Fault-tolerant reference counting for garbage collection in
distributed systems

L.V. Mancini, S.K. Shrivastava

Abstract

The function of a garbage collector in a computer system is to reclaim
storage that is no longer in use. Developing a garbage collector for a
distributed system composed of autonomous computers (nodes) connected
by a communication network poses a challenging problem: optimizing
performance whilst achieving fault-tolerance. The paper presents the
design and implementation of a reference-count garbage collection
scheme which is both efficient and fault-tolerant. A distributed object-
based system is considered where operations on remote objects are
invoked via remote procedure calls. The orphan treatment scheme
associated with remote procedure calls has been enhanced to enable the
collection of garbage arising from node crashes.
Bibliographical details

MANCINI, Luigi Vincenzo


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1988.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 260.)

Added entries

SHRVASTAVA, S.K.
UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Laboratory. Technical Report Series. 260

Abstract

The function of a garbage collector in a computer system is to reclaim storage that is no longer in use. Developing a garbage collector for a distributed system composed of autonomous computers (nodes) connected by a communication network poses a challenging problem: optimizing performance whilst achieving fault-tolerance. The paper presents the design and implementation of a reference-count garbage collection scheme which is both efficient and fault-tolerant. A distributed object-based system is considered where operations on remote objects are invoked via remote procedure calls. The orphan treatment scheme associated with remote procedure calls has been enhanced to enable the collection of garbage arising from node crashes.

About the authors

Mr. Mancini has been at the Computing Laboratory since May, 1985 as a Research Associate.

Professor Shrivastava joined the Computing Laboratory in August, 1975, where he is a Professor.

Suggested keywords

DISTRIBUTED SYSTEMS
FAULT-TOLERANCE
GARBAGE COLLECTION
OBJECT-BASED SYSTEMS
ORPHAN DETECTION
REFERENCE COUNTING
RELIABILITY
REMOTE PROCEDURE CALLS

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 001.6425
U.D.C. 519.687 681.322.06
1. **Introduction**

Many programming systems require an automatic garbage collection facility for storage management. In such systems, objects reside on a heap and are garbage collected after they become inaccessible. In many systems, the heap is kept in primary (volatile) storage, so that a system crash destroys the objects on the heap. For programming systems which permit access to remote objects a distributed garbage collection facility is required. Such systems can be supported by a number of autonomous computers (nodes) connected by a communication network.

This paper describes a cheap and efficient *fault-tolerant distributed garbage collection* scheme based on the well-known reference counting technique. A distributed garbage collection facility should closely approximate the behaviour of its non-distributed counterpart despite the occurrence of failures (such as lost messages and node crashes). We take this to require that if a node containing remote references crashes and therefore leaves garbage on other nodes, then that garbage should eventually be collected. Designing an efficient scheme for garbage collection from primary storage is a sufficiently challenging problem and will be discussed in detail in the paper. Enhancements to cope with objects on secondary storage will also be presented.

The design presented here has several interesting features:

1. It is tolerant to the following types of failures: node crashes (*fail-silent* behaviour will be assumed, that is, a crashed node completely ceases to function), and lost, duplicated, delayed and out-of-order messages.

2. It does not require elaborate facilities such as failure-atomic procedures or synchronized clocks.

3. Individual nodes in the system are free to choose any local garbage collection scheme.

4. The design relies on a close integration with the orphan detection scheme of a remote procedure call mechanism, thus enabling the exploitation of existing fault-tolerance facilities.

The basic idea behind our scheme is quite straightforward. We assume an *object-based* system, where operations on remote objects can be invoked by *remote procedure calls* (RPCs). We require that RPCs provide, even in the presence of failures, the same semantics as local calls. To meet this requirement, we need an orphan detection and killing mechanism. To appreciate this, consider the following situation: a computation running on node B makes an RPC to some object at node A and then node B crashes, leaving an *orphan computation* running on node A. In order to guarantee that the executions for all post-crash calls from node B to node A succeed the executions for all pre-crash calls, it is necessary for node A to detect and abort orphans before executing post-crash calls from node B. Given that each node has an orphan detection facility, it seems natural to embellish it for the garbage detection. Referring to the example just discussed, a crash of node B can leave garbage at node A, which can be detected by node A while detecting orphans. Such an
integrated orphan detection and garbage collection mechanism is the main subject of this paper. Note that since our scheme is based on the reference-count technique, it suffers from the limitation (which is well-known) that it is incapable of collecting objects if inter-node references form a cycle. However, the scheme is capable of collecting cyclic structures which are broken by a crash.

