Garbage Collection of Persistent Objects in a Distributed Object-Oriented Platform

Pedro Sousa

pms@sabrina.inesc.pt

IST – INESC

R. Alves Redol, 9, 1000 LISBOA, Portugal

Abstract

This document describes distributed garbage collection in IK, an object-oriented platform which facilitates the construction of persistent and distributed applications. Object references can be freely passed during remote invocations or stored persistently. IK supports orthogonal persistence with type inheritance. Objects become persistent when reachable from a Persistent Root. Objects are created and manipulated uniformly, independently of the time they persist, or their physical location.

1 Introduction

IK [1,2] is an object-oriented platform which intends to simplify the development of distributed and persistent applications. It generalizes the view of volatile and persistent data by considering all objects to be maintained by the system while being part of the transitive closure of an Eternal Root. Persistence is a dynamic attribute of objects, and orthogonal to their classes. From the programmers point of view objects are created and manipulated uniformly, independently of their type and the length in time they persist. IK has an Eternal Root, a well known repository of object references, and objects are maintained by the system while reachable (either directly or indirectly) from this root.

Objects are uniquely identified through long lived, context independent identifiers, called Low Level Identifiers (LLI). A Run-Time Support (RTS) linked with applications swizzles LLIs to and from pointers whenever references enter or leave context boundaries. Thus LLIs are not seen by applications code.

Object invocation is the basic primitive of the system, embodying all of the features required for transparent handling of persistence and distribution. Object faults are triggered solely by intercepting object invocations, and direct accesses to other object's instance data is not allowed. Faulty objects are either mapped and invoked locally or, if already mapped elsewhere, remotely invoked. The platform also provides primitives to migrate objects, but migration policies must be defined by applications.

IK is mainly targeted to cooperative applications executing in a local network comprising a few tens of workstations under a common administration policy. Applications are written in EC++ [3], a language similar to C++ with some semantic extensions and restrictions. The programming environment is based on a version of the ET++[4] library ported to our platform, providing a framework for application development. Some of the applications built on top of IK are a cooperative calendar manager that allows several users to schedule and negotiate meetings, a cooperative graphical editor that supports multiple users concurrently, a browser and an inspector for application development and a tool for the analysis, design and implementation of distributed applications using a uniform high-level description and semantic replication. The applications comprise 452 classes and 6216 methods, a number we consider sufficient for a minimal evaluation of the platform.

The work started seven years ago within the framework of the ESPRIT project COMANDOS[5]. IK runs in user mode on a network of heterogeneous UNIX machines – Suns (3 and SPARCstations with SunOS/Solaris), Bull DPX/2 (B.O.S.) and i386 based machines [6] (running the Mach 3.0 micro-kernel).

System Architecture

We distinguish two different type of nodes in the system: computing nodes and storage nodes. The former may have low availability, while the later must be highly available. Applications can run in any node but objects are stored persistently only in storage nodes.

A set of Storage Servers (SS) and Name Service (NS) run in storage nodes. The SS provides support for object storage and the NS manages a hierarchical name space spanning machine boundaries. The RTS uses these two services to store and retrieve persistent objects and their user-defined names.

Implementation

Although the model does not distinguish volatile objects from persistent objects, we exploit their different characteristics: when created objects are volatile, and promoted to persistent when become globally known[1]. An object becomes Globally Known when an LLI is assigned to it. This happens when references to the object are sent as parameter in remote invocations, or inserted in the Eternal Root. Both volatile and persistent object can be clustered and stored in disk without changing their visibility[2] (see figure 1). Globally Known objects become clusters when stored on the persistent store. The RTS keeps track of all LLIs received or created during the execution of an application in the KOT table (*globally Known Object Table*).

[1]We call persistent to globally known objects and volatile to non globally known objects for historical reasons.

[2]The clustering algorithm may eventually assign LLIs to Volatile Objects promoting them to Persistent Objects.

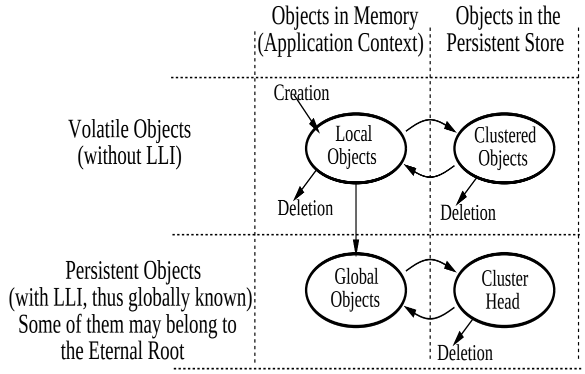

Figure 1: Visibility of objects when mapped in applications and in the persistent store.

