Distributed Garbage Collection in the System is Good

David Plainfosse, Marc Shapiro

Distributed Garbage Collection in the System is Good

David Plainfossé Marc Shapiro

INRIA, BP 105, 78153 Rocquencourt Cédex, France

Abstract

We present a protocol for the distributed detection of garbage in a distributed system subject to common failures such as lost and duplicated messages, network partition, dismounted disks, and process, site and disk crashes. The protocol uses only information local to each site, or exchanged between pairs of sites; no global mechanism is necessary. Since overhead is low, the protocol can be integrated into the operating system. We show than a logical separation of responsibility allows to integrate in the operating system the detection rôle leaving up to the application the collection rôle.

1 Introduction

Recent development of the object-oriented technology has sparked interest in low-level support systems for user-defined objects. A number of operating systems [4, 7] and database systems [9] offer such support.

Until recently, garbage collection has been often judged too language-dependent, too complex and too costly for general-purpose systems. In contrast, we think that operating systems should be designed and implemented to offer support for garbage collection. Our approach is to provide a generic service for distributed garbage detection, building upon existing, language-dependent, local garbage collectors.

One important aspect of objects is that one may contain *references* to other objects. A program's activity creates objects and modifies the references between them; an object for which no reference remains has become inaccessible *garbage* and could be de-allocated. Manual deallocation is notoriously error-prone. Automatic garbage collection (GC) is a valuable service as it frees programmer resources and is safer than manual collection.

Moreover GC must be provided for persistent storage. Without garbage collection, persistent garbage objects are not collected and eventually fill the storage entirely. A stable storage server benefits from garbage collection in two ways: GC saves disk space, and it reduces traffic at commit time.

In distributed object oriented systems, objects are spread among *spaces*. Objects localized in separate spaces can reference each other. Our distributed garbage protocol traces those remote references to propagate object accessibility between spaces. Distributed GC is harder than local GC because the local collectors must be coordinated, to keep track of changing references between spaces. The problem is further complicated by considering the common failures of distributed systems such as lost, duplicated, and late messages, and crashes of individual spaces. Finally, a realistic distributed protocol must scale to any number of network nodes.

One can consider that GC is not an OS problem and should be implemented at the language or application level. In contrast, we divide our distributed GC into two layers: a set of local GCs (LGC), knowing about objects' structure and/or semantics; and a generic, OS-supplied, distributed garbage detector (DGD). The integration of the GC into OS simplifies OS mechanisms such as object migration. In particular, migrating objects are still accessible during transit. For instance, a port can be migrated whille receiving messages. Such messages will be forwarded granted to GC informations.

In contrast to previous proposals[10, 5], our DGD protocol is based on reasonable, weak assumptions. It will scale to any number of nodes. It continues to function correctly in the presence of lost, duplicated, or out-of-order messages, or of node crashes (in a fail-stop manner); it allows objects to migrate or become deleted while referenced. Both transient and persistent objects or spaces are supported.

The rest of the paper proceeds as follows. First section 2 discusses the principles of garbage detection and collection, and previous work on distributed GC. Then the section 3 describes briefly the interactions between each protocol's layers. The following section 4 presents the protocol itself and some specific subprotocols. In Section 5 we develop the fault-tolerance aspects. Fi-

2 Principles of distributed garbage detection, and previous work

The purpose of garbage detection is to distinguish objects accessible from a so-called *root*, from others which are called *garbage*. Classically one distinguishes two rôles: the *mutator* and the *collector* [2]. Dividing the work of the collector into two parts, garbage detection and garbage disposal, we will speak of the *detector* rôle.

There are two well-known families of garbage collection algorithms. Reference counting algorithms attach a counter to each object, maintaining the (very strong) invariant that the value of the counter is at all time equal to the number of references to the object. This is inherently hard in a distributed environment, especially with failures. We will not consider reference counting algorithms any further. Note however that in the case where messages are reliable, an ODT entry (defined later) degenerates to a local reference count.

In tracing, the detector performs a walk of the "refers to" graph, starting from the root; at the end of the graph traversal, any objects not reached are garbage.

In Vestal's [10] tracing algorithm, the universe is divided in "areas", in which parallel collection may occur. It is characterized by a high space overhead, and does not take advantage of locality: each collector performs a global transitive closure starting at the root of one area.

Hughes [5] proposes a garbage collector for a distributed-memory multiprocessor with a single clock. Each area has its own local root; a detector repeatedly starts at each local root. The simplicity of Hughes' algorithm is quite appealing : each globally shared objects is timestamped with the latest time at which it was inaccessible from some space. The algorithm is based on the premise that timestamps of accessible objects continue to increase while those of garbage objects eventually become constant. An object is identified as garbage when its timestamps is lower than a global lower bound. However determining this global lower bound requires a termination algorithm.

