Larchant-RDOSS: a distributed shared persistent memory and its garbage collector*

Marc Shapiro, Paulo Ferreira
INRIA Rocquencourt
shapiro@sor.inria.fr

INRIA Rapport de Recherche no. 2399
Cornell Computer Science TR94-1466

November 1994

Abstract

Larchant-RDOSS is a distributed shared memory that persists on reliable storage across process lifetimes. Memory management is automatic: including consistent caching of data and of locks, 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 approximates a global trace by a series of local traces, with no induced I/O or locking traffic, and no 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, distributed system, non-coherent shared memory) and should work well also in more favorable settings.

*This work was conducted in part during the first author's sabbatical year at Cornell University, supported by funding from Cornell University, Isis Distributed Systems Inc., INRIA, and NATO.

1 Introduction

1.1 Setting

The Reliable Distributed Object Storage System (Larchant-RDOSS) is an execution environment based on the abstraction of a distributed shared memory. The implementation differs very much from this abstraction.

Applications see a single memory, containing dynamically-allocated data structures (or objects) connected by ordinary pointers. Internally, the memory is divided into granules called *clusters* (of arbitrary size, typically of the order of a few kilobytes). An application maps only those clusters that it is currently reading or updating; updates remain local until the application commits.

Pointers and allocation information are supplemented, internally, by location-independent data structures. From these, the system 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" [1]), by tracing from a persistent root. Such automatic management results in a simple and natural programming model, because the application need not worry about 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.

1.2 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. (Coherence is costly in the large scale, unnecessary for some of the intended applications, and will block in the presence of network partitions and during disconnected operation.) An algorithm that performs well in such a setting can be expected to apply also to a less demanding environment such as a multiprocessor or a well-connected local network.

These performance constraints appear to clash with persistence by reachability. Distributed tracing requires global synchronization.[1] Accessing remote

[1] We deliberately ignore reference counting (RC), for the following reasons: (1) RC is not complete (it does not collect cycles of garbage). (2) The existence of multiple kinds of roots (transient roots and persistent roots) makes RC awkward. (3) We have already studied extensively a wide-area, fault-tolerant version of reference counting [8, 7] and have found it inadequate for a shared-memory environment. As we shall see, our new algorithm combines tracing, whenever economically feasible, with a variant of RC when tracing would be too expensive.

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. A local 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 writer at any point in time. For performance we use asynchronous messages, relying on causally-ordered delivery for correctness.

1.3 Larchant-RDOSS vs. Larchant-BMX

Two slightly different versions of the Larchant architecture are being developed, called respectively Larchant-BMX and Larchant-RDOSS. Both systems implement essentially the same ideas and algorithms but differ in some important aspects.

Larchant-BMX implements an object-granularity entry-consistent [3] single distributed address space abstraction. The collector runs concurrently with the mutator, in the same address space; therefore updates must be reported to the collector. Pointer comparisons must use a special primitive.

Larchant-RDOSS is a simplified version of Larchant-BMX. Larchant-RDOSS provides a more restrictive model, that of single-process transactions, separate address spaces, and causal broadcast of committed writes. A pointer is swizzled (but only at first use); the collector runs independently from the mutator (but only collects the persistent memory).

1.4 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 heuristics for choosing the subset that avoids message, I/O and lock traffic. In Section 4, we extend the algorithm to the case where a cluster is multiply cached, even when the replicas are not known to be coherent. Section 5 concludes with a summary of our ideas and results.