Object Clustering. We use a clustering policy where each cluster has a single globally known object, called cluster head. The body of the cluster is composed by all objects that are only reachable through the head object, i.e. there are no references outside the cluster for the cluster body, only for its head object. Therefore, objects belonging to the cluster body do not have LLIs, and the whole cluster is identified by the LLI of its head object. Objects are appended to clusters as they are being traversed in a depth-first approach. Clusters are stored in individual files, in a standard data format (XDR).

In IK, objects are clustered when they leave application's address space, either to be saved in the persistent store or migrated to other applications. A cluster is mapped upon invocation of its head object. When that happens the RTS memory-maps the whole cluster and unfolds the head object: converts its data to the current machine format and swizzles its references. The other objects will be unfolded only when invoked.

Garbage Collection. We use three independent garbage collectors: a first one recycles objects within an application address space (Local Objects according to figure 1), a second one to recycle objects within a cluster (Clustered Objects), and a third one to eliminate clusters which are no longer reachable from the Eternal Root.

Local objects are recycled using a generation scavenging algorithm. In our environment, for local collections to proceed autonomously, it is also necessary to consider the entries in KOT as local roots [7]. When a cluster is mapped, clustered objects are considered to belong to the oldest generation.

Cluster-wide garbage collection recycles dead objects within each cluster. Clusters are garbage collected off-line, by a process running continuously on each storage node. A cluster is locked while is being recycled, forcing applications willing to use it to wait until the cluster is saved back.

Finally, a system-wide garbage collector deletes clusters in the persistent store that are no longer reachable from the Eternal Root. A first version of this collector was a straightforward implementation of the mark-and-sweep algorithm. It was comprehensive, centralized, non-robust and the whole system must stop for garbage collection. Nevertheless, it allowed us to characterize the population of clusters, and design a new system-wide garbage collector, briefly presented here.

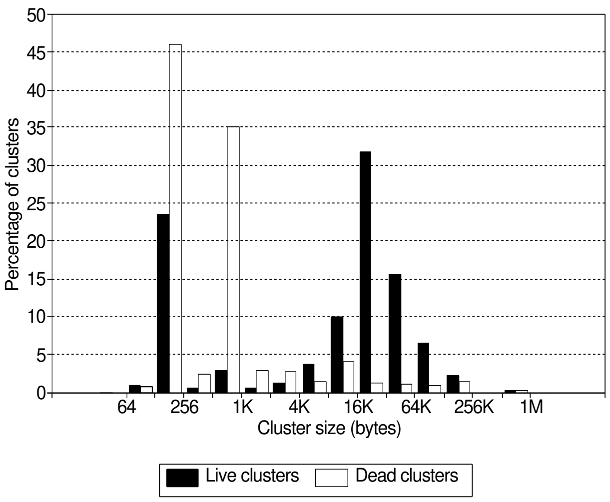

Figure 2: Distribution of Clusters Population.

2 Cluster Population

In figure 2 we present the distribution of cluster population according to their size. Bars represent the percentage of clusters whose size is between two consecutive powers of two. White bars refer to dead clusters and dark ones refer to cluster reachable, directly or not, from the Eternal Root. They are not related to each other as that would make them depend on the periodicity of garbage collector runs. Thus the sum of white bars is 100 %, as well as the sum of dark bars. The average size of objects in clusters is approximately 42 bytes.

One can see that approximately 70% of live clusters are bigger than 4K, most of them containing between 5 hundreds and 10 hundreds of objects. They result from applications that handle information in a hierarchical fashion, where nodes of the graph are not exported outside the graph.

The remaining 30% of live clusters consist of very small graphs of object, containing a few objects. Typically, such clusters are generated when graphs are traversed remotely. In fact, some of our applications do tend to fragment clusters completely, because they scan lists mapped remotely, forcing the system to promote each visited element to a cluster. This is the main drawback of our clustering approach. However, such applications tend to perform poorly given the unbounded number of cross-context invocations. Some of these applications have been changed to improve performance, mainly by reducing the amount of cross-context invocations. They were redesigned to hold local replicas of frequently accessed objects, being able to compute with a reduced number of remote invocations. The reduction on clustering fragmentation was noticeable.

Analysis of the population of dead objects has revealed that about 80% of the clusters are smaller than 1K. They contain mainly transient information used in communication between the various processes of a distributed application. In most cases, such objects become globally known when references to them are sent as parameters in remote invocations. They are stored when the processes holding them finish their execution.

