Larchant-RDOSS: a Distributed Shared Persistent Memory and its Garbage Collector*

Marc Shapiro** and Paulo Ferreira***

INRIA Rocquencourt
email: shapiro@sor.inria.fr, Web: http://prof.inria.fr/

Abstract. Larchant-RDOSS is a distributed shared memory that persists on reliable storage across process lifetimes. Memory management is automatic: caching of data and of locks, coherence, collecting objects unreachable from the persistent root, writing reachable objects to disk, and reducing store fragmentation. Memory management is based on a novel garbage collection algorithm, that (i) approximates a global trace by a series of partial traces within dynamically determined subsets of the memory, (ii) causes no extra I/O or locking traffic, and (iii) needs no extra synchronization between the collector and the application processes. This results in a simple programming model, and expected minimal added application latency. The algorithm is designed for the most unfavorable environment (uncontrolled programming language, reference by pointers, non-coherent shared memory) and should work well also in more favorable settings.

1 Introduction

The Reliable Distributed Object Storage System (Larchant-RDOSS) is an execution environment based on the abstraction of a persistent distributed shared memory. Applications see a single memory (a single heap), containing dynamically-allocated data structures (called objects) connected by ordinary pointers. Internally, the memory is divided into granules called *bunches.* An application maps only those bunches that it is currently reading or updating; updates remain local until the application commits.

The system automatically determines which objects are actually shareable by other applications (any object reachable from a persistent object is itself persistent; this is called "persistence by reachability" [3]), by tracing from a persistent root. Such automatic management results in a simple and natural programming model, because the application programmer need not worry about distribution, input-output or deallocation.

The intended application area is programs sharing a large amount (many gigabytes) of objects on a wide-area network, *e.g.,* across the Internet. Examples include financial databases, design databases, group work applications, or exploratory applications similar to the World-Wide Web.

* Presented at WDAG'95, 9th International Workshop on Distributed Algorithms, Le Mont Saint Michel (France), September 1995.

** Work conducted in part during sabbatical year at Cornell University, supported by funding from Cornell University, Isis Distributed Systems Inc., INRIA, and NATO.

*** Student at Université Pierre et Marie Curie, Paris; with support from a JNICT Fellowship of Program *Ciência* (Portugal)

1.1 Larchant-RDOSS

This setting imposes performance constraints, such as avoiding I/O and synchronization. Furthermore it is not reasonable to expect any strong coherence guarantees.

These performance constraints appear to clash with persistence by reachability. Distributed tracing requires global synchronization. Accessing remote or on-disk portions of the object graph requires costly input-output and network communication. Published concurrent GC typically assume a coherent memory, and require a strong, non-portable synchronization between the application (the "mutator") and the collector.

Our algorithm works around these difficulties. Instead of a global trace, we perform a series of opportunistic local traces that together approximate the global trace. Each local trace scans a dynamically-chosen subset of the memory; each site chooses its subset independently from other sites, in such a way that the trace requires no remote synchronization and no I/O. We avoid mutator-collector synchronization, by relying on the trace itself to discover new pointers. We avoid relying on any particular coherence model by delaying the delete of an object until it has been detected unreachable everywhere. We do however assume that a granule has no more than a single owner at any point in time. For performance we use asynchronous messages, relying on causally-ordered delivery for correctness.

1.2 Outline

This paper briefly describes the overall design of Larchant-RDOSS, in Section 2. Section 3 outlines our garbage collection algorithm, ignoring replicated caching; it can collect an arbitrary subset of the object graph independently; we suggest a locality-based heuristic for choosing the subset that avoids message, I/O and lock traffic. In Section 4, we extend the algorithm to the case where a bunch is replicated into multiple caches, even when the replicas are not known to be coherent. Section 5 presents related work. Section 6 concludes with a summary of our ideas and results.

2 Overall Design

The architecture of Larchant-RDOSS is illustrated in Figure 1. An application program on some site access the shared memory through that site's Cache Server. Stable versions of bunches are stored on disk by Backup Servers. Together, the Cache Servers and the Backup Servers form the Object Storage Service. Auxiliary Collector Processes perform the garbage collection algorithm on behalf of a Cache or Backup Server. This section describes the applications and the cache and backup servers; garbage collection will be described in more detail in Sections 3 and 4.

2.1 Bunches and Pointers

The distributed shared memory is subdivided into bunches. A bunch is a set of memory segments, containing objects. An application holding a pointer to an object may map the bunch containing it (thus gaining access to the object's data) through the Cache Server of its site. The bunch is the elementary granule of mapping, of locking, of coherence and of garbage collection.