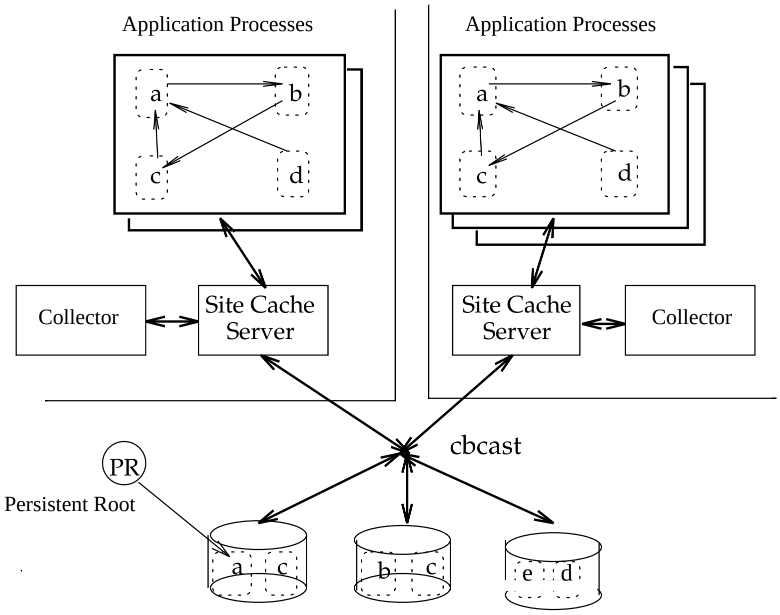

Figure 1: *General architecture of Larchant-RDOSS*

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 clusters 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 Application interface

Applications that use Larchant-RDOSS are ordinary Unix processes. The shared memory is very similar to an ordinary memory-mapped shared region containing pointers. An application accesses clusters through the local Cache Server (see Section 2.3). The API primitives are directed at the local Cache Server.[2]

[2]In our current specification, applications can meaningfully communicate pointers to each other only via the Larchant-RDOSS shared memory. The specification could be extended to sending pointers in messages or through other channels, by integrating the SSPC protocols [7] into Larchant-

2.2 Clusters and pointers

The Larchant-RDOSS primitive startup() provides a process with its initial connection, by maping in an initial cluster containing a persistent root object, itself containing pointers to other objects. The application follows some arbitrary path through the object graph, by following pointers. 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 a pointer up-calls a language-dependent type-checking and unmarshalling module. Binding also 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 latter is not actually bound until it is accessed.) At the same time, the pointer is "swizzled," *i.e.,* a correct value is assigned to it according to the result of the reservation.[3]

Allocating a new object in a cluster 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, as long as the new pointer values are valid, and the updates do not change the object layout.

This shared memory abstraction is natural, and can be used even from primitive languages such as C or C++. Binding does not differ substantially from the familiar Unix mmap primitive, and locking is usual for concurrency control. A call to the allocation primitives malloc or new must provide an extra argument containing the layout of the new object. Programmers must avoid ambiguous unions or casts (where a pointer and a non-pointer could occupy the same location). These rules may be unfamiliar, but do not represent a major language change, and do not require any compiler changes.

Updates remain private until the application process commits, at which point all updates are propagated at once. Locking may be pessimistic (locks are enforced when the application requests them) or optimistic (locks are not enforced until commit time; if a conflict has arised in the meanwhile, the application aborts and its updates are lost.) Optimistic locking is useful and efficient in many types of applications; it turns out that the garbage collector uses it also.

An application pays overhead only at first use of a datum (when binding) and when committing an update (when unbinding). The binding cost is that of getting a lock, mapping in a private copy and swizzling pointers. The unbinding cost is that of constructing the new reference map, unlocking and

RDOSS. This is topic for future work.

[3]This follows the model of Wilson [9]. To simplify the implementation, the first version of RDOSS does not swizzle, and a cluster resides at a fixed address at all client processes.

sending updates. In between, the application runs at full processor speed with no foreground overhead.

Note that we are not assuming the existence of a garbage collector at the application process. One may exist or not. The garbage collector described hereafter only works on the persistent store, and executes in the Collector Processes.

2.3 Object Storage Service

The Object Storage Service (OSS) is composed of Cache Servers and Backup Servers. (A single server process can actually play both rôles.)

A Backup Server (BS) caches recently-accessed clusters 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 clusters recently accessed by local application processes. It caches both cluster data and the associated lock tokens. A cache miss causes a cluster 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 clusters to its cache server, which multicasts the changes to all other copies (different coherence protocols can be plugged in).