Detect and prevent the storage of this garbage objects proved to be a difficult task because:(1) most of the garbage objects form distributed cycles and, (2) some of the objects are still alive when the process holding them finishes, they only become garbage when other processes of the distributed application have finished as well.

Trying to prevent the storage of objects that are garbage at the termination of a given process requires an “on-line” algorithm. Due to the distributed cycles, refence counting algorithms (e.g. [8]) are not very useful here. The same applies to protocols tightening incoming and outcomming references [9]. We could follow Lieberman suggestion and evacuate objects from terminating processes to remaining ones that still refer to them [10]. However, this is also not feasible in persistent environments because it may lead to an unbound number of object migration. One could send instead incoming and outgoing references and the paths between them to a central entity [11], which would identify garbage objects. Although this could prevent the storage of objects that are garbage at the termination of each process, it is not a complete solution. Therefore, we decided to start with a more general approach, and develop an acceptable distributed garbage collector of persistent objects. Early detection of garbage objects at process termination is left for later optimizations.

3 Collecting Persistent Objects

The algorithm intents to recycle globally known objects (clusters) residing in the persistent store that are no longer reachable from the Eternal Root. Our system-wide collector is a mark and sweep with the following properties:

- Is "non-comprehensive". We do not intent to recycle garbage referenced by running applications. We wait for applications to terminate, or cease to refer to such objects. The algorithm does not reclaim cycles between the persistent store and the applications, but it reclaims distributed cycles between the several Storage Servers that make the Persistent Store.

- Is "non-stop-the-world", although it may delay a mutator while a cluster which the mutator wishes to access is traversed (not its descendents).

- Is robust (i.e. behaves correctly) in the presence of certain failures:

  - **Network failures**. We assume pair-wise availability between storage servers with no long-term congestion or partitioning. Communication is based on "at-least-once" and "at-most-once" asynchronous message delivery semantics. The algorithm tolerates duplication of the former messages and losses of the later. It also tolerates reordering of messages, although delivery is expected to be time bounded.

  - **Node failures**. The algorithm tolerates "fail-stop" failures in computing nodes. They can restart computation without keeping previous state. Failures in storage servers are tolerated as long as they be able to resume computation, and may prevent the algorithm to progress. If a failed storage server cannot resume computation, garbage collection will be gracefully aborted.

- Is "safe" (i.e. it never collects non-garbage), provided no other but the failures mentioned occur.

- Is "conservative", in that garbage referenced by running mutator is not collected in a timely fashion.

- Is "live" (i.e. it eventually collects all garbage), provided that: each graph of garbage is not referenced by any mutator, at least during one garbage collection.

- Is control centralized. Although a central entity controls the logical rendezvous of the phases of the collector, the the load due to garbage collection is distributed in proportion to the population of objects at each node.

The core of the algorithm executes within the storage servers of each storage node, but it needs cooperation from the RTS of mutators to trace incoming and outgoing references.

Typical mark-and-sweep algorithms use a three color marking schema [12]: black objects are live and traversed, gray objects are live but not traversed yet, and white objects are potentially garbage. Black objects do not refer to white objects directly, and mutators can only compute on black objects. Accesses to gray objects must be trapped, and gray objects promoted to black prior the resume of mutator execution. In these schemas, objects are marked twice, one to move from white to gray an another from gray to black.

In a persistent store, moving an object from gray to black may yield many disk accesses to make its descendents gray, which may result in unacceptable delays in running mutators.

A two color (white and black), incremental, tracing algorithm can be achieved by trapping accesses to both white and black objects. Notice that, in a persistent store, trapping accesses to objects means trapping object mapping and unmapping, which is already trapped for pointer swizzling and other housekeeping operations. After fetching objects from disk, mutators can compute freely. Thus, the overhead in trapping accesses to black and white objects is minimum.

In our algorithm, as traditionally, white objects are potentially garbage objects and black objects are live objects. However, a black object may have both black and white descendents. An object becomes black after being traversed (in mutual exclusion) and its references inserted in a table. Such tables are called Mutator Roots (*MR*), and are also considered to be part of the garbage collection roots. Collector traverse object graphs starting from the roots, and promotes visited objects to black. If a mutator reads an object (either black or white), its references are inserted in MR tables. These tables are maintained permanently updated by mutators. From their point of view, their is no "collection" phase, and the RTS always behave similarly.

The invariant of the algorithm is that, all references to the persistent store maintained by mutators are registered in MR tables.

