, according to Rule IV (Clean Propagation), x^{j+1} is GC-cleaned. Therefore x^{j+1} is scanned, $\mathsf{stub}(\mathsf{Bx}^{j+1}, \mathsf{Dz})$ is created and $<\mathsf{send.increment.scion}(\mathsf{Bx}^{j+1}, \mathsf{Dz})>_{\mathsf{p}^w}$ is performed. Therefore, the lemma remains true as x^{j+1} is GC-clean and $\mathsf{scion}(\mathsf{Bx}^{j+1}, \mathsf{Dz})$ exists.

This terminates the induction over the size of the set containing granules pointing to z, and over the size of the set containing processes caching replicas of the granules pointing to z. □

Now, we prove the following theorem:

Theorem 1 *Let n different granules $\mathsf{x}^1, \ldots, \mathsf{x}^n$ be the set of all protected granules pointing to $\mathsf{z} \in \mathsf{D}$ owned by process k, located respectively in bunches $\mathsf{B}^1, \ldots, \mathsf{B}^n$. Let m different processes $\mathsf{p}^1, \ldots, \mathsf{p}^m$ be the set of all processes caching a replica of all bunches mentioned above. Then, $\forall i, 1 \leq i \leq n$, $\mathsf{scion}(\mathsf{B}^i\mathsf{x}^i, \mathsf{Dz})$ exists at k.*

This theorem implies Safety Invariant 2. Whereas the latter states that, whatever number of cross-bunch pointers point to $\mathsf{z} \in \mathsf{D}$, at least one scion protects z, Theorem 1 is stronger, saying that at least one scion per bunch containing a pointer to z, protects z. Note that it does not say whether this scion effectively corresponds to an existing pointer to z.

We prove this theorem by induction: *(i)* over the size of the set containing granules pointing to z in a single process, and *(ii)* over the size of the set of processes caching replicas of granules pointing to z.

Base case:
Assume that initially $\mathsf{x}^1 \in \mathsf{B}^1$ and $\mathsf{x}^2 \in \mathsf{B}^2$ are both owned by p^1, only x^1 points to $\mathsf{z} \in \mathsf{D}$ owned by k, $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ exists, and process p^2 does not cache any granule pointing to z. This initial situation obviously verifies the theorem as $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ exists.

Now, consider that $<\mathsf{x}^2 := \mathsf{x}^1>_{\mathsf{p}^1}$ is performed (thus, x^2 also points to z); therefore x^2 is GC-dirty. If x^1 is not assigned to thereafter, the theorem remains trivially true, since $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$ exists.

Therefore, consider an assignment such as $<\mathsf{x}^1 := 0>_{\mathsf{p}^1}$. (The actual value of the right-hand side does not matter for the proof, except that when the right-hand side is a pointer to z it is as if the assignment did not take place.) Until a bunch tracing or a propagate operation takes place at p^1, no $<\mathsf{send.increment.scion}>$ or $<\mathsf{send.decrement.scion}>$ is performed, and the theorem remains trivially true.

Now, suppose a bunch tracing does occur at p^1; by Lemma 1 we have that at any moment, there exists at least a scion protecting z: either $\mathsf{scion}(\mathsf{B}^1\mathsf{x}^1, \mathsf{Dz})$, or $\mathsf{scion}(\mathsf{B}^2\mathsf{x}^2, \mathsf{Dz})$, or both.

Suppose a $\mathsf{propagate}(\mathsf{x}^1)$ occurs at p^1; by Lemma 2 we have that $\mathsf{scion}(\mathsf{B}^2\mathsf{x}^2, \mathsf{Dz})$ exists before $<\mathsf{send.propagate}(\mathsf{x}^2)>_{\mathsf{p}^1}$.

Induction case:
Let j different granules $\mathsf{x}^1, \ldots, \mathsf{x}^j$, located in bunches $\mathsf{B}^1, \ldots, \mathsf{B}^j$, $1 \leq j \leq n$, all pointing to z, all owned by $\mathsf{p}^h, 1 \leq h \leq m$. Initially, $\mathsf{x}^{j+1} \in \mathsf{B}^{j+1}$, owned by p^h does not point to z, $\mathsf{scion}(\mathsf{B}^j\mathsf{x}^j, \mathsf{Dz})$ exists, and process p^{h+1} does not cache

any granule pointing to z. This initial situation obviously verifies the theorem as scion($B^j x^j$, Dz) exists.

Now, consider that $<x^{j+1} := x^j>_{p^h}$ is performed (thus x^{j+1} also points to z), therefore x^{j+1} is GC-dirty. If x^j is not assigned to thereafter, the theorem remains trivially true as scion($B^j x^j$, Dz) exists.

Therefore, consider an assignment such as $<x^j := 0>_{p^h}$ (again, the actual value of the right-hand side does not matter for the proof). Until a bunch tracing or a propagate operation takes place at p^h, no <send.increment.scion> or <send.decrement.scion> is performed at p^h, and the theorem remains trivially true (scion($B^j x^j$, Dz) still exists).

In the first case, *i.e.* when a bunch tracing takes place at p^h, by Lemma 1 we have that at any moment, there exists at least a scion protecting z: either scion($B^j x^j$, Dz), or scion($B^{j+1} x^{j+1}$, Dz), or both.

In the second case, *i.e.* when event $<$send.propagate$(x^{j+1})>_{p^h}$ happens, by Lemma 2 we have that scion($B^{j+1} x^{j+1}$, Dz) exists. □

A.2 Liveness

Our distributed GC algorithm is clearly not complete because it does not reclaim all cross-bunch cycles of garbage. Thus, we only consider the existence of acyclic cross-bunch garbage.

We assume that: *(i)* every bunch is eventually traced, *(ii)* intra-bunch tracing is complete w.r.t. that bunch, and *(iii)* increment.scion, decrement.scion, and union messages are eventually delivered in causal order.

We will not present here the full proof of liveness given its lack of interest. We simply show the conditions that must hold and how they are ensured.

The obvious liveness condition is:

Liveness Condition 1 *A granule not reachable is eventually reclaimed.*

Given that we are considering cross-bunch collection, Liveness Condition 1 implies:

Liveness Condition 2 *A granule not protected by any scion is eventually reclaimed.*

This condition is obviously ensured by the assumptions that every bunch is eventually traced, and bunch tracing is complete w.r.t. that bunch.

Note that, for liveness we must ensure that if a granule is no longer reachable from any incoming cross-bunch pointer, eventually no scion protects it. Thus, the following liveness condition must hold:

Liveness Condition 3 *A granule no longer reachable is eventually not protected.*

This condition is obviously ensured because: *(i)* we assumed that every bunch is eventually traced and bunch tracing is complete w.r.t. that bunch, therefore eventually there will be no stubs for disappearing outgoing pointers, and *(ii)*

we assumed that both messages for deletion of scions and union messages are eventually delivered.

Therefore, a scion representing a incoming cross-bunch pointer no longer existing will be eventually deleted, and the corresponding object reclaimed.

This article was processed using the LaTeX macro package with LLNCS style