At some point in time, any single cluster 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 cluster, or was the last to hold it, is the *owner* of that cluster. 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 Isis "causal broadcast" [4].[4]

2.4 Data structures

The contents of a cluster 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 cluster. An *In-List* indicates which of these objects might be pointed at from another

[4]Communication between the application and collector processes and the storage service, and between the storage servers, uses RDO, a Corba-compliant veneer to Isis, a product of Isis Distributed Systems, Inc. Since application processes only communicate via the shared memory, they need not be aware of RDO nor use Corba interfaces.

clusters. Each in-list element, called a *scion*, identifies a different potential {source cluster, 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. Type information is language-dependent. The API `bind` primitive up-calls a language-dependent type-checking module to type-check a pointer against its target.

A *Reference Map* indicates the location and type of pointers inside the cluster.[5] Each pointer is described in location-independent form, *i.e.,* the Reference Map identifies the target cluster, and object within that cluster, of each pointer. An *Out-List* indicates which of those references cross out of the cluster's boundaries; elements of the out-list are call *stubs*.

All the above data structures are normally stored within the cluster itself, making the cluster 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 cluster. Classes and types are not used by the language-independent layers of RDOSS, and will be ignored in the remainder of this document.

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

The ideal GC would 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 such as the Internet.

Second, at any point in time, the major part of the object graph is swapped out to disk, and cannot be accessed economically. Since by definition swapped-out data has not been accessed for some time, it is unlikely that tracing it would discover any new garbage.

A beneficial side-effect of some garbage collection algorithms is to improve locality by *compacting* sparsely-populated clusters. A live object is re-allocated at a new address, and pointers to it *patched* to refer to the new location. Hence

[5] This information is redundant with classes, but is self-contained and language independent.

the third problem, that relocating appears to require write-locks that compete with the applications'.

Fourth, existing concurrent GC algorithms require synchronization between the mutator and the collector. This is implemented by instrumenting *every* mutator instruction that could either read (this is called a "read barrier") or assign (a "write barrier") a pointer, in order to inform the collector of the value read or written. Since what would have been a single assembler instruction is instead replaced by a sequence of code, this slows down the applications considerably. The collector is not portable, being strongly coupled with a particular compiler that will generate the correct barrier code.

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 opportunistic, non-synchronized, piecewise, local traces. Each cluster is traced at each site where it is cached, and the results summarized at the cluster owner (this will be explained in Section 4.2.1). Groups of multiple clusters mapped at some site are scanned at once, thus collecting cycles of garbage that span clusters.

In order to not compete with the application, the GC works on a separate data set; namely, any datum mapped 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 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 cluster, apart from the assumption that only the owner can commit a write. 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 for 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 read- or write-barrier; the algorithm discovers pointer assignments the next scan of the collector itself, effectively batching pointer assignments.

The collector is composed of three sub-algorithms:

1. Collecting a single replica of a single cluster on a single site,
2. Collecting a group of clusters on a single site,
3. Collecting the multiple, possibly incoherent, replicas of a single cluster at all its sites.

We will explain each of these sub-algorithms and argue their correctness, while also justifying the design decisions listed above. The first two sub-algorithms 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* clusters within groups limited by what is economically feasible. Specifically, trace stops when it would require input-output or network or lock traffic. Trace groups are different at each site and change dynamically. (ii) The algorithm uses *reference counting* (via the scions, at the cluster and group boundary) when tracing would be too expensive.

3.3 Scanning a single replica at a single site

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

The in-list presented in Section 2.4 identifies all the pointers that reach into a cluster. 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 w.r.t. the cluster, *i.e.,* it will deallocate any cycle of garbage that is entirely within the cluster. However it is conservative w.r.t. other clusters, since it can not deallocate a cycle of garbage that crosses the cluster boundary; thus, the in-list serves as a reference counter for inter-cluster references.

Here is how a trace proceeds. Any object pointed from the in-list is marked reachable. An object inside the cluster, pointed from a reachable object, is itself marked reachable. If a reachable object points outside the cluster, a corresponding stub is allocated in the out-list (with no attempt to mark the pointed-to object). 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 cluster 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-cluster reference has been created; the collector sends a `create` message to the (owner of the) target cluster so that it can create the corresponding scion.