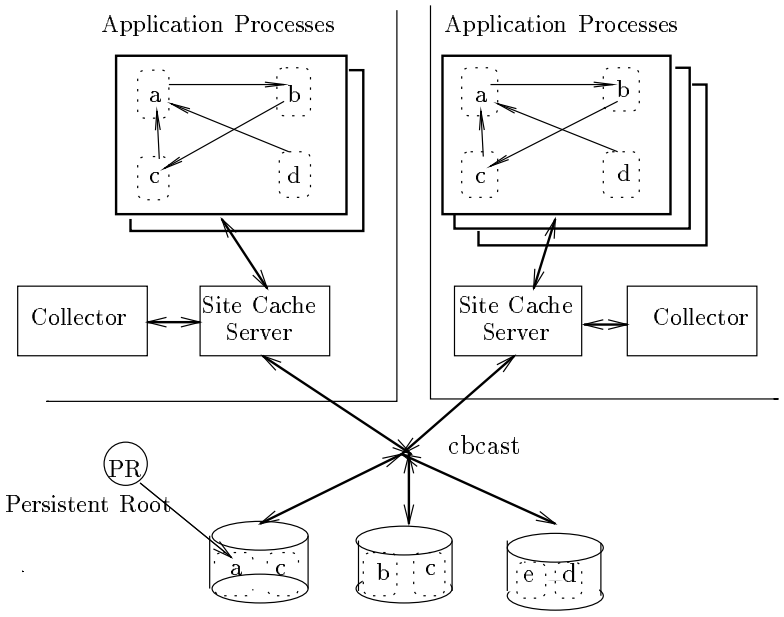

Fig. 1. *General architecture of Larchant-RDOSS*

Multiple applications at multiple sites may map the same bunch, which is said *replicated* into the corresponding site caches. The same bunch may be backed up to multiple disks for availability; it is said replicated on disk.

A process in Larchant-RDOSS starts up with an initial bunch containing a persistent root object, itself containing pointers to other objects. The application follows some arbitrary path of pointers through the object graph. When following a pointer for the first time, the application binds it. The bind primitive ensures that the target object is completely initialized, and sets a lock. Supported lock modes include the both standard consistent locks (read and write), and weak locks (optimistic, no-guarantee).

Binding ensures that, if the target pointer contains a pointer itself, the latter is valid, by reserve-ing a memory location for its own target. The pointer is "swizzled" but the target is not actually bound until it is accessed, as in Texas [22].[4] The primitive reserve sets a special lock ensuring that the address of the corresponding objects will not change.

Allocating a new object in a bunch declares its *layout* to the system, *i.e.*, the location of any pointers it contains. The application freely reads and updates a bound object, including the pointers it contains. This shared memory abstraction is natural, and can be used even from primitive languages such as C or C++.

Updates remain private until the application process commits, at which point all updates are propagated at once. Locking may be pessimistic or optimistic; optimistic locking is useful and efficient in many types of applications; it turns out that the garbage collector uses it also.

[4] On storage and in a cache manager, a reference has a location-independent representation. In memory, a reference is a pointer to the memory location of the target. The location of a same object may differ between in different application processes. Swizzling a reference means to compute the address of its target and to fill the corresponding pointer with that address. The application code only observes swizzled pointers and can dereference them with no overhead.

We are not assuming the existence of a garbage collector at the application process. The collector described here collects the persistent store, and executes in the Collector Processes.

2.2 Object Storage Service

The Object Storage Service (OSS) is composed of Cache Servers and Backup Servers.

A Backup Server (BS) caches recently-accessed bunches in memory, and stores them on disk. Application and Collector processes access the store through the single Cache Server (CS) running on the local machine. A CS caches bunches recently accessed by local application processes. It caches both bunch data and the associated lock tokens. A cache miss causes a bunch to be copied from disk into the corresponding BS cache, sent to a CS and copied into its cache, and from there copied into the requesting application or collector. When an application commits, it propagates any changed bunches to its cache server, which multicasts the changes to all other copies.

At some point in time, any single bunch can be replicated in any number of Cache Servers, of Backup Servers, of disks, of application processes, and of Collector processes. The server that, either holds the exclusive (write) token for a bunch, or was the last to hold it, is the *owner* of that bunch. An update may commit updates only at the owner site. Updates flow from an application or a collector process, to the local (owner) CS, which propagates it to the other servers. A BS stores updates on disk. Since an update or a token flows only from the owner to other processes, it can be sent using causal broadcast [5].

2.3 Data Structures

The contents of a bunch is described by some special data structures. These contain all the information needed for garbage collection and swizzling.

An *Object Map* describes the location and *class* of objects inside the bunch. An *In-List* indicates which of these objects might be pointed at from another bunches. Each in-list element, called a *scion*, identifies a different potential {source bunch, target object} pair. A special form of scion indicates a persistent root.

An object's class describes its layout and type. A class is an object itself, named by a pointer. The layout gives the location and type of pointers inside objects of that class.