In the next section we briefly introduce the object-based model of computation and the terminology employed in the paper. Section 3 contains a brief review of existing work on distributed garbage collection and section 4 gives an overview of an RPC mechanism with an orphan detection and killing facility designed and built at Newcastle. Section 5 describes the enhancements necessary for our purposes. Conclusions are presented in the last section.

2. Object-based Systems and Reliability Requirements

In recent years a great deal of interest has developed in object-based programming and systems (e.g. C++ [Str86], Emerald [BHJL87], and Smalltalk [GoRo83]). Simply stated, objects are instances of abstract data types, where a type encapsulates some data and provides a set of operations for manipulating those data, these operations being the only means of data manipulation. In a typical implementation, each object is associated with a unique name - a capability - which is used to control access to the object, both in terms of who may use the object and what operations may be performed on it. A capability is context independent in that, regardless of where the capability is stored in the system, it always refers to the same object. To emphasize the distributed nature of the system, we shall refer to capabilities for remote objects as remote capabilities (RCs) and assume the existence of some method for locating objects efficiently, given their RCs. In such a capability-based system, an object without any capabilities for it will be treated as garbage.

In a distributed system, an operation on a remote object is typically performed by invoking the operation of the object via an RPC with the RC for the object as one of the arguments. Below we introduce some terminology, and illustrate it with the help of Figure 1, which shows an object O_k at node B holding an RC for an object O_i at node A (this is indicated by the dashed line). The node where an object is located is called the owner of the object, the object is local to that node. An object will be termed public if its owner has sent its capability to some other node (so O_i is a public object). A local object that is not public will be termed private. We assume that objects are large (hundreds of bytes), and capabilities are relatively small (few bytes). Some mechanism is required for implementing the abstraction of objects holding RCs for other objects (such as O_k holding an RC for O_i). One such mechanism is illustrated in Figure 1. Each node maintains two objects called the export list and the import list. The export list of a node maintains a list of all public objects of that node, whilst the import list maintains all the RCs of that node. Specific details of how objects come to hold RCs for other objects are not directly relevant for our discussions, so will be glossed
over. We will, however, assume that objects are capable of transferring (copying) their RCs to other objects.

A distributed computation is performed by client and server processes. The invocation of an operation on $O_i$ by $O_k$ will be carried out as follows: a client process at node B obtains RC $j$ for $O_i$ from the local import list and sends a call request containing $j$ to a server process at A. The server process at A uses the RC $j$ received to get the address of object $O_i$ from the export list; it then performs the requested operation on $O_i$ and sends the results back to the client. It will be assumed that a server process can be used for serving a sequence of calls from a given client. Servers and clients are created by the RPC mechanism as the need arises. As stated before, it will be assumed that a crash of a node causes volatile objects to be destroyed; in addition a crash also destroys all the processes of that node. A node can also own stable objects which are not destroyed by a crash. There are thus three possible types of object-based systems from the point of view of fault-tolerance:

1. All objects are volatile (temporary) and are lost with crashes. In such a system, if node B crashes then $O_k$, the client process, the export and import lists of B and therefore the RC for $O_i$ at node B vanish and the server on A will become an orphan. Assuming that only $O_k$ held an RC for $O_i$, then $O_i$ will become garbage. It will be assumed that the lifetimes of volatile objects do not exceed that of the computations which created them; thus, a volatile object with RCs will always have one or more server processes associated with it.
2. All objects are stable (persistent): objects, including import and export lists, survive crashes. The lifetime of stable objects can exceed the lifetime of the computations which manipulate them. In this case, \( O_k \), the import list and therefore the RC for \( O_i \) survive a crash of B. It is worth while to note that such a crash will cause the server process to become an orphan, but \( O_i \) will not become garbage. A distributed system with stable objects will typically need to structure its computations as atomic transactions [Gra86] in order to maintain consistency. However, such a provision is orthogonal to our garbage collection scheme.

3. A subset of the objects is stable and the remaining part is volatile. Naturally, only the volatile objects of a node will vanish because of a crash with the possibility of creating garbage on other nodes.