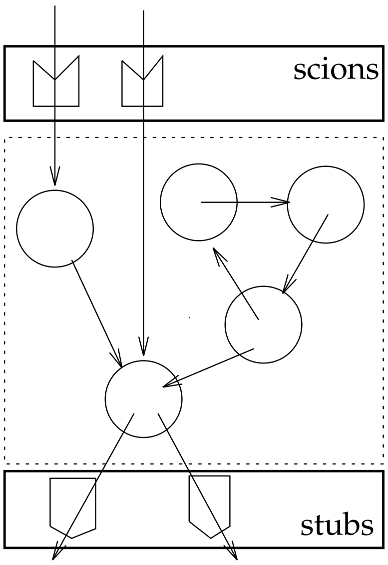

Figure 2: *Collecting a single replica of a single cluster*

Pointer updates are not noticed until the cluster is scanned (in contrast to concurrent garbage collectors where the mutator must immediately inform the collector 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 *delete*s.

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.,*

y := x; x := 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-cluster reference no longer exists. A delete message is sent, asynchronously, in order to remove the corresponding scion in the target cluster. To avoid race conditions, no delete message is sent until all the create messages have been. Furthermore, in the case of a replicated cluster, 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 cluster, and an application later takes a write lock and modifies it, then the collector aborts and its effects are undone. 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 cluster, but conservative with respect to possible garbage referenced from another cluster.

The collector may choose to move a reachable object to a different address (possibly in a different cluster). To do this, the collector must take a write reserve lock (recall that the reserve lock protects the addresses); which it will only do if this site is the owner of the cluster and no application already has a read or a write reserve lock. This ensures that the collector will not cause any reserve lock traffic and will not compete for locks with the application.

When an object has been moved, any pointers that reference it must be patched. This would entail finding the clusters containing these pointers, and taking a write lock on them. In fact, pointer patching can be delayed until such source clusters are next mapped and swizzled.[6] Consistency is not a problem because the collector will move an object only if the source cluster is not protected by a reserve lock.[7]

3.4 Group scanning of multiple clusters at a single site

Just as the collector can scan a single cluster replica at a single site, it can scan any group of clusters 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 cluster), and that scanning continues across cluster boundaries, as long as the group is not exited. Figure 3 shows an example group of two clusters.

A group will contain only clusters that are not write-locked by an application. As above, a collection aborts if the application modifies a cluster that has already been scanned. This ensures that the collector does not compete with the application locks. For the same reasons as the single-group case, the algorithm is trivially correct. Group collection is complete w.r.t. clusters in the group, *i.e.*, a cycle of garbage, possibly crossing cluster boundaries, but remaining within the group, will be collected. It is conservative w.r.t. clusters not in the group.

---

[6] Since the first version of Larchant-RDOSS does not swizzle, it follows that its garbage collector doesn't move objects either.

[7] Larchant-BMX [6, 5] does not have reserve locks. Like any update, patch consistency is ensured by the fact that only the owner of an object can move it.

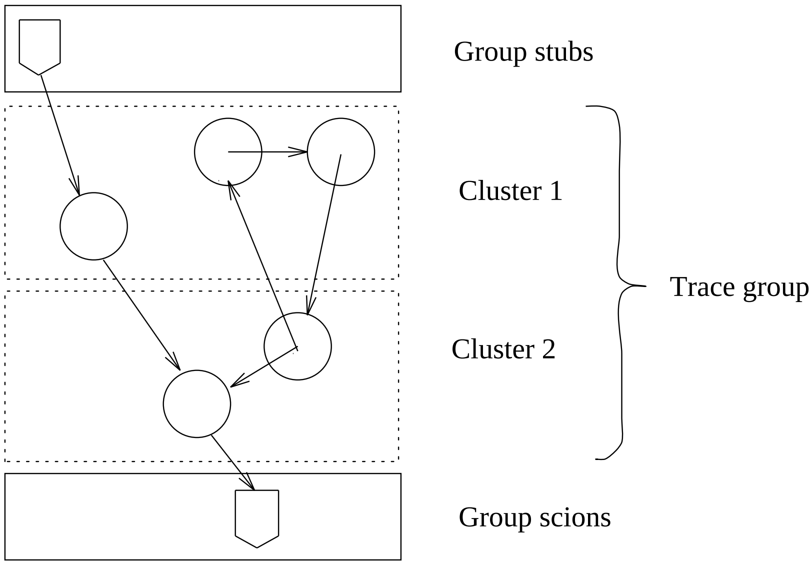

Figure 3: *Collecting a group of clusters at a single site*

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 clusters 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.