3.1 Overview of the Algorithm

We first describe the name space where collection occurs, then we explain how a non-stop-the-world algorithm is achieved, and then how mark and sweep phases are decentralized.

Name Space

System wide garbage collection follow LLIs, and sees no other level of identifiers. As presented in figure 3, LLIs may be found in the Persistent Store, in Mutator's address space or in transit between Mutators.

Within the Persistent Store, LLIs may be found in the Eternal Root tables, within clusters outgoing tables and within MR tables. Each storage server holds a private local Eternal Root table, and the set of all local tables constitute the Eternal Root. The same occurs with the MR tables. Clusters are not replicated among the storage servers, thus although the object graphs may span remote machines, a cluster is local to a single storage server.

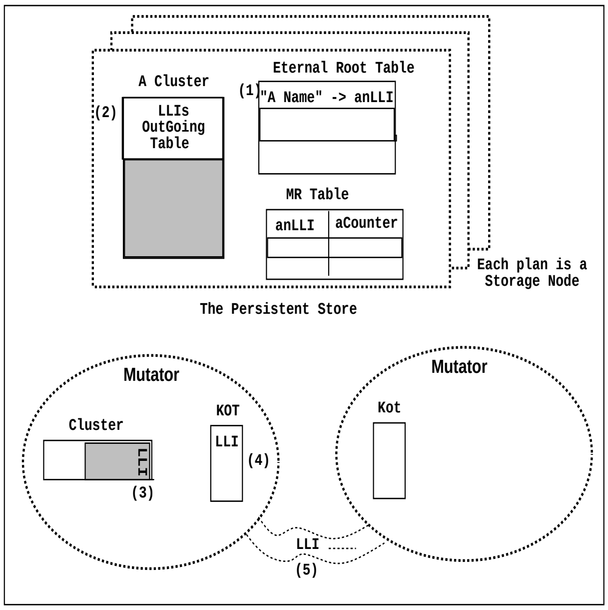

Figure 3: Visibility of objects when mapped in applications and in the persistent store.

Within mutators address space, LLIs can be found either in the KOT table or in outgoing table of clusters mapped. LLIs are registered in the KOT as references are being swizzled from LLIs to local pointers (i.e. as objects are being invoked). Thus the KOT only holds the LLIs actually seen by the application.

Finally, LLIs can be fond in the "on-the-wire" during remote invocations. Although the KOT of sending application holds a copy of the LLIs sent, "on-the-wire" LLIs may be unique replicas if the sending application crashes while the LLI is in transit.

Concurrency with Mutators

Storage servers recycle persistent object without traversing the graphs mapped in mutators' address spaces. This is done by keeping a record of the references that mutators have to the persistent store, and treating them as roots of the garbage collection. There is one MR table for each storage server, containing the references that running mutators have the objects stored in that server.

Each entry in the MR table has a counter that indicates the maximum number of running mutators that currently hold copies of that reference. Such counters are incremented with at-least-once messages and decremented with at-most-once messages, both messages sent by the RTS when LLIs are inserted to or removed from the KOT tables of each mutator. Due to messages reordering in the network or batching in the RTS, the counters may be temporarily wrong[3]

There are only two basic situations in which mutators can acquire a reference into the persistent store: by receiving a reference, either from the persistent store or from another mutator, or by storing a newly created object and keeping the reference in memory. Both cases are trapped by the RTS, which increments the counters of the storage server bound to the reference[4]. When and object is stored, the corresponding MR entry is also incremented.

Although the mutators and storage servers may access clusters concurrently, no synchronization is required other than simple mutual exclusion accesses. Since storage servers operate on disjoint sets of object graphs, the only possible interference occurs between an storage server and the mutators using the objects kept in that storage server. No unique references can exist on clusters written from application to the persistent store, since they would have been trapped and inserted in the MR tables. The clusters written may indeed contain unique references to objects in memory, never stored persistently. However, such objects are not seen by the system wide garbage collector.

If a mutator crashes or messages are lost or duplicated, the corresponding entries in the MR tables will not be decremented, and would remain their permanently. To prevent against such cases, the entries of MR tables are time bounded, and discarded by timeout. Long lived applications must refresh their entries periodically.

[3]The time that counters may be inconsistent is bounded. Since the counters are only read at beginning of a global collection, it must be less than the fastest global collection.

[4]It is not relevant here the association between LLIs and Storage Servers. See [2] for further details.