A *Reference Map* indicates the location and type of pointers inside the bunch. Each pointer is described in location-independent form, *i.e.,* the Reference Map identifies the target bunch, and object within that bunch, of each pointer. An *Out-List* indicates which of those references cross out of the bunch's boundaries; elements of the out-list are called *stubs*.

All the above data structures are normally stored within the bunch itself, making the bunch a self-contained unit of I/O, storage, and collection. The exception is a class, which is identified by a pointer, and hence may be stored in another bunch. Classes and types are not used by the language-independent layers of RDOSS, and will be ignored in the remainder of this document.

To allocate an object, a program calls a typed version of malloc which allocates both the memory for the new object, and the corresponding map entries. Layout information is extracted automatically from the binary files by our programming environment. Some unsafe contructs are forbidden (specifically, the union of a pointer

and a non-pointer, and casts from non-pointer to pointer) but otherwise the language and the compiler remain unchanged.

3 Garbage Collection of the Persistent Shared Memory

We now focus on the Garbage Collection of the persistent shared memory.

3.1 Requirements and Limitations of Existing Algorithms

A garbage collection algorithm must be correct and have good performance. It is correct if it is both safe and live, *i.e.,* if it never collects a reachable object, and never deadlocks or livelocks.

Ideally, it would also be *complete, i.e.,* would eventually collect all garbage. However, completeness is at odds with performance and scalability. To be complete, a GC would have to globally trace the whole graph of objects reachable from the persistent root set (called *live* objects). But, in a large-scale persistent shared memory, this is not feasible, for a number of reasons.

First, all known tracing algorithms require a global synchronization, which is not realistic in a large-scale system.

Second, at any point in time, the major part of the object graph is swapped out to disk, and cannot be accessed economically.

Third, object copying and pointer patching appear to require write-locks that compete with the applications'.

Fourth, existing concurrent GC algorithms require instrumenting *every* mutator pointer instruction with a "barrier" in order to inform the collector of the value read or written. This slows down the applications considerably and is not portable.

3.2 Main Ideas of our Algorithm

Apparently, the problem is hopeless. But we will now show a solution that gives an excellent approximation of the global concurrent trace and does not have the same drawbacks.

Instead of a global synchronized trace, of the whole object graph on the whole network, it approximates the same result with a series of replicated, opportunistic, non-synchronized, piecewise, local traces. Each bunch is traced at each site where it is cached, and the results summarized at the bunch owner (this will be explained in Section 4.2). Groups of multiple bunches mapped at some site are scanned at once, thus collecting cycles of garbage that span bunches.

In order to not compete with the application, the GC works on a separate data set; namely, any datum write-locked by an application is ignored by the collector (*i.e.,* is conservatively considered live). To avoid input/output, the collector also ignores data that is swapped to disk.

The collector does not cause any lock traffic, because it locks an object only at a cache server where the lock is already cached. Its locks do not compete with application locks, because the collector runs as an optimistic transaction.

The collector makes no assumptions about the mutual consistency of the many replicas of some bunch, apart from the assumption that only the owner of a bunch

can commit a write into that bunch. Thus, data can be incoherent. The GC compensates for incoherence by being more conservative. Any object that has been reachable continues to be considered reachable as long as the collector does not positively determine, at all copies and independently of the coherence protocol, that it is not.

There is no barrier; the algorithm discovers pointer assignments the next scan of the collector itself, effectively batching pointer assignments.

For clarity, we will explain the algorithm in three steps: (i) collecting a single replica of a single bunch on a single site, (ii) collecting a group of bunches on a single site, (iii) collecting the multiple, possibly incoherent, replicas of a single bunch at all its sites. We will show each step and argue their correctness, while also justifying the design decisions listed above. The first two steps are quite simple and will be detailed next. The last one will be explained in Section 4.

Our algorithm is a hybrid of tracing and counting: (i) It *traces* bunches within groups limited by what is economically feasible. Specifically, the trace stops when it would require input-output or network or lock traffic. (ii) The algorithm uses *reference counting* (via the scions, at the group boundary) when tracing would be too expensive. (iii) A tracing group changes dynamically, seamlessly, and at each site independently from other sites.

The group-formation heuristics must trade off performance and scalability on the one hand against completeness; our current heuristics favor the former.

Our algorithm (following the example of some centralized GC algorithms, such as reference counting [21] and conservative GC [8]) is not provably complete. The trade-off between completeness and performance can be tuned by the choice of the grouping heuristic. Our current heuristic, based on locality (see Section 3.5) has a good chance of collecting cycles of garbage in all real-life situations.

3.3 Scanning a Single Replica at a Single Site

A particular replica of a bunch can be scanned on its site, independently of other bunches and independently of remote versions of the same bunch.