These heurisitics do not collect cycles of garbage that reside partially on disk, on another server, or in a cluster that is actively used by applications (a "hot spot"). Collecting such a cycle involves costs that need to be balanced against the expected gain. A cluster swapped to disk has not been accessed recently, so it has a low probability of containing new garbage.

The following strategy theoretically collects all cycles of garbage that span any set of clusters small enough to fit in some server: repeatedly cache a random subset of clusters, and collect it as a group. While theoretically complete, this strategy is extremely inefficient. This result does tell us however that for completeness, a heurisitics should contain a stochastic component.

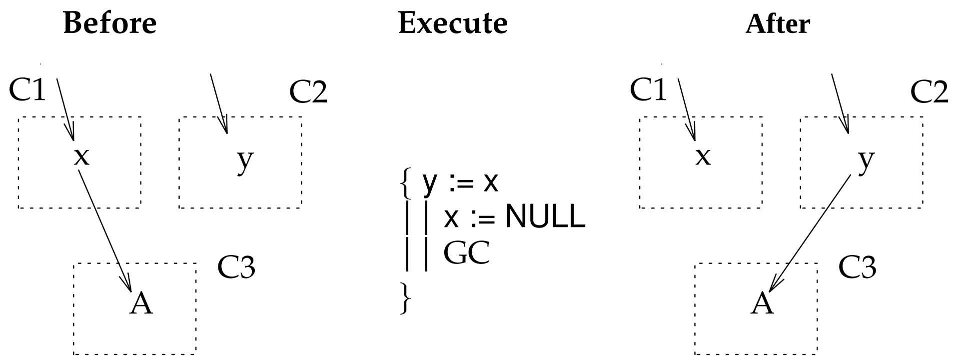

Figure 4: *Possible race conditions with concurrent read-write-scans of a replicated cluster.*

We plan to implement only the locality-based heuristic as a first shot. If experimental results mandate, we will explore others.

4 Collecting the multiple copies of a cached cluster

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 cluster 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 cluster C1 points to object A in cluster 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 cluster 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

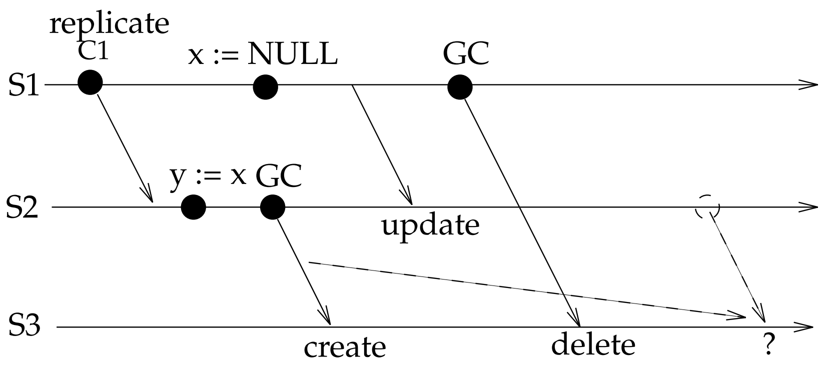

Figure 5: *Timeline for the execution of Figure 4, showing the effect of a late create or a slow create message.*

delete message".) Furthermore, 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.

Figure 6 shows an obvious solution to this problem: serializing creates with respect to deletes. In this example, scanning a cluster as soon as a write commits ensures against early deletes. The create message is sent synchronously. An update message is also synchronous, from a site performing an update to the cached copies of the cluster, ensuring that a delete message is not sent until the corresponding update has been applied at all copies.

