PerDiS PPF Case Study: Fitting a Distributed Garbage Collection Algorithm to a Persistent Distributed Store Architecture

Xavier Blondel[1] Paulo Ferreira[2] Marc Shapiro[3]

August 11, 1997

Abstract

Several garbage collection algorithms exist, but it is not always straightforward to adapt them to a given platform. In this article, we describe our work to fit a distributed garbage collection algorithm to a persistent distributed store called PerDiS. We pinpoint ordering and consistency problems, and present the way we solved them. We also address the dilemma of performance versus modularity.

1 Introduction

This article describes our work aiming at integrating a distributed garbage collection algorithm, designed for the Larchant project [FS96], into a persistent distributed store named PerDiS.

This article is organized as follows: Section 2 describes the PerDiS project and explains the needs for a distributed garbage collector; section 3 describes the Larchant Garbage Collection algorithm and section 4 describes how we fitted this algorithm to the PerDiS platform.

2 The PerDiS project

This section describes the PerDiS project, and focuses on the PerDiS architecture. This will help to understand the needs for a distributed garbage collector.

2.1 Context

The PerDiS project is described by Shapiro et al. in [SKR97], and in the PerDiS Web site [Per97]. PerDiS is a Long Term Research ESPRIT project aimed at designing and implementing a **PER**sistent **DI**stributed **S**tore, for applications in the cooperative engineering area. We provide a *persistent* store, based on Persistence By Reachability [ABC+83], which relieves the programmer of the burden of saving and retrieving data to and from a stable storage. This store is also *distributed*, giving a simple, secured and efficient way of sharing data.

This project is in its first year; we are currently working on a Preliminary Platform (or PPF for short). This platform will evolve next year to an Intermediate Platform (IPF) and we will end up with the Advanced Platform (APF). This article addresses the implementation of a Garbage Collector in the PerDiS PPF α-release.

The way the PerDiS architecture is designed dictates the way we will implement a distributed garabge collector for this architecture.

2.2 PerDiS basic components

The fundamental PerDiS component is the *cluster*. It is the user-visible unit of memory, storage, naming, and protection. Each cluster has a name, and can be seen as a distributed shared memory, therefore containing data. These data can be modelled as a graph, where the nodes are the objects and the edges are the references between these objects. Some of these objects are known as *roots*: they also have a name, and they are the object graph entry points for an application; that is, an application willing to access a given object retrieves a root object, using its name, and follows a chain of references leading to the target object.

[1]Xavier.Blondel@inria.fr, INRIA – Projet SOR—Work done while PhD student at CNAM, Laboratoire Cédric, Paris, France.
[2]Paulo.Ferreira@inesc.pt, INESC
[3]Marc.Shapiro@inria.fr, INRIA – Projet SOR

It is allowed to have a reference going from one object in a given cluster to an object in another cluster; this is called an *inter-cluster reference*, which is the opposite of an *intra-cluster reference* linking two objects in the same cluster.

In memory, both intra- and inter-cluster references are simple pointers. Nevertheless, for Garbage Collection and swizzling[4] purposes, an outgoing inter-cluster reference is represented by a *stub*, whereas an incoming reference is represented by a *scion*. There is always one and only one scion for each stub.

Sharing data is performed, in the PPF, using *cluster replication* and *transactional semantics*. An application willing to read from a cluster gets a replica of it; an application writing to a cluster also gets a replica of it, but all other copies of this cluster are invalidated. Accessing a replica implies locking the cluster. A global locking mechanism, based on entry-consistency, is provided to applications by PerDiS. Pessimistic and optimistic transactions use these locks in different ways, in order to behave according to their own semantics.

The last site which wrote to a given cluster is the *owner site* of this cluster; the replica owned at this site is the *master replica* of the cluster.

In further platforms, replication will be done with a finer grain, reducing concurrency of access.

2.3 PPF architecture

The PerDiS architecture is based on two main components: the *User Level Library* (ULL) and the *PerDiS Daemon* (PD). The ULL is linked to the application code, providing the PerDiS execution environment. The PD provides the applications with the PerDiS functionalities. There may be several applications, but there is only one PD per site.