The in-list presented in Section 2.3 identifies all the pointers that reach into a bunch. Ignoring for a short while concurrent updates, it is safe to scan, by considering the in-list as its root set. This is illustrated in Figure 2. Such a collection is complete with respect to the bunch, *i.e.,* it will deallocate any cycle of garbage that is entirely within the bunch. However it is conservative with respect to other bunches, since it can not deallocate a cycle of garbage that crosses the bunch boundary; thus, the in-list serves as a reference counter for inter-bunch references.

Here is how a trace proceeds. Any object pointed from the in-list is marked reachable. An object inside the bunch, pointed from a reachable object, is itself marked reachable. If a reachable object points outside the bunch, a corresponding stub is allocated in the out-list. The result of the walk is a reachable-set and a new out-list. Any objects not in the reachable-set can be deallocated locally; the bunch owner may safely reallocate a deallocated object.

The new out-list is compared with the one resulting from the previous scan. Stubs that didn't previously exist indicate that a new inter-bunch reference has been created; the collector sends a `create` message to the (owner of the) target bunch so that it can create the corresponding scion.

Pointer updates are not noticed until the bunch is scanned (in contrast to concurrent garbage collectors where the mutator must immediately inform the collector

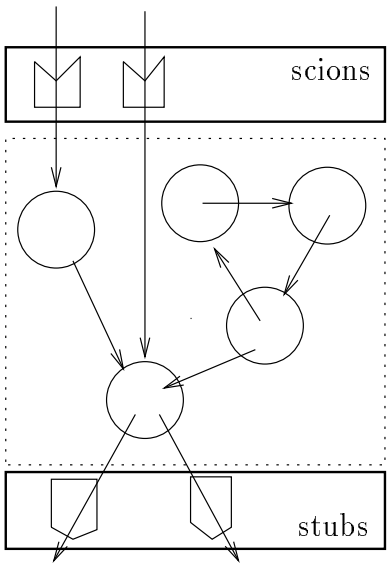

Fig. 2. *Collecting a single replica of a single bunch*

of pointer updates, by using a read or write barrier), and in an arbitrary order. To ensure safety, all create messages are sent before any deletes.

To see why this is important, consider a pointer x pointing to an object A, and a pointer y; the program assigns the value of x to y, then modifies x, *e.g.,*

$$\mathsf{y} := \mathsf{x};\ \mathsf{x} := \mathsf{NULL}$$

The scan could discover the second assignment earlier than the first. Ordering the create before the delete ensures that the target A is not deallocated prematurely, even if x contained the last pointer to A. We will see in Section 4 that deletes may be further delayed in the case of multiple copies.

A create message is sent asynchronously. Since a new pointer value can only, either point to a locally-created object, or be a copy of an existing, reachable pointer, it follows that the target object will not be collected prematurely.

Stubs that have disappeared since the latest scan indicate that an inter-bunch reference no longer exists. A delete message is sent, asynchronously, in order to remove the corresponding scion in the target bunch. To avoid race conditions, no delete message is sent until all the create messages have been. Furthermore, in the case of a replicated bunch, a delete message may need to be delayed even further (see Section 4).

The collector runs as an optimistic transaction: if the collector has started scanning a bunch, and an application later takes a write lock and modifies it, then the collector aborts and its effects are undone. (A liveness assumption is that the GC transaction eventually commits.) Thus, the collector does not compete for data locks with the application, but it is still safe to ignore concurrent mutator updates. The collector is trivially safe and live; it is complete with respect to cycles of garbage enclosed within the bunch, but conservative with respect to possible garbage referenced from another bunch.

When an object has been moved, any pointers that reference it must be patched. This would entail finding the bunches containing these pointers, and taking a write

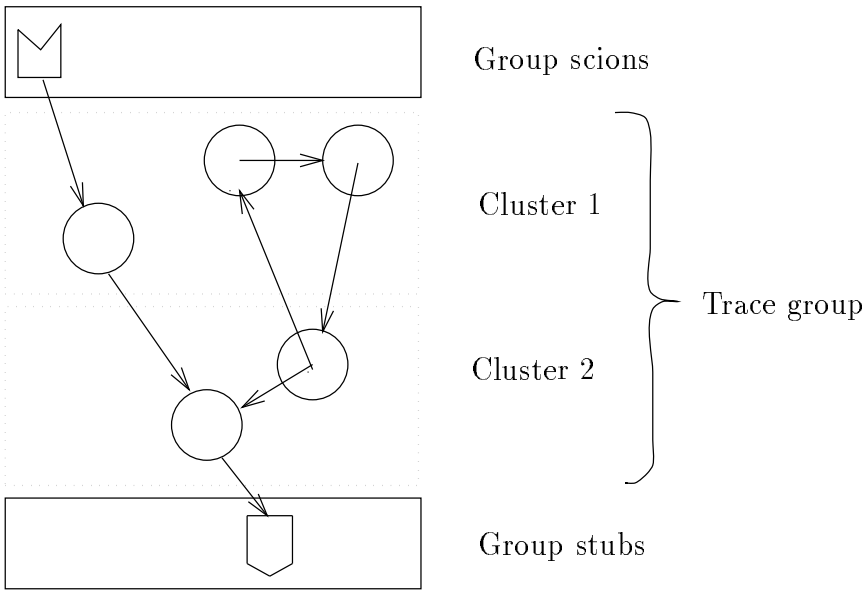

Fig. 3. *Collecting a group of bunches at a single site*

lock on them. In fact, pointer patching can be delayed until such source bunches are next mapped and swizzled.