An RC will be called stable if it is held by a stable object, and volatile if it is held by a volatile object. A crash of a node can create two kinds of undesirable situations: (i) some remote public objects become garbage, and (ii) some RCs (on remote nodes) point to non-existing objects destroyed in the crash. We have taken the view that a fault-tolerant garbage collector should be capable of dealing with the first situation. Consider the system shown in Figure 2. A crash of node D will cause \( O_k \) and \( O_j \) to hold RCs for \( O_h \) which no longer exists. As such, an invocation of some operation on \( O_h \) by \( O_k \) or \( O_j \) may well cause a run-time exception whose treatment, in our view, should be orthogonal to the functioning of the garbage collection system. However, suppose that \( O_k \) deletes its RC for \( O_i \) at node A and then a crash of node C occurs; in this case, \( O_i \) becomes garbage and must be reclaimed by the garbage collection system.

The requirements that a distributed garbage collector for an object-based system should meet are given below.

1. The collection method should be capable of handling both volatile and stable objects of varying size.

2. The method should be applicable to both real-time and interactive programming environments. For example, a method which required stopping all ongoing computations in the entire system while performing garbage collection would be unacceptable.

3. The method should be fault-tolerant. In a distributed system, part of the system can fail while other parts still function. This behaviour imposes two reliability-related requirements on garbage collection. First, collection of garbage created by node crashes should be guaranteed. Second, the collection mechanism should be able to cope with the transfer of RCs among nodes in the presence of failures (see Section 5.2).

4. In most distributed systems, sending a message from one process to some remote process is a relatively slow operation (consuming anything from a few to several milliseconds of
time), so the garbage collection method should strive to minimize network communication requirements.

The distributed garbage collection method presented here is independent of the particular local garbage collection technique in use at individual nodes. Relevant information about inter-node references is stored at each node using a technique based on the reference-count method. A reference-count service can be designed using the orphan detection scheme discussed in Section 4.

3. Notes on Distributed Garbage Collection and Correctness Requirements

The function of a garbage collection scheme is to automatically reclaim storage that is no longer in use by computations. This automatic collection of storage frees the programmer from dealing with the complexity of dynamically determining which objects are needed and which are not at any particular time. Storage for objects is allocated from a heap. In simple systems the heap is kept in the primary store, so objects are volatile. An object is defined to be accessible if it is reachable from a fixed object called the root.

The two main garbage collection schemes are (1) mark-scan, and (2) reference-count.

1. A great majority of garbage collectors for non-distributed systems employ the mark-scan method [Knu72]. Mark-scan garbage collection needs to be invoked only when there is no free storage available; otherwise it imposes no performance penalty. When the collector is
invoked, all other computations are stopped and storage for objects that are not accessible is collected for reuse. Starting from the root, the first phase (mark) causes all references to be traced and every object actually in use to be marked. The scan phase examines the mark on every object; unmarked objects are free and their storage spaces are collected together for reuse.

A major objection to the mark-scan technique is that all of the ongoing computations must be halted when the collector is invoked. This has the effect of making an application suddenly unresponsive while the collection is taking place. Such unpredictable and often lengthy interruptions are unacceptable in real time applications. In a distributed system the problem is even more serious since work on all nodes must be halted for a global search to take place when any one processor runs out of memory storage. Another disadvantage is that all objects must be scanned (no matter how many are free), so the cost of this technique is proportional to the total number of objects in the system.

A number of proposals have been made to circumvent these problems. Although versions of mark-scan have been developed which operate in parallel with normal processing [DLMS78], the garbage collection is still global in the sense that the entire system needs to be searched.

2. A simple way to automatically collect unused storage is to associate with each object a reference-count field recording the number of references to that object. The reference-count is incremented each time a new reference is created by an object and decremented each time an old reference is removed by an object. When the count falls to zero, no references remain and the storage block can be deallocated [Coh81].

Reference counting, unlike mark-scan, does not require that application processes be halted during collection. The overhead due to the algorithm is spread among all objects, making this technique suitable for real-time and interactive programming environments. Moreover, reference counting is localized, an object is responsible for updating the reference-counts of only those objects it refers to. So this method appears suitable for implementing garbage collection in a distributed system. The major objection raised to this scheme is that it cannot collect cyclic linked data structures. An unused cyclic list will not be reclaimed - each individual cell in the list will have a non zero reference-count, although the list as a whole is no longer needed. Several algorithms to solve this problem whilst retaining most of the advantages of reference-count over mark-scan garbage collection have been proposed [Bob80, Bro85, Ves87].

Garbage collection of a single storage heap has been widely discussed for many years; here we are concerned with garbage collection in distributed, unreliable systems. In such systems, "the heap" turns out to be distributed among the nodes of the system. Such a distributed heap can be viewed as a heap whose root is distributed and consists of the union of the roots at all nodes. In such an
environment, an object is accessible if it is accessible from one of the roots. Several algorithms to perform distributed garbage collection have been published recently.