The PD is composed of a Transaction Module, a Cache Module, a Cluster manager, and a Distributed Garbage Collection module. In the PPF, caching is based on pages, and the Cache Module is not aware of their semantics (e.g the difference between meta-data and plain data). This semantics, however, is known to the Cluster Manager, which is a sort of interface between the ULL and the PD. The Cluster Manager keeps track of the types.

Swizzling is performed by the Cluster Manager, translating persistent addresses in the store to virtual addresses in the application addressing space, as described in [Mos92]. The Cluster Manager uses type information in order to swizzle pointers contained in objects in the persistent store. The Garbage Collector will take advantages of this type information.

[4]Swizzling will be defined in section 2.3.

2.4 Needs for a Distributed Garbage Collector

A Garbage Collector has two main goals: first, it avoids memory leaks, by automatically deallocating unused memory; second, it avoids dangling pointers, freeing only unreachable objects.

It is very difficult to manually spot useless objects in any application with a rather complex object graph. It is impossible if the application is distributed: how might one application know if there is no application still needing a given object?

3 Larchant Garbage Collection algorithm

The Larchant Garbage Collection algorithm, as described in [FS96, Fer96], is a Distributed Garbage Collection Algorithm, designed to minimize induced communication and synchronization costs.

The Larchant algorithm collects each cluster independently from each other. It uses a tracing garbage collection algorithm —either mark-and-sweep or copy, as described in [Wil92]— inside a cluster; the set of scions is the root of this tracing.

Inter-cluster references are collected using a reference listing method. The scions of a cluster represent the incoming references; deleting an object containing an outgoing reference leads to delete the corresponding stub and scion. Therefore, this deleted scion is no longer a root for the intra-cluster collection, enabling to spot more garbage objects.

But reference listing, as well as reference counting, prevents us from spotting garbage cycles, as this was first pinpointed by McBeth in [McB63]. More precisely, in our case we miss distributed garbage cycles. To solve this problem, the Larchant Garbage Collection algorithm collects several clusters at the same time. Such clusters are grouped with an heuristic that tries to minimize extra I/O: we collect the group of clusters that is currently locally cached.

A great deal of effort was spent on minimizing synchronization costs induced by the Garbage Collector on applications functioning. As a matter of fact, Garbage Collection imposes minimal consistency requirements, as proved in [FS94].

To avoid distributed race conditions, the Larchant Garbage Collection has strong ordering

constraints on stub/scion creation and deletion, as described in [Fer96]. As a matter of fact, a race condition between a scion creation message and a scion deletion message may lead to the following scenario: first, the scion is deleted; then, the object referred to by the scion is also deleted; finally, a new scion is created, which refers to the deleted object. We then conclude, as expected intuitively, that it is mandatory to perform scion creations before any scion deletion.

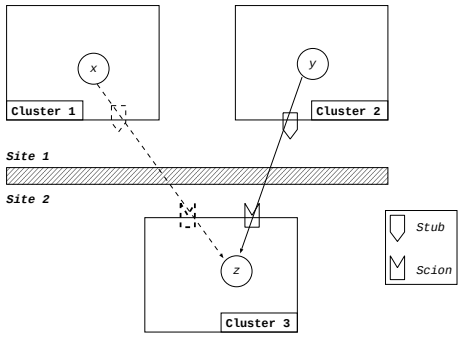

Figure 1: An example of the need of a remote scion creation

4 Integrating the Larchant Garbage Collection algorithm into PerDiS

The Garbage Collector has two main functionalities: the first one, **Stub/Scion Creation**, creates stubs and scions upon which the second functionality, **Garbage Reclamation**, relies to perform the actual garbage collection. The second operation can not be started as long as the first one is not over. This way, the ordering constraints described in the previous section are ensured.

4.1 Stub/Scion Creation

Stubs and scions must be created in a way that makes them available in any replica of the cluster. Hence, we decided to create stubs and scions in the master replica.