Consistency is not a problem because the collector will move an object only if the source bunch is not protected by a reserve lock.

3.4 Group Scanning of Multiple Bunches at a Single Site

Just as the collector can scan a single bunch replica at a single site, it can scan any group of bunches at a single site. The algorithm is exactly the same as above, except that scions for pointers internal to the group are not considered roots (this is easy to check because a scion identifies its predecessor bunch), and that scanning continues across bunch boundaries, as long as the group is not exited. Figure 3 shows an example group of two bunches.

A group will contain only bunches that are not write-locked by an application. (A liveness assumption is that no bunch will remain locked indefinitely.) As above, a collection aborts if the application modifies a bunch that has already been scanned. This ensures that the collector does not compete with the application for

locks. For the same reasons as the single-group case, the algorithm is trivially correct. Group collection is complete with respect to bunches in the group, *i.e.,* a cycle of garbage, possibly crossing bunch boundaries, but remaining within the group, will be collected. It is conservative with respect to bunches not in the group.

3.5 Group Heuristics

The significance of group scanning is that *any arbitrary subset of the persistent memory can be scanned,* on a single site, independently of the rest of the memory. The choice of a group can only be heuristic, and should maximize the amount of garbage collected while minimizing the cost.

We will use the locality-based heuristics of a group of all the bunches that are cached on the site at the time the collector happens to run, except those currently being written by an application. This heuristics avoids all input-output costs, and minimizes aborts. Furthermore there is no lock traffic cost since any locks are taken only if cached locally.

This policy favors performance and scalability over completeness. We believe it has a high probability of collecting garbage in all real-life situations, for the following two reasons. First, empirical results from centralized GCs show that cycles are rare and usually small. Second, our intuition is that since a bunch swapped to disk has not been accessed recently, it has a low probability of containing new garbage. However at this point, we do not yet have experimental results confirming this intuition.

We are aware of the limitations of our current policy. For instance, it does not collect cycles of garbage that reside partially on disk or on another server. Collecting such a cycle involves I/O costs, which must be balanced against the expected gain. An I/O is very costly, but in the general case there is no way of knowing how much garbage a bunch contains short of loading it into memory and scanning it.[5]

We are implementing the locality-based heuristic as a first shot. If experimental results show that an excessive amount of garbage is retained, we will explore others. To improve completeness it is necessary to cache more aggressively, for instance by following pointers from already-cached bunches, or by loading random bunches into a cache. Such an aggressive policy might compete in the cache with the applications' working set.

4 Collecting the Multiple Copies of a Cached Bunch

4.1 Garbage Collection and Incoherent Replicas

The simple technique of avoiding concurrent mutator updates, by running the collector as an optimistic transaction, does not work well if replicas of a bunch are present at multiple storage servers.

Since we do not assume coherent copies, the collector could observe some pointer value on one site, while the mutator has assigned a different value on another site. Let us illustrate this problem with an example (see Figure 4). Imagine that pointer x in bunch C1 points to object A in bunch C3. An application program at site S1 assigns NULL to x. Then the collector of site S1 runs. Site S1 is the owner of C1.

Concurrently, another application program assigns pointer y (within bunch C2) with the value of x, as such: y := x. Then the collector runs at sites S2 and S3 (the owner sites of C2 and C3, respectively). To make the example interesting, we suppose that the assignment to y is ordered before the assignment to x, *i.e.,* y now points to object A.

When the collector runs at each site, site S1 sends a delete (C1, A) message to S3, and site S2 sends create (C2, A), also to S3. However, in an asynchronous system, a message can be delayed indefinitely, so the delete could be received before the create, with catastrophic results. (We will call this a "fast delete message".) Furthermore,

[5] This is in contrast to the Sprite Log File System [19], where the semantics allow the collector to quantify the amount of garbage in a "segment" (their equivalent of a bunch) without actually reading it. Note also that Cook *et al.* [9] propose some heuristics for estimating the relative amount of garbage in partitions (*i.e.,* in bunches) without reading them; this work does not take their ideas into account.