Distributed Marking

The distribution of the algorithm is similar to the presented by Hudak [13], but with no real-time constrains. It uses a master process only for the purpose of logical rendevouz between the phases of the collector. Each phase is initiated by the master by sending a "at-least-once" *start* messages to all storage servers. The master process recognizes the end of a phase upon the reception of "at-least-once" *end-phase* messages from the storage servers.

The algorithm has three phases: *Init*, *Mark* and *Sweep*. In the *Init* phase, storage servers clear entries in their local MR table whose counters remained zero since the last *Init* phase.

In the *Mark* phase, each storage server promotes white objects to black, begun with the eternal roots and MR entries found locally. When the graph spreads to other storage servers, propagate messages are sent with "at-least-once" semantics, indicating that particular entry points are non-garbage. Marking then continues from such points on the remote node and may lead to further propagate messages. These messages are reported back with reply messages (also "at-least-once") when the corresponding entry points have been fully traversed.

When all propagate messages sent during the traversal of the local eternal roots have been reported back, the storage server send an end-phase message to the master. After sending the end-phase message, storages server may still have a lot propagate messages to proceed with. All marking will be finish when the master receive end-phase messages from all storage servers.

Finally, in the *Sweep* phase, storage servers delete all white clusters found locally.

In algorithms' simpler form, the three phases process sequentially, and the *Init* phase follows the *Sweep* phase of the previous collection. By changing the encoding of white and black tags, the *Sweep* phase of one collection can be done during the *Mark* phase of the next collection, or even be a continuous task, independently of the *Mark* phases [14].

Conclusions

Although with no practical results yet, we believe our approach is suitable for recycling persistent objects in distributed object oriented systems. We tried to minimize the pauses in running applications, and prefer to delay the recycling of garbage produced by an application until it completes execution. This seems appropriate for most cases when applications are short lived, but long lived applications must use other protocols to clean entries in MR tables.

References

[1] José Alves Marques and Paulo Guedes. Extending the Operating System to Support and Object-Oriented Environment. In *Proc. of the OOPSLA 89*, New Orleans, USA, 2-6th Octer 1989.

[2] Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, Paulo Guedes, and José Alves Marques. Distribution and Persistence in the ik Platform: Overview and Evaluation. *Usenix Computing Systems*, 6(4), Fall 1993. To appear.

[3] Manuel Sequeira and José Alves Marques. Can C++ be Used for Programming Distributed and Persistent Objects? In *Proc. of the Int. Workshop on Object Orientation in Operating Systems - IEEE*, Palo Alto, USA, October 1991.

[4] Erich Gamma, André Weinand, and Rudolf Marty. ET++ - An Object-Oriented Application Framework in C++. In *EUUG*, Cascais, Portugal, October 1988.

[5] José Alves Marques, Roland Balter, Vinny Cahill, Paulo Guedes, Neville Harris, Chris Horn, Sacha Krakoviak, Andre Kramer, John Slattery, and Gerard Vandôme. Implementing the COMANDOS Architecture. In *Proc. of Esprit Technical Week*, Brussels, Belgium, November 1988. North-Holland.

[6] Miguel Castro, Nuno Neves, Pedro Trancoso, and Pedro Sousa. MIKE: a Distributed Object-oriented Programming Platform on top of the Mach Micro-Kernel. In *Proceedings of the USENIX Mach Conference*, Santa Fé, USA, April 1993.

[7] Paulo Ferreira. Reclaiming Storage in an Object Oriented Platform Supporting Extended C++ and Objective-C Applications. In *Proc. of the Int. Workshop on Object Orientation in Operating Systems - IEEE*, Palo Alto, USA, October 1991.

[8] Peter W. Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures.* PhD thesis, University of Cambridge Computer Laboratory, 1992.

[9] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Symp. on Principles of Distributed Computing*, pages 135–146, Vancouver (Canada), August 1992. ACM.

[10] H. Lieberman and C. Hewitt. A real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(2):419–429, June 1983.

[11] Barbara Liskov and Rivka Ladin. Highly-avai;ab;e Distributed Services and Fault-tolerant Distributed Garbage Collection. In *Proceedings of the 5th annual ACM Symposium on Principles of Distributed Computing*, pages 29–39, Vancouver (Canada), August 1986. ACM.

[12] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of 1992 International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lectures Notes on Computer Science.

[13] Paul Hudak and Robert M. Keller. Garbage Collection and Task Deletion in Distributed Applicative Processing Systems. pages 168–178, 1982.

[14] Niels Christian Juul. *Compreensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System Emerald.* PhD thesis, University of Copenhagen, February 1993.