Hudak's collection scheme is based on performing a global mark-scan collection beginning at a unique, system-wide root object [HuKe82]. Each object, beginning with the root, first checks if it has been marked. If not, it marks itself, sends messages to each object that it references, and awaits replies from all these objects. This may be viewed as each object containing a mark procedure that recursively calls the mark procedures of all objects reachable from it. The collection terminates when the root procedure returns.

Ali describes a number of algorithms for use in a distributed system [Ali84]. The most advanced of his algorithms adopts a copying technique [Bak78], and does not require any sort of synchronized global collection - a collector only examines a portion of the total space each time. This technique also permits the collector to perform in parallel with other processes. However, his method cannot collect cycles that span more than one node.

None of the methods discussed so far have addressed the problem of fault-tolerant garbage collection in distributed systems. This topic, although important, has not received much attention. We are only aware of [LiLa86, Ves87] which addresses this issue.

The scheme presented in [LiLa86] exploits a reliable central service to store information about inter-node references. The nodes communicate with the central service periodically, to inform it about their references to objects at other sites, and to inquire about the accessibility of any local objects that might be referred to at other sites. This approach requires the central service to use a large amount of storage to record the map of the distributed heap - in the worst case such a storage might be as large as the whole distributed heap.

In [Ves87], two fault-tolerant garbage collection algorithms for object-based distributed systems are presented. The first algorithm, which is the closest to our approach, combines reference-count with an algorithm to collect circular object structures. Vestal's solution maintains a separate reference-count, called local reference-count, in every node that contains any capabilities for a given object. The object itself contains a list of nodes that have local reference-counts for it. The actual reference-count is obtained by an object by summing all the local reference-counts. These local reference-counts will continually experience creation, change, and deletion during the operation of the system. The problem then arises of computing a single global reference-count for an object in parallel with other processes. A solution is proposed requiring the synchronization of the physical clocks and the execution of certain procedures atomically with respect to failures. The failure atomicity property is also exploited to guarantee reliable copy of a remote reference among nodes. The second of Vestal's algorithm uses a parallel mark-scan collector based on the algorithm presented in [DLMS78]. It resembles the solution in [Ali84], but can collect cycles that span more than one node.
The scheme presented by Liskov and Ladin is different from ours in that it employs a centralised (replicated) service for recording object references whereas our scheme does not employ such a service. Compared to the Vestal's approach, our solution provides broadly similar functionality without requiring synchronized clock or special failure atomic procedures.

There are two correctness requirements for a reference counting garbage collector:

SAF  if the reference-count of an object is zero, then there are no capabilities for that object;

LIV  if there are $n$ ($n \geq 0$) capabilities in the system for an object, then that object will eventually have its reference-count equal to $n$.

Bearing in mind that in a reference counting scheme an object is collected if and only if its reference-count is zero, these two requirements can be seen as the statements of safety and liveness properties. The first requirement, SAF, states the safety property that nothing bad happens (viz. objects with capabilities for them do not get collected), but it does not ensure that something good happens: the garbage collector might never collect any objects and still satisfy SAF. The liveness property LIV is therefore needed to guarantee that actual progress does take place. The liveness property requires the updating of reference-counts.

4. RPCs and Orphan Detection and Killing

Orphans are unwanted executions that can manifest themselves due to communication or node failures [Nel81]. We will say that a call terminates abnormally if the termination occurs because no reply message is received from the called server. Network protocols typically employ timeouts to prevent a process waiting for a message from being held up indefinitely. Assume that a client process waiting for results from the called server has some such protocol dependent mechanism which notifies the client if no reply is received after some duration. If the call terminates abnormally then there are four mutually exclusive possibilities to consider: (i) the server did not receive the call message; (ii) the reply message did not reach the client; (iii) the server crashed during the call execution and either has remained crashed or is not resuming the execution after crash recovery; and (iv) the server is still executing the call, in which case the execution could interfere with subsequent activities of the client. Figure 3 depicts a particular example of interference where calls are nested.