This simple solution is undesirable, because it slows down the application of updates, and couples the GC to the coherence protocol. It should be possible to improve this situation by batching some messages, but a solution that uses only asynchronous messages will have better performance.

4.2 An asynchronous solution to the problem

An asynchronous solution to the above problem requires avoiding 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 7 and 8.

4.2.1 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

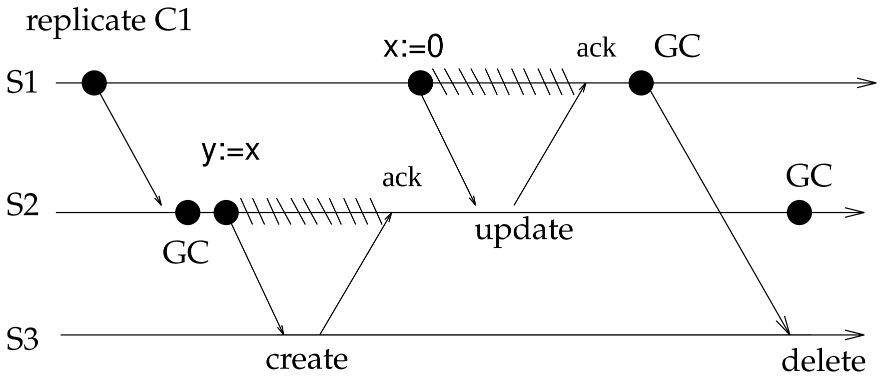

Figure 6: *Timeline from Figure 5, modified with additional synchronization to delay the delete after the create.*

update has been applied at all the copies of the cluster, 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 clusters before accepting updates on the same cluster.

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 cluster is collected at its site, and the results are summarized at the cluster owner. We can now explain precisely what that means.

Each cluster 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 takes into account the objects reachable at that site. After the collection, each Partial Out-List is sent to the owner of the cluster in a POL message.

The owner collects all Partial Out-Lists; the complete out-list for the cluster is just the union of the most recent Partial Out-List of each copy. The owner sends 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 cluster can make a new stub appear in that cluster.