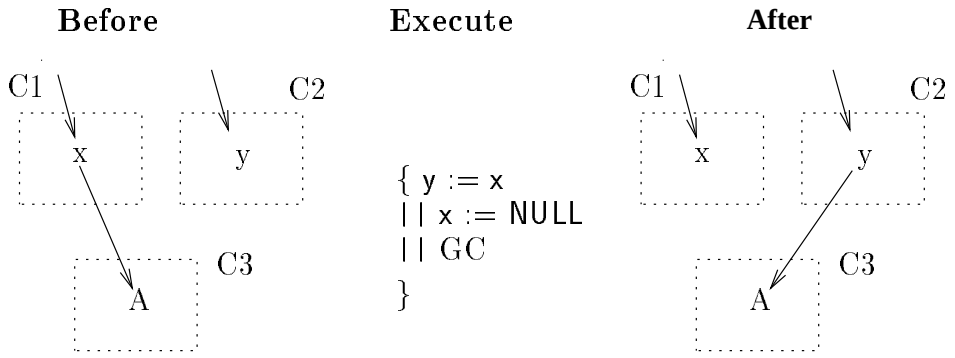

Fig. 4. *Possible race conditions with concurrent read-write-scans of a replicated bunch.*

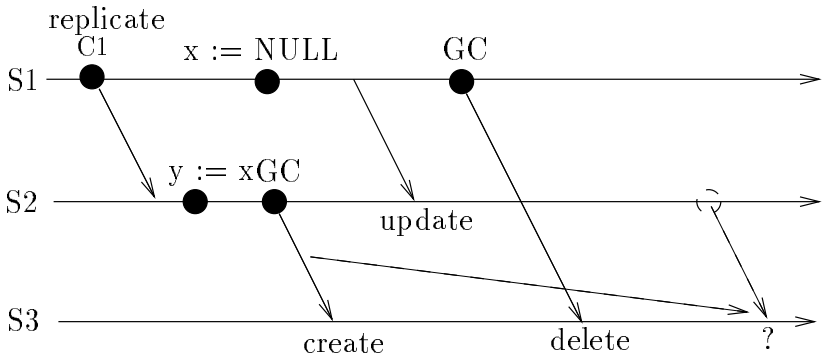

Fig. 5. *Timeline for the execution of Figure 4, showing the effect of a late create or a slow create message.*

since the collectors run at unpredictable times, the delete could actually be sent before the create, with equally catastrophic results. (This will be called a "early delete".) These conditions are illustrated by Figure 5.

4.2 An Asynchronous Solution to the Problem

One possible solution would be to impose coherence between mutators and add synchronization between mutators and collectors; this is essentially what is found in existing concurrent GCs. This solution is undesirable for performance reasons. Instead, we propose an asynchronous solution that avoids both early deletes and fast delete messages. We will look at these two elements in turn; our solution, the "Union of Partial Out-Lists" (UPOL) is illustrated by Figures 6 and 7.

Avoiding Early Deletes We delay the sending of a delete message until all logically-preceding creates have been sent. To do this: (i) we delay sending deletes until the corresponding update has been applied at all the copies of the bunch, and (ii) we force any creates from some site to be sent before applying any update at that site.

To get (ii), we scan any modified bunches before accepting updates on the same bunch (this may cause an extra delay at commit time).

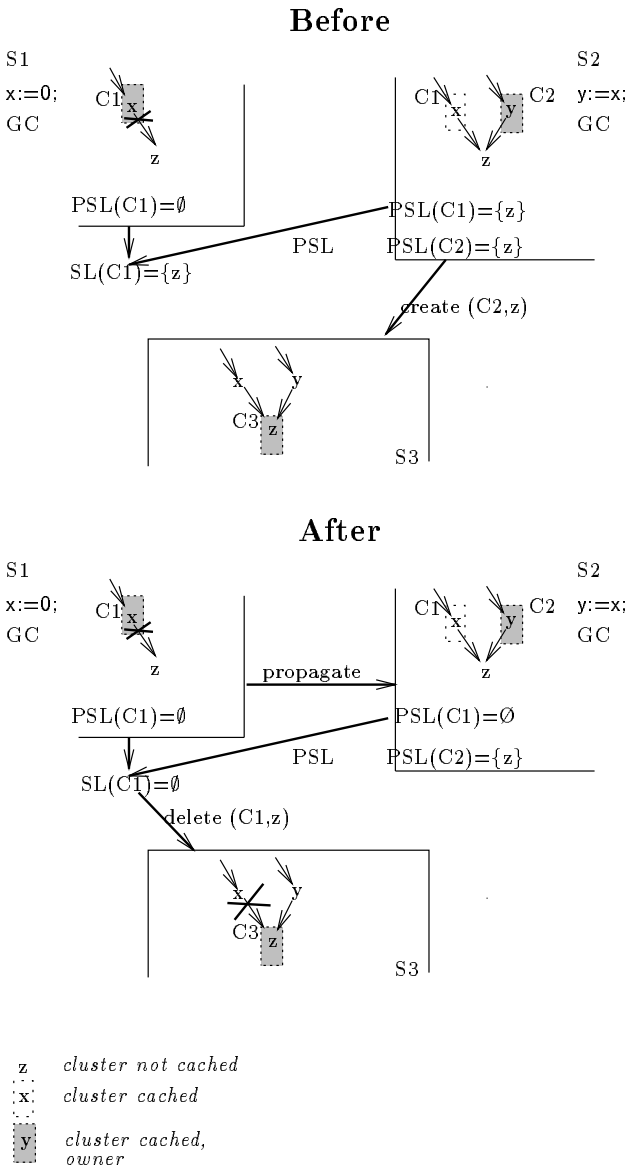

Fig. 6. *Union of Partial Out-Lists solution, before and after application of a pointer update.*