Stub creation is trivial. Creating a stub in a cluster means an outgoing reference has been added to this cluster, i.e. this cluster is write-accessed: it is the master replica. So, before the transaction releases locks on this cluster, we create the stubs by spotting every new outgoing reference. At unswizzling time, the Cluster Manager walks every object and turns every reference contained to a persistent one; during that phase, it is easy to spot outgoing references, and to create the corresponding stubs, if they do not exist yet. Since this master replica was write-accessed, there are no other valid replicas.

Unfortunately, the Cluster Manager cannot create the corresponding scions. Indeed, the cluster which will contain a to-be-created scion is not necessarily locally owned, maybe not even locally cached. Let us consider the example in Figure 1.

Site 1 accesses two clusters, **Cluster 1** and **Cluster 2**. The latter contains one object y, with a reference to the object z in **Cluster 3**, which is owned by **Site 2**. Let us assume that **Site 1** performs the following operation: $< x := y >$; **Cluster 1** now has an outgoing reference to z, drawn with a dotted line in the figure. It is easy to create the stub, but the scion (drawn with a dotted bold line) must be created on **Site 2**, i.e. remotely.

To avoid distributed races, the Larchant Garbage Collector has ordering constraints on scion creation and deletion, as mentioned in section 3. So, the Garbage Collection Module takes care of remote scion creation. It is the only module in the PD aware of both distribution and meta-data (the Cluster Manager knows the latter, not the former; the Cache Module knows the former, not the latter). Before releasing the locks of the transaction that created the new reference, the Garbage Collector transmits the remote scion creation requests to the owner sites of the target clusters and waits for a succesful acknowledgement. An owner site PD receiving a scion creation request creates the corresponding scion.

This master replica of the cluster may not be the only one, since the transaction creating clusters did not necessarily write-lock this cluster; so, these replicas must be updated.

Let us consider an application, which is running on **Site 3**, using the clusters of the example in Figure 1. We have the situation shown in Figure 2, where *old* and *new* refer to *before* and *after* the scion creation discussed in the former example (it is obvious that all the master replicas are *new*). **Cluster 1** was formerly write-accessed by another application: it has been invalidated. On **Site 3**, we have a *new* replica of **Cluster 1**. On the contrary, an *old* replica of **Cluster 3** is still stored on **Site 3**.

The application is reading **Cluster 1**. It tries to follow the reference starting from x, so the Cluster Manager at **Site 3** accesses the newly created stub, and tries to access the scion in **Cluster 3**. But it

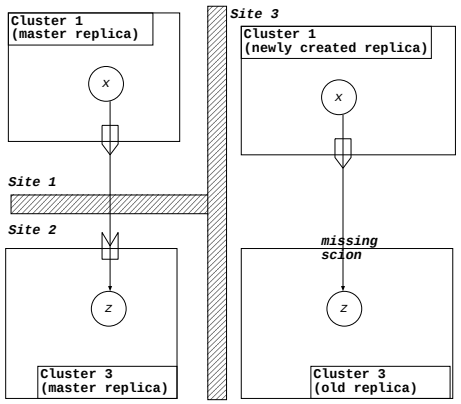

Figure 2: An example of a missing scion

is an *old* replica of this cluster which is stored at **Site 3**: regarding scions, it is not up to date. The Cluster Manager then knows **Cluster 3** meta-data is out of date, and just retrieves it from the owner site. This way, the meta-data is propagated lazily but safely.

We just saw how stubs and scions are created. Now, let us see how they are used during the Garbage Reclamation phase.

4.2 Garbage Reclamation

While integrating the Larchant Garbage Collection algorithm into PerDiS, we try to limit module dependencies and we need a way to make a consistent snapshot of the memory.

4.2.1 Module dependencies

We want the Garbage Collector to be as independent of other modules as possible, i.e. we do not want that modifying a module implies modifying the Garbage Collector. In fact, we want to encapsulate the various modules' functionalities.

Since Stub/Scion Creation does not imply complex operations for the Garbage Collector (only scion creation), module dependencies are rather weak. On the contrary, Garbage Reclamation is a complex operation, and the easiest way to implement it would be to increase module dependencies.