Liskov and Ladin [6], describe a fault tolerant distributed garbage detector based on a highly available service, which is logically centralized but physically replicated. Nodes may crash (fail-stop) and recover, messages may be lost or delivered out of order. All objects and tables are assumed backed up in stable storage. Clocks are synchronized, and message delivery delay is bounded. The distributed garbage detector relies on local tracing garbage collectors informing the centralized service about its references to remote objects. Local collectors query the centralized service about the real accessibility of their public objects to better estimate their root.

Dead inter-site cycles are detected by the centralized service. Based on the paths transmitted, the centralized service builds the graph of inter-site references, and detects dead cycles with a standard GC algorithm.

3 Separation of Responsibility

The application (mutator) rests upon two separate layers of object management (see Figure 1). The bottom layer is independent of object semantics, structure, or programming language: this is the distributed garbage detector (DGD) specified in [8] and described briefly here. The DGD only propagates accessibility information supplied by the upper layer.

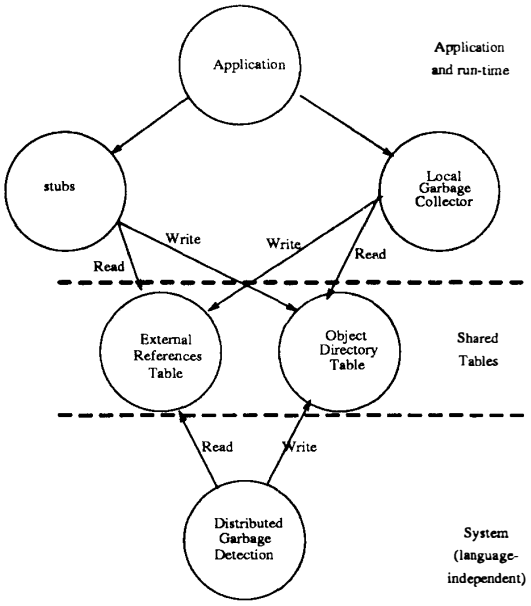

Figure 1: System Layering

The upper layer is a standard (language-specific) run-time, extended to interface with our DGD. In the upper layer one finds storage management (object allocation and local tracing garbage collection) as well as remote invocation functions (communication stubs).

The two layers share information in the form of tables of incoming and outgoing references, respectively

called the ODT and ERT (see below). The cooperation between layers is limited to simple interactions to maintain these tables. In particular, no extra system calls are necessary.

Mutators in different spaces communicate by RPC-style invocation (i.e. by messages). An invocation is mediated by mechanically-generated stubs for marshalling and unmarshalling messages; a stub interfaces between the application and the system, by encoding type information into a typeless form. The arguments to an invocation or return contain any mixture of pure data, references, and migrating objects. When sending or receiving a message, the stub first writes information about the remote references or objects into tables, which are shared between the LGC and the DGD layers.

To provide fault tolerance, extra time and ownership information is piggy-backed onto the existing mutator messages, and occasional control messages are exchanged to inform LGDs of inaccessible objects.

4 Distributed Garbage Detection Mechanisms

The detection protocol is based on local garbage collectors of the marking family, and interaction of a number of sub-protocols: the finder protocol, a reference-sending protocol (omitted here), an object-migration protocol, an interaction protocol between local collectors, and a cycle detection protocol.

Each disjoint *space* maintains a list of potential incoming and outgoing references, called respectively the Object Directory Table (ODT) and External Reference Table (ERT). Both the ODT and the ERT are conservative estimates. If two different spaces possibly refer to a single object of space A, each will be assigned its own ODT entry in ODT_A. This differs from reference counting, because we need an entry per remote space to deal with unreliable communication.

Local garbage collection proceeds from the union of the local root and the ODT and remove entries in the ERT, which in turn allows previously-pointed-to ODTs to be collected. Since local GC starts from the union of the local root with the (conservatively estimated) ODT, all non-reachable local objects are true garbage. Each local GC cleans the ERT of useless stubs. In turn, ERTs are used to clean the ODTs, yielding successively better estimates.

To sum up, global garbage detection occurs by pairwise cooperation between spaces. Local GC conservatively updates the ERT; each such ERT is in turn used to conservatively create new versions of the ODTs. Local GC contributes to clean up the local ERT, and hence remote ODTs.

This discussion shows that the garbage detection algorithm sketched above is indeed correct (a conservative estimate of live objects is maintained at all times) and does eventually find some garbage. However, as we will see next, it does not detect inter-space cycles of garbage.