Property (i) could be achieved by getting an acknowlegment from the coherence layer; but this is not necessary, since the necessary information already is available from the collector. We stated earlier that each copy of a bunch is collected at its site, and the results are summarized at the bunch owner. We can now explain precisely what that means.

Each bunch copy is collected according to the algorithm in Section 3.3 or Section 3.4, creating a new out-list, the *Partial Out-List* (POL) for that copy. It is partial because it only lists the stubs reachable at that site. After the collection, each Partial Out-List is sent to the owner of the bunch in a POL message.

The owner collects all Partial Out-Lists; the complete out-list for the bunch is just the union of the most recent Partial Out-List of each copy. The owner sends

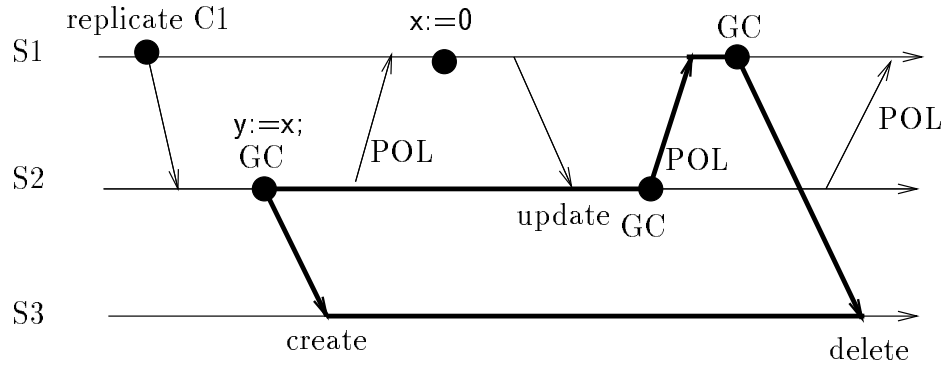

Fig. 7. *Timeline for the Union of Partial Out-Lists solution, illustrating asynchronous messages and causal ordering.*

a delete message only when a stub disappears from the complete out-list, and a non-owner never sends a delete.

This works because of three properties of stubs: (i) a reachable stub can become unreachable at any site; (ii) a stub that is unreachable at all sites will never become reachable; and (iii) only the owner of a bunch can make a new stub appear in that bunch.

Property (i) is a consequence of the transitivity of the reachability property. For instance, suppose that the variable x in the previous example is reachable from the persistent root only through pointer z located in another bunch, say C4. The application running at the owner of C4 can modify z, making x unreachable, hence the stub from x to A is also unreachable, even though the bunch C1 containing x has not been modified at that site.

Property (ii) is by the stability property of garbage.

Property (iii) is because we assume that only the owner of a bunch can write into that bunch.

Because of these properties, and assuming (possibly unreliable) FIFO communication, it is safe for the owner consider only the most recently-received version of each site's Partial Out-List. It doesn't matter how old it is: it can only err on the side of conservativeness, *i.e.,* of considering as live an object that is not reachable any more.

A limitation of the UPOL algorithm is that collection is sensitive to the way cycles of garbage are replicated. Suppose a cycle of garbage involves objects located in bunches A and B. If A and B are both cached at site 1, the cycle can be collected locally. If A and B are both replicated at sites 1 and 2, both sites can also collect the cycle. But if A is replicated at 1 and 2, and B is only cached at 1, then site 2 cannot collect the cycle, and by the UPOL algorithm it cannot be collected at all. If this turns out to be a problem in practice, a possible approach would be to bias the caching policy to avoid the above situation.

Avoiding Fast Delete A careful examination of Figure 7 shows that the POL message creates a causal dependency between the create message and the delete message (the thick arrows in the figure). Any of the well-known techniques for causal delivery of messages [4] will therefore ensure that the create message will not be overcome by the corresponding delete. Larchant-RDOSS sends the create, POL, and delete messages using the Isis cbcast primitive.

5 Related Work

Multiprocessor and concurrent GC algorithms [2, 10] are not directly applicable, because they typically are based on strong consistency and scale assumptions. Even Le Sergent and Berthommieu [15] consider a small-scale, strongly-consistent DSM.