We assume an exactly-once semantics for RPCs: a normal termination (the client receives a reply from the called server) implies exactly one execution. An abnormal termination can mean zero, partial or one execution at the called server. Next, we present a simple and intuitively appealing correctness criterion for an RPC implementation. This criterion is motivated by the principle that the semantics of RPCs should be as close as possible to those of local single-machine calls.
Figure 3. Possible interference in a nested call

Let C_i denote a call made by a client and W_i represent the corresponding computation invoked at the called server. Let C_i and C_j be any two calls made by a client such that: (i) C_j happens after C_i (denoted by C_i then C_j); and (ii) computations W_i and W_j share some data such that W_i and/or W_j modify the shared data. Then we say that an RPC implementation should meet the following correctness criterion, in the presence of specified types of failures:

\textbf{CR} \quad C_i \text{ then } C_j \text{ implies } W_i \text{ then } W_j.

The criterion \textbf{CR} states that a sequence of calls at a client should give rise to computations invoked in the same sequence. In the absence of any failures, the synchronous nature of calls guarantees that \textbf{CR} will be satisfied. However, failures can create orphans that do require special measures in order to meet \textbf{CR}. In the presence of node crashes for example, an RPC mechanism ought to guarantee that all the executions of post-crash calls of a node succeed all pre-crash ones. The concurrency such as depicted in Figure 3 should be regarded as undesirable, since it is expected that the execution of a sequential program should give rise to a sequential computation characterized by a single flow of control. What is required, at least, is that node C should abort the orphan before executing the second call from client K. In any large distributed system, communication and node failures can be relatively frequently occurring events, so any well engineered RPC mechanism should strive to meet \textbf{CR}. 
Based on the client-server model briefly described in Section 2, [PaSh88] have recently developed a very efficient scheme for orphan treatment for general purpose RPCs. There are three mechanisms used for treating orphans:

(i) Every call contains a deadline, indicating to the server the maximum time available for execution. If the deadline expires, then the server aborts the execution and the call terminates abnormally. It is worth while to note that if there are no node crashes in the system, then this mechanism will be enough to cope with orphans. The remaining two mechanisms cope with crashes.

(ii) Every node maintains a variable - called the crashcount - which is initialised to the current value of the local stable clock immediately after a node recovers from a crash. A node also maintains a table of crashcount values for clients that have made calls to it. A call request contains the client’s crashcount value - if this value is greater than the one stored in the table at the called server node, then there could be orphans at the server node which are first aborted before proceeding with the call.

(iii) Every node has a terminator process that occasionally checks the crashcount values of other nodes - by sending messages to them and receiving replies from those that are up - and aborts any orphans when it detects any crashes.

These mechanisms have been optimized to provide a very cheap orphan treatment system. In particular, no stable storage is required (other than the stable clock which is available in most computers anyway) and there is no need to keep clocks synchronized. Further, not each and every call has to be checked by a server for crashcount comparison. Finally, the terminator based mechanism has been optimized as follows: a server that has not received calls from a client for a while marks itself as a potential orphan. The terminator need only perform its checks for potential orphans. Lastly, the RPC mechanism copes with message failures (lost, duplicated and delayed messages) by employing well-known protocol related techniques which will not be discussed here.

Techniques for orphan treatment can be expensive, so they have not been integrated into general purpose RPCs (e.g. Courier [Xero81]), although schemes have been developed for systems supporting atomic actions [LSWW87,McHe86]. The method presented in [McHe86] uses a deadline based mechanism to eliminate orphans created by crashes and aborts. The method is based on clocks local to each site and it performs best when clocks are synchronized, although non-synchronized clocks do not produce inconsistencies.

In the subsequent sections of the paper we will describe enhancements made to the mechanisms (ii) and (iii) above to provide garbage collection.
5. Fault-Tolerant Garbage Collection

The main features of a distributed fault-tolerant garbage collection scheme exploiting the above orphan treatment system will be presented in the following sections. In Section 5.1 we present a simple fault-tolerant scheme for volatile objects to be used when transferring of RCs is not permitted. In Section 5.2 the refinements required to cope with RC transfers will be discussed. We shall exploit only the mechanisms (ii) and (iii) of the orphan detection scheme discussed in the previous section, namely every node is required to maintain a crashcount and to run a terminator process occasionally. Note that mechanism (i) is not needed, because we are interested in reclaiming garbage created due to node crashes. Section 5.3 discusses how the scheme can be extended to cope with stable objects.

5.1. Treatment of Node Failures

Nodes are responsible for doing local garbage collection. Only private objects are candidates for garbage collection at a node. Each local garbage collector treats the objects not accessible from the local root as garbage. Since the export list is always accessible from the local root, all the "public" objects not accessible through the export list become private. Therefore the problem of designing a fault-tolerant distributed garbage collector reduces to the design of a protocol to keep the exports lists consistent with the import lists throughout the system. It is worth while to note that public objects may be used locally as well; these objects will be collected only when neither local nor remote capabilities exist for them.