The main problem is with the Cluster Module structure: if we plug the algorithm in this structure, we then obtain an intricated piece of code, which is difficult to modify. So, we decided to rely on generic interfaces rather than changing the Cluster module; we favor independence over performance.

The Cache module gives us the data, whereas the Cluster module gives us access to meta-data (stub/scion list and free list) and type information. We have everything we need to perform Garbage Reclamation.

4.2.2 Consistency worries

As we said in section 3, the Garbage Collector does not need a consistent view of the memory to perform its operation, but we are not totally out of consistency worries.

The problem is the following: each cluster the Garbage Collector is working on must be consistent. Indeed, it would be impossible to follow a chain of references if the cluster is only partly written: the object graph inside the cluster would not be consistent.

The Garbage Collector gets copies of the clusters from the Cache Module in order to garbage collect them. The Cache module has an interface based on pages; to read a cluster from the Cache, we need to read pages, *which can not be done in an atomic operation without locking this page, at least for reading.*

Moreover, the Garbage Collector also has to modify meta-data of clusters, possibly concurrently with other applications. Indeed, an application may be allocating while the Garbage Collector is deallocating, causing concurrent accesses to the free list, i.e. the meta-data. Here, again, we need serialization of accesses.

One possibility is to perform the Garbage Collector as an optimistic transaction, which aborts in case of concurrents writes by an application [SF95]. However, better performance will be possible using concurrency control based on the fact that we know the semantics of accesses performed by the Garbage Collector.

The problem is the following: the Cache Module provides the applications with global locks. Several read locks, but only one write lock, may be taken at the same time on a given cluster by applications. For a Garbage Collector using this kind of locks, writing in a cluster means invalidating all the replicas of this cluster; but we want to minimize synchronization costs induced by the Garbage Collector. Therefore, we implemented a specific Garbage Collector locking mechanism, which is not based on the kind of locks used by transactions. It works as follows: when the Garbage Collector starts, it asks the Cache for non-write-locked clusters. It then

reads them and performs the Garbage Reclamation. During this phase, if it happens that a cluster is write-locked by an application, the Cache sends a notification, forbiding the Garbage Collector to update this cluster.

When Garbage Reclamation is over, the Garbage Collector updates the Cache with the cleaned clusters, taking, to do so, an exclusive lock, suspending applications trying to read- or write-lock a cluster. Note that these Garbage Collection locks are local, hence not leading to invalidations, even though clusters are written by the Garbage Collector. These Garbage Collection locks are only serializing writes performed by the Garbage Collector, avoiding that an application reads a half-written cluster.

For the α-release of the PerDiS Preliminary Platform, we only refuse to write in a write-locked cluster; work performed by the Garbage Collector in this cluster is lost, and will have to be done again later. Further versions will include a system of *Garbage Recycling*, which will enable modifying the meta-data of a write-locked cluster, in order to have better performances.

5 Conclusion

We presented the way we fitted an existing distributed garbage collector algorithm into a persistent distributed store. We noticed that such an operation arises a lot of ordering and consistency problems. We also noticed that lowering module dependencies is a trade-off between modularity and performance.

We are currently finishing the implementation of the PerDiS PPF α-release. All the algorithms are stable, and we are only dealing with coding.

Future work includes elaborating the design of the Garbage Recycler, to enable concurrent writes. We will also have to deal with future evolutions of the PerDiS platform, mainly fine-grain locking, fault-tolerance and security.

References

[ABC+83] 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.

[Fer96] 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. English, with a long abstract in French.

[FS94] 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.

[FS96] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proc. 16th Int. Conf. on Dist. Comp. Syst. (ICDCS)*, Hong Kong, May 1996.

[McB63] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, Sep 1963.

[Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[Per97] PerDiS ESPRIT Project—LTR 22533. The PerDiS project: a Persistent Distributed Store, 1997.

[SF95] 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.

[SKR97] Marc Shapiro, Sytse Kloosterman, and Fabio Riccardi. PerDiS — a persistent distributed store for cooperative applications. In *Proc. 3rd Cabernet Plenary W.*, Rennes (France), April 1997.

[Wil92] 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.