Much previous work in distributed garbage collection [6, 20] considers processes communicating by messages (without shared memory), using a hybrid of tracing and counting. Each process traces its internal pointers; references across process boundaries are reference-counted as they are sent in messages. Some object-oriented databases use a similar approach [1, 23, 16].

In order to collect cycles of garbage, Lang, Queinnec and Piquer [14] augment a hybrid algorithm with a scan of dynamically-changing groups of processes. Their groups are remote and entail a complex joining/disbanding protocol and complex synchronization inside a group. Our grouping algorithm is a simplification of the Lang-Queinnec-Piquer proposal since our groups are local. Furthermore our groups are not determined *a priori* but opportunistically.

The concept of PBR was first proposed by Atkinson and Morrison [3, 17] in the early 1980's. PBR-related collection is considered for instance in O'Toole and Nettles [18], where the collector scans a possibly old copy of the data. We have found two distributed shared memories with PBR in the literature. The specification of the Casper collector [13] is sketchy and seems incapable of collecting a persistent object that has become garbage. EOS [12] has a tracing and copying GC that takes into account user placement hints to improve locality. However, their GC is quite complex and has not been implemented.

Larchant-RDOSS is a simplified version of our own work on Larchant-BMX [11]. Whereas in Larchant-RDOSS the collector may abort in case of conflicts with the mutator, the Larchant-BMX collector is fully concurrent. It uses the O'Toole and Nettles algorithm. RDOSS allows non-coherent memory whereas Larchant-BMX is based upon entry consistency, and therefore does not need the UPOL technique. In Larchant-BMX the granule of consistency and locking is different from the GC granule: the former is the object and the latter is the bunch.

Many hybrid GCs have been published (*e.g.,* Bishop [7]). Ours has the following novel features. The tracing is replicated. Collectors are unsynchronized. Replicas need not be consistent. The counting boundaries change dynamically and are different at each site. Our collector discovers changed pointers at collection time and orders safely the corresponding events. The collector runs as an optimistic transaction.

6 Conclusion

The problem of tracing a large-scale shared distributed store seems intractable at first glance. We have shown an algorithm that gives an approximation of the global trace, with none of the drawbacks. This algorithm causes no input-output nor lock traffic. It opportunistically scans groups of bunches, according to a locality-based heuristics. The algorithm is independent of any particular coherence management (it does not assume coherent memory) but does assume a single owner per bunch. There is no coordination or synchronization between the application programs (mutators) and the collector. It works even with primitive programming languages, with no language or compiler changes (but small programming restrictions are necessary).

Collector messages are asynchronous but require a causally-ordered communication layer.

We explained our algorithm in the context of a shared persistent virtual memory containing ordinary memory pointers. Since this is the worst-case scenario, the same algorithm should be applicable to many other cases, such as persistent object stores and shared-memory multiprocessors.

Larchant-RDOSS is currently being implemented, as well as a similar (but slightly different) design, Larchant-BMX [11]. When stable, the code will be made publicly available.

References

1. Laurent Amsaleg, Michael Franklin, and Olivier Gruber. Efficient incremental garbage collection for workstation/server database systems. Rapport de Recherche RR-2409, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1994.
2. Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proc. Prog. Lang. Design and Implementation*, pages 11–20, 1988.
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. Özalp Babaoğlu and Keith Marzullo. *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*, chapter 4, pages 55–93. Addison-Wesley, ACM Press, second edition edition, 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. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, NC (USA), December 1993.
7. Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Mass. (USA), May 1977. Technical report MIT/LCS/TR-178.
8. Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9):807–820, September 1988.
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. 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.
11. 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.
12. Olivier Gruber and Laurent Amsaleg. Object grouping in Eos. In *Proc. Int. Workshop on Distributed Object Management*, pages 184–201, Edmonton (Canada), August 1992.
13. Bett Koch, Tracy Schunke, Alan Dearle, Francis Vaughan, Chris Marlin, Ruth Fazakerley, and Chris Barter. Cache coherency and storage management in a persistent object system. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 99–109, Martha's Vineyard, MA (USA), September 1990.
14. Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, Albuquerque, New Mexico (USA), January 1992.

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. Umesh Maheshwari. Distributed garbage collection in a client-server, transactional, persistent object system. Technical Report MIT/LCS/TM-574, Mass. Inst. of Technology, Lab. for Comp. Sc., Cambridge, MA (USA), October 1993.
17. R. Morrison, M. P. Atkinson, A. L. Brown, and A. Dearle. Bindings in persistent programming languages. *SIGPLAN Notices*, 23(4):27–34, April 1988.
18. James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 161–174, Asheville, NC (USA), December 1993.
19. Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
20. 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), nov 1992. Also available as Broadcast Technical Report #1.
21. 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.
22. Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 364–377, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.
23. V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proc. of the Data Engineering Int. Conf.*, pages 120–133, Houston TX (USA), February 1994.