At each node there is a reference-count service, integrated into the RPC mechanism, which is responsible for determining the accessibility of public objects. The reference-count service of a node achieves its aims by updating the export and import lists mentioned earlier. An entry is added to the export list the first time a capability for a local object is sent to another node (i.e. when a private object becomes public). This entry includes a reference-count field indicating the number of objects that hold RCs for this public object. The export list provides the local garbage collector with the information necessary for detecting objects that are no longer public (an object whose reference-count field in the export list reaches zero becomes private and therefore a candidate for garbage collection if no local references exist). The objects listed in the export list may be a superset of those actually used by other nodes. For example, referring to Figure 1, suppose that $O_k$ at node $B$ holds the only RC for $O_l$ at node $A$, and that $O_k$ is deleted at $B$. Object $O_l$ is no longer accessible, yet there will be a positive reference-count in $A$'s export list until some further action is taken at $A$.

During local garbage collection, the collector is required to construct a list -called del- of all the imported RCs deleted, and then update the import list after finishing the local garbage collection. The reference-count service of a node is responsible for distributing the del list to other nodes.
Thus the receiving nodes are provided with the information to update their export list in order to assess the accessibility of their public objects. The del list need not be kept stable since the garbage due to node crashes is detected by the orphan detection mechanism. It is worth noting that the construction of the del list does not require any additional scan of the storage, since the import list can be kept updated by treating its entries as private objects.

Each node does its garbage collection independently of other nodes, using an algorithm of its choice. The algorithm must however be extended slightly to take account of the export, import and del lists.

A data structure referred to as a client list, which is a list of ClientElem (see Figure 4), is

```c
typedef ClientElem = struct {
    Name clientNode  % client node address %
    Real crashCount  % crash count value of the clientNode %
    % list of servers created for the clientNode %
    ServerList serverList
    % list of RCs offered to the clientNode %
    RClIst rcList
};
```

Figure 4. Client list data structure

maintained by the RPC orphan detection scheme at a node and contains information about all the client nodes that have made calls to this particular node [PaSh88]. An entry of type RClIst is required for garbage detection purposes. The rcList field lists the remote capabilities that have been used by the client whose name is recorded in the clientNode. The serverList field contains the names of local servers which have been created for the clientNode. The client list and export list of a node are initialized to be empty at the node startup time.

The protocol followed at each node in order to support the distributed garbage collection service will now be discussed.

When a capability for a local object at some node A is to be exported to some other node B as a result of a call request invoked by a client process at node B, the called server process running on node A performs the following steps:

1. If the export list at node A contains an entry for the capability being exported then its reference-count value is incremented by one, otherwise a new element is added to the export list, with the reference-count field initialized to one;

2. The capability being exported is inserted in the rcList field of the entry for node B in the client list at node A.
Whenever an orphan server is aborted at node A because a crash of node B is detected either by some server at node A or by the terminator of node A (respectively mechanism (ii) and (iii), Section 4), the following steps are performed at node A:

1. The reference-count values of all the RCs recorded in the rcList field of the client list entry for node B are decremented by one in the export list at node A. Entries with reference-count field containing zeros are deleted from the export list thus making the relevant objects private;

2. The entry for node B is removed from the client list. Thus ensuring that the previous step is performed only once.

A node, say B, periodically sends its del list to other nodes. Upon receiving this list every node performs the following operations for each RC in the del list sent by node B:

1. It checks if the RC is in the rcList field of the entry for node B in the client list, and if so,

2. It deletes the RC from rcList and decrements the appropriate reference-count field of the export list by one. If the field is zero then that entry is deleted as stated earlier.

It is worth noting that inaccessible cyclic structures of RCs can be collected if a crash of a node breaks the cycle. In this case orphan servers of that node will be detected on at least one other node forming the cycle causing the storage for the cyclic structure to be reclaimed.

The above mentioned operations represent minor modifications to the existing orphan detection and killing system whose design has been analysed in a formal setting in [PaSh].

It is worth noting that the scheme presented so far ensures SAF and LIV requirements in the absence of RC transfers. SAF, which requires that only those objects for which no RCs exist (viz. those objects with reference-count equal to zero) become private, is ensured because the objects listed in the export list are always a superset of those actually in use by other nodes. If the reference-count of a public object, say $O_i$, is decremented, then this is because either some node holding an RC for $O_i$ crashed causing the RC to vanish or some node sent a del list containing the RC for $O_i$. LIV is ensured in the presence of crashes because orphan servers will eventually be aborted causing the updating of the relevant reference-counts.