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 cluster, say

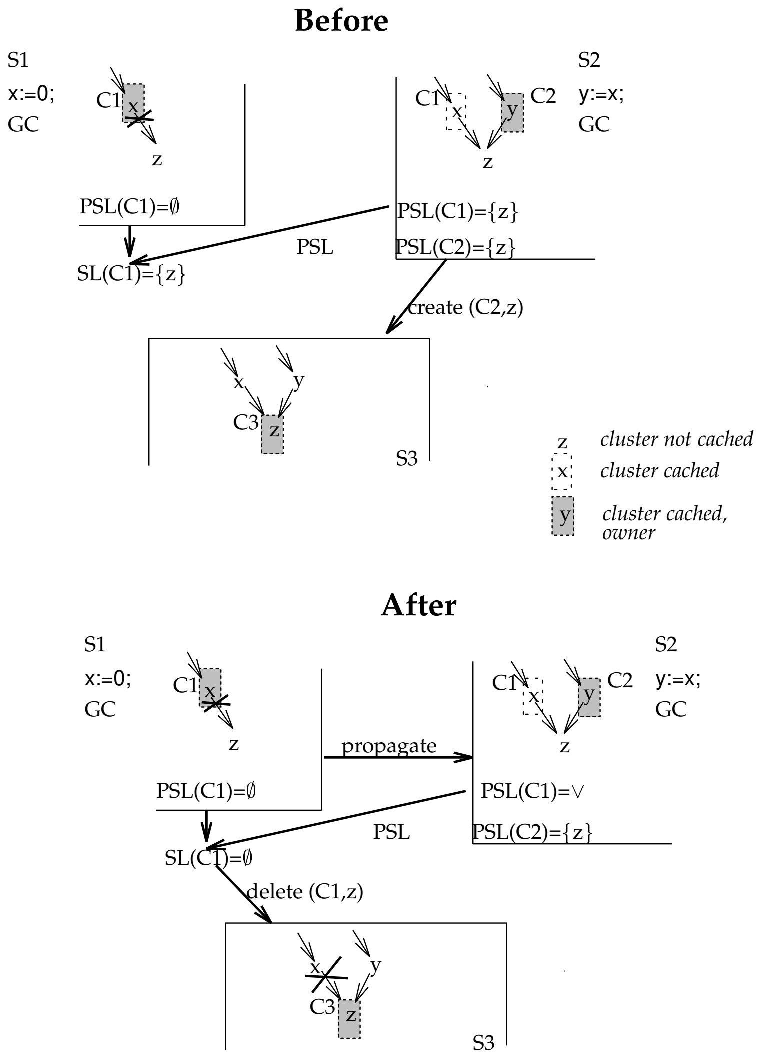

Figure 7: *Union of Partial Out-Lists solution, before and after application of a pointer update.*

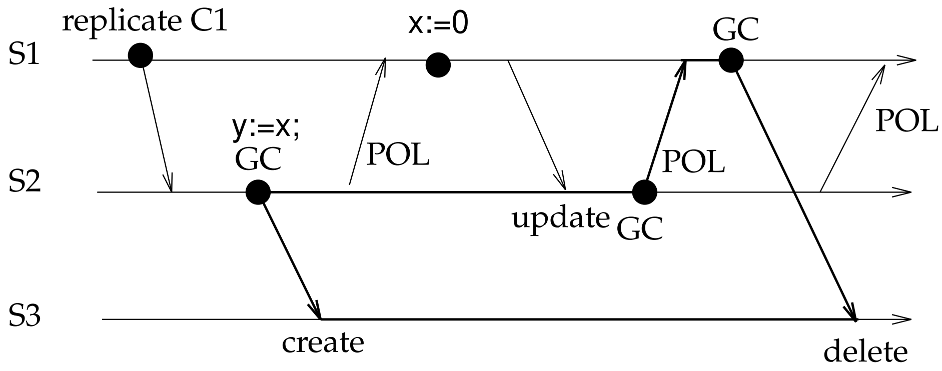

Figure 8: *Timeline for the Union of Partial Out-Lists solution, illustrating asynchronous messages and causal ordering.*

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 cluster C1 containing x has not been modified at that site.

Property (ii) is by definition of garbage.

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

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.

4.2.2 Avoiding fast delete

A careful examination of Figure 8 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 [2] will therefore ensure that the create message will not be overcome by the corresponding delete.

Since Larchant-RDOSS runs on top of Isis, we simply send the create, POL, and delete messages using the causal communication primitive cbcast. Larchant-BMX [6, 5] does not depend on Isis; in this system, we use piggybacking to implement causality.

5 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 clusters, 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 writer per cluster. 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).

We have furthermore shown a version in which all collector messages are asynchronous; however this version either rests upon or simulates 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.

An application process bears a cost at the time of binding and unbinding a datum into memory. The bind time cost is the cost of mapping a copy of the data, swizzling, and applying a lock. The unbind cost is modifying the reference map(unswizzling), unmapping, unlocking, and propagating changes. Swizzling and unswizzling are useful for supporting memory compaction. The other costs are the unavoidable consequence of sharing.

Acknowledgements

We wish to thank Martin Hampton and Warren Reier for patiently explaining the requirements of financial applications, and providing the original impetus for this work. Rich Zippel provided an attentive ear, numerous suggestions, and moral support. Thanks also to Ken Birman and Robert Cooper for their help and for the opportunity to do this work. This work benefited from discussions with Liuba Shrira and Luc Bougé.

References

[1] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *Computer*, 26(4):360–365, 1983.

[2] 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.

[3] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, Pittsburgh, PA (USA), September 1991.

[4] Kenneth Birman, Andre Schiper, and Pat Stephenson. Fast causal multicast. Technical Report TR-1105, Dept. of Comp. Sc., Cornell University, Ithaca, NY (USA), April 1990.

[5] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA (USA), November 1994. ACM.

[6] Paulo Ferreira and Marc Shapiro. Garbage collection of persistent objects in distributed shared memory. In *Proc. of the 6th Int. Workshop on Persistent Object Systems*, Tarascon (France), September 1994. Springer-Verlag.

[7] 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.

[8] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.

[9] 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.