4.1 Finding an object

To use an object reference it is necessary to track the corresponding object's location. This function is performed by a system component called the "finder" by Fowler [3]. The finder's operation includes "path compression", i.e. the elimination of chains of indirect references, via multiple ERT/ODTs. This s beneficial to both the mutator's and the garbage detector's performance. Conversely, the finder benefits the collection of garbage location information.

Object deletion and migration, and space crashes, are handled transparently by the object finder.

Attempting to find an object possibly located in a disconnected space may be delayed until that space either reconnects or terminates.

Object finding is accelerated by using location information maintained in the ERTs. Finding an existing object, based on a reference, succeeds even if the available information is stale, because if an object has migrated, a forwarding information remains behind. In turn, up-to-date location information from the finder is used to update the information in ERTs, making operation of the detectors more efficient.

4.2 Object Migration

The migration of objects is correctly suported by the our protocol, accessibility of migrating object is enforced by updating ODTs and ERTs in a conservative way. Precisely, whenever an object is about to migrate, a potential reference is created toward the target space. Thus any local reference to the migrated object remains consistent.

For instance, consider some object x which migrates from space A to B. There may exist references toward x (in space A, or from any other space into A). Therefore, we consider a potential reference is created from A to x in space B. *Before* transmitting the message containing x from A, we have to create an entry in ERT_A for that object.

When the message is received by B, a potential reference exists from A to B; create an entry in ODT_B,

before delivering the message to the mutator.

That object x may itself contain a reference to an other object y. Then, in addition to the migration protocol above, we execute the normal procedure for transmitting @y from A to B. If any indirections form, they will be eliminated by the finder.

4.3 Removing Inter-spaces Cycles of Garbage

Since our distributed protocol is based on reference counting, it fails to collect cycles. A separate subprotocol deals with inter-space cycles of garbage. A simple solution [1] is to migrate the cycle to a single space, where it will be collected by the normal operation of local GC. Moving objects converts a inter-spaces cycle to an intra-space cycle which will be collected as soon as the space is garbage collect. This has the desirable property of improving locality. However, this method may not be suitable for distributed system. Moving an object can be expensive, especially if spaces belongs to an heterogeneous net. Since some objects cannot migrate, not all dead cycles are guaranteed to be collected.

An alternative is Hughes' algorithm (described in 2). Nevertheless, the collection of distributed cycles is completely disabled during the time that even a single processor is disconnected (i.e. communication is impossible with it). To compensate for each alternative's drawbacks, in our implementations we plan to combine them both. If the migration strategy failed to collect inter-spaces cycles, we will use Hughes algorithm.

5 Fault Tolerance

Our DGD protocol is fault-tolerant : precisely, space crashes or communication failures do not lead to an inconsistent state w.r.t our DGD protocol. In particular, the liveness and the safety properties remain effective in spite of unreliable communication. Our fault-tolerant DGD protocol, however does not make applications fault-tolerant. A fault-tolerant application benefits from its fault-tolerant aspect : it preserves accessibility to remote objects.

We first describes how our DGD deals with communication failures : duplicated, lost, out-of-order messages. Then we consider space crashes and their consequences on our protocol.

5.1 Communication Failures

Duplication of a message causes no problem w.r.t our DGD protocol. Its only effect will be to redo the same action twice. Garbage detection protocol actions are idempotent.

In contrast, lost messages is a real issue. An application message can be lost forever (i.e it never reaches its target space) due for instance to a networking partition. To address this issue, our protocol requires that any reference sends to a remote space is first registered in our ODT. This earlier registration in ODT may generates table inconsistency : an ODT entry without any corresponding entry in an ERT. In order to maintain the the liveness property of our DGD protocol, spaces exchange periodically background messages (ERT messages) containing a subset of the ERT table. Each subset is composed of all stub referencing the same remote space. But to keep the scalability property of the DGD protocol, ERT messages are not broadcasted. In contrast, they are only send to referenced spaces. In other words, a space A will only send ERT messages to those spaces which, either currently appear in ERT_A, or were recently removed from it. This is not enough, since after the last reference to some space B is removed, B might never receive ERT messages from A and hence never collect the corresponding ODT entry. To deal with this tricky case, when a space suspects to hold an useless ODT entry it asks for an ERT message to the corresponding remote space.

The assumption of instantaneous message transmission allows a total event ordering. When some space A receives a message from space B all messages sent in the opposite way, from A to B, have been either received or are lost. When message transmission takes no zero time, as in real distributed systems, this total ordering is lost. The delivery delay must be taken into account in a distributed GC. The problem is quite simple : whenever a message containing a reference (creation message) is concurrent to an ERT message, the order of delivery is not equivalent. If the ERT message is delivered whereas the creation message is still in transit, the ERT message must be ignored. In order to detect messages in transit we timestamp events at a space from a local, monotonically increasing "clock". Each transmitted messages is stamped with the value of the clock on transmission. Furthermore each ODT entry is stamped with the clock value of the last corresponding message sent.