Inconsistencies can arise due to crash of nodes during the transfer of RCs. In this case the scheme presented so far does not ensure that only objects without RCs for them will have reference-counts equal to zero. In the following section this and other issues will be discussed.

5.2. Reliable Transfer of Remote Capabilities

One additional mechanism is required to transfer RCs reliably while preserving SAF and LIV. Consider the following example. Node A is the owner of a public object and node B holds an RC for that object. B now transfers this RC to some node C as a result of a request by C. Inconsistencies can arise if B crashes (causing its RC to vanish) after sending its RC to C but before informing A.
In this case SAF may not be satisfied - since the public object owned by A can be garbage collected, leaving C to hold an RC for a non-existing object. In order to satisfy SAF, the RC transfer should only be regarded as completed normally if the export list of A and its client list have been updated properly. Consider then the following protocol. Whenever a server discovers that it is transferring an RC as a part of its RPC reply message, it first makes an RPC (inform-done message pair, see Figure 5 where numbers indicate the sequence in which the messages are sent) to the owner of the relevant object so that the owner can update the export list and make an entry in the client list (for C in this case). Only if this call terminates normally does the server send the reply message transfer with the RC. Referring to the example, if the call by C to acquire the RC from B terminates normally, it is ensured that the export list and client list at A have been updated. Thus the protocol guarantees the SAF requirement.

Let us now consider situations where the LIV requirement can be violated. With reference to Figure 5, suppose that B crashes after informing A but before sending the transfer message to C. In this case LIV may be violated - the object reference-count in A may be higher than the number of RCs in the system. The terminator and potential orphan mechanism mentioned in the previous section can be suitably modified to cope with such situations. The potential orphan mechanism operating at A makes sure that if no calls are received from B or C for a long time, then enquiry messages will be sent to them to detect crashes. This mechanism can be enhanced to take care of unused RCs, i.e. RCs listed in the client list that remain unused for a long time. In the situation presented above, eventually A will send an enquiry message to C and will be able to adjust its relevant entries since C does not hold the RC. It is worth while to note that the protocol discussed
can be seen as a technique for A to cope with crashes of B; crashes of C are dealt with by A in the same manner.

To summarize, nodes are periodically required to exchange three types of information: (i) lists of potential orphans, (ii) lists of unused RCs, and (iii) del lists. The first type of information is required for orphan detection, and the remaining two for garbage detection. A simple optimization is for a node to construct a single message containing all the three components for distribution.

5.3. Treatment of Stable Objects

The scheme presented so far deals with the treatment of volatile objects. In the following it will be discussed how to enhance it first to cope with stable objects (i.e., objects are persistent and survive crashes), and then to cope with the mixed approach where both stable and volatile objects are permitted.

In order to implement the abstraction of a stable heap, all the bookkeeping information about stable objects must also be kept stable, therefore each node must maintain its export, and import lists, and del list on stable storage. Since these data structures are kept stable, node crashes in this case cannot produce garbage on other nodes. The protocols discussed in Section 5.1 need only one modification: the updating of the export list when orphan servers are detected is no longer required - the export list of a node is updated only when a del list is received. However, the mechanism discussed in Section 5.2 for preserving SAF and LIV is still required for transferring RCs between nodes, as the following example illustrates. Suppose the transfer protocol is not employed, then the following situation is possible (Figure 5): B deletes its RC after sending it to C and then crashes before informing A about the transfer. If garbage collection is done at A using post-crash information from B (note that the del lists are kept stable while the information about the RC transfers are not), the object referred by the RC at C might be collected by mistake. An alternative to our solution for solving the above possible inconsistency is to keep a stable log of all in-transit references [LiLa86].

The discussed scheme for the garbage collection of a stable heap will continue to satisfy both SAF and LIV properties. SAF is satisfied since objects are collected only if all their stable RCs have been explicitly deleted. LIV is preserved since an object will be eventually informed about deleted stable RCs because the del lists are kept stable by the ex-holders; thus the object reference-count will eventually become consistent with the number of RCs in the system.

Let us now consider the provision of garbage collection in distributed systems where both volatile and stable objects are supported. In such a system volatile and stable RCs for the same object are permitted (e.g RCs to O; in Figure 2). An example of such an environment could be a network of nodes some of which are diskless workstations. In such a system RCs held by diskless workstations are volatile and if such workstations crash garbage might be created in other nodes.
In order to implement such a mixed scheme, it is necessary to record the type of RCs a node holds; this can be performed in the client list (see Figure 4) by requiring each element of the RC list to contain two fields - the RC offered to the client node, and in addition a flag indicating whether the RC is stable or not. The bookkeeping information regarding objects (export, and import lists, and the del list) can also be split in two parts with lists on volatile store recording information about volatile objects and stable lists recording information about stable objects. Naturally, a public object will become private only when its reference-count becomes zero on both the export lists. Given this organization, the garbage collection schemes presented for volatile and stable objects can coexist together. For example, whenever orphan servers are deleted at a node, reference-counts of only those RCs which are recorded as volatile in the client list are decremented in the volatile export list. This mixed approach will continue to satisfy both SAF and LIV properties. For example, if C crashes (refer to Figure 2) then the reference-count of O1 will be decremented by one when that crash is detected at A; however, O1 will not be deleted since there still exists a stable RC naming O1.

The scheme presented here has similar functionality to that given in [LiLa86] with the following differences: (i) there is no need to keep in-transit references on stable storage; any inconsistencies caused by crashes during an RC transfer are detected and removed by the enhanced orphan detection scheme discussed in Section 5.2; (ii) the scheme provides a uniform way of treating objects with different lifetime assumptions.

5.4. Performances

A prototype version of the design presented in Sections 5.1 and 5.2 has been implemented on a network of Flex object-based systems [FCE82] running on ICL Perq workstations connected by an Ethernet. Measurements of the overhead caused by the scheme have been made. The measurements were made on a lightly loaded Ethernet. The Ethernet had a raw data rate of 10 megabits per second and was shared with other users.

The measurements have been carried out for null procedure calls performed by a client process to a remote existing server, which returns a new RC at each call. We have measured the average time taken for such a call to complete. This time interval includes the time spent by the client to look up the import list and update it when the call returns with the newly created RC, plus the time the server spends on inserting a new RC in the export list and client list. The average time per call (averaged over 1000 calls) is 51 milliseconds, while the same call without any provisions for fault-tolerance and garbage collection takes 43 milliseconds. The performance degradation due to this scheme is thus of the order of 16%.

We have not measured the influence of the rate for the distribution of the del lists on performance. By empirical observations, it appears that the collection of public objects could be done at intervals of the order of several minutes to hours without affecting the overall performance of the
system. The local garbage collection of every node runs at a much higher rate and it is capable of providing the required storage for ongoing computations.

Finally, it is worth noting that, apart from an extra RPC for the transfer of an RC, the scheme does not require any more RPCs specifically for garbage collection purposes.

6. Concluding Remarks

The topic of fault-tolerant garbage collection in distributed systems, although important, has not received much attention. We are only aware of [LiLa86, Ves87] which addresses this issue. We have presented a practical solution which is both cheap and efficient. By realizing that orphan treatment has much in common with collecting garbage in the presence of faults, we have been able to present an integrated scheme. The garbage collection scheme presented represents small modifications to an efficient orphan treatment scheme implemented at Newcastle [PaSh88], so there is every reason to believe that the technique is of practical value. The performance figures presented bear out this observation.

Some of the advantages of the distributed garbage collection scheme presented here are given below:

1. Collection takes place asynchronously with respect to other activities, including local garbage collection, and creation and deletion of private and public objects;
2. It is independent of the local garbage collection schemes employed at various nodes;
3. It is tolerant to node crashes and communication failures that occur during collection;
4. It is capable of treating both volatile and stable objects;
5. It does not require elaborate facilities such as failure-atomic procedures or synchronized clocks.

Although the scheme as described here has been developed for an RPC based system, there is no reason why the reference-count service could not be implemented on its own for systems not employing RPCs with orphan detection. In this case it will be necessary to implement the crashcount and terminator based mechanism for detecting garbage.

The reference-count based scheme suffers from the limitation (which is well-known) that it is incapable of collecting objects with inter-node references to each other. It would be possible to remove this limitation by extending our design with the ideas reported in [Bob80, Bro85, Ves87].

Acknowledgements

The authors are grateful to Giuseppe Pappalardo, Graham Parrington, Geppino Pucci, Brian Randell, and Simon Wiseman for their technical comments on the ideas presented here. This work was supported in part by a grant from the Royal Signals and Radar Establishment of the U.K. Ministry of Defence.
References