Finally, each stub keeps the the vector of highest timestamps (HTS) received from each other space. Moreover, ERT messages carry HTS value correspond-

ing to the target space. Thus, ERT messages are considered valid if HTS value is bigger or equal to ODT's timestamp. In other words, if no messages are in transit between these two spaces.

Our use of timestamps to guard against possibly in-transit references works well if communication are FIFO : if messages are received in the order they are sent. With a small extension, our protocol can tolerate some amount of non-FIFO ordering. The only use of FIFO ordering is the rejection criterion for removal messages in case a reference is in transit. An out-of-order message could invalidate this criterion.

5.2 Space Crashes

We assume crashes are fail-stop, therefore the only consequence of a crash is temporary disconnection, loss of volatile memory and halt of computation. Objects and references stored only in volatile memory disappear; objects and references in non-volatile memory persists across crashes and become active again when space recovers and reconnects. The problem is to ensure that ERTs and ODTs remain consistent through the crash and recovery, that garbage detection continues to behave correctly and that no object, which is reachable after the recovery are incorrectly believed to be garbage during the crash.

When a persistent space recovers, its ODT persists and continues to contain a superset of its remotely reachable objects. Unused ERT entries will be recovered by the next of the local garbage collector. Therefore the DGD does continue to detect garbage.

When a volatile space crashes, all information about the objects that it contains is lost, in particular its ODT and its ERT. Moreover indirections crossing this space are lost and can lead to reclaim non-garbage objects. To solve this case we plan to keep ODT and ERT in stable storage. But this is clearly too expensive in the common case. Therefore we are considering an hierarchal design for the Soul implementation. We distinguish three levels of reliability. At the bottom level each address space map a space. Communication are assumed to be reliable and instantaneous. At the middle level, each processor in a local network is a space. Communication is unreliable and non-instantaneous, but the number and names of spaces are static and well-known. At this level, exhaustive search of objects is acceptable. Communication between networks is managed by "gateway" processors. The ODT of a gateway is modified atomically and backed in stable storage. The third level is the long-haul net. Indirections are eliminated aggressively at this level; a reference is not delivered to the application until all indirections have been eliminated.

6 Conclusion

In recent years, a new organization has emerged for operating systems, in which a "micro-kernel" provides only the basic functions of memory management, process scheduling and communications. More elaborate services are implemented as specialized servers.

We think distributed GC should be considered as a valuable basic functions of systems: GC simplifies programming, and it simplifies system design.

But it is not straightforward to perform efficient GC in a distributed environment. A good candidate must fulfill at least this set of requirements:

- The algorithm should execute in parallel on many spaces.
- The algorithm should tolerate space crashes and unreliable communication. In particular, if the GC algorithm required global synchronization, it would stop if some space becomes unreachable.
- The algorithm must scale to large architectures.

Our protocol fulfills these requirements. It is based on any standard local tracing garbage collector. It is simple and deals gracefully with common error occurrences of real distributed systems.

This protocol has still to be tested in practice. Two implementations are under way. The first one prototypes the protocol on a multiprocessor Lisp system. We have currently achieved the implementation itself and we begin to fake failures and make performances measurements. In addition to the protocol described above, this implementation deals with object replication. The second is for the multiprocessor object-support database system EOS, being specified at INRIA project Sabre.

References

[1] P. B. Bishop. Computer systems with a very large address space, and garbage collection. Technical Report MIT/LCS/TR-178, Mass. Institute of Technology, MIT Laboratory for Computer Science, Cambridge MA (USA), May 1977.

[2] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[3] Robert Joseph Fowler. Decentralized object finding using forwarding addresses. Technical Report 85-12-1, Department of Computer Science, University of Washington, Seattle, WA (USA), December 1985.

[4] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov. COOL: Kernel support for object-oriented environments. In *ECOOP/OOPSLA '90 Conference*, volume 25 of *SIGPLAN Notices*, pages 269–277, Ottawa (Canada), October 1990. ACM.

[5] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985. Springer-Verlag.

[6] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 29–39, Vancouver (Canada), August 1986. ACM.

[7] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[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] Fernando Velez, Guy Bernard, and Vineeta Darnis. The O_2 object manager: an overview. Technical Report 27-89, GIP-Altaïr, Rocquencourt (France), February 1989.

[10] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, Dept. of Comp. Sc., U. of Washington, Seattle WA (USA), January 1987. U. of Washington Tech. Report 87-01-03.