Fault-tolerant Reference Counting for Garbage Collection in Distributed Systems

L. MANCINI and S. K. SHRIVASTAVA

1 Dipartimento Informatica, Universita di Pisa, 5600 Pisa, Italy
2 Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU, U.K.

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: optimising 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.

Received December 1990, revised May 1991

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. For programming systems which permit access to remote objects, a distributed garbage-collection facility is required. We assume that such systems have been built out of a number of autonomous computers (nodes) connected by a communication network.

This paper describes a 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, 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 tolerates node crashes (fail-silent behaviour will be assumed, that is, a crashed node completely ceases to function).

(2) It does not require elaborate facilities such as failure-atomic procedures or synchronised 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 the underlying remote procedure call mechanism, thus enabling the exploitation of existing fault-tolerance facilities. Moreover, the garbage collection presented copes with message failures (lost, duplicated, delayed and out-of-order messages) by relying on well-known protocol-related techniques implemented within the remote procedure call mechanism.

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 would like RPCs to provide the same semantics as local calls, even in the presence of failures. 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 on 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 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. The design presented here can also be implemented on its own, if integration with the underlying RPC is not practicable. Although our scheme is based on the reference-count technique, which suffers from the well-known limitation that it is incapable of collecting objects if inter-node references form a cycle, a way to remove this limitation will also be discussed.

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++23 Emerald3 and ANSA4). 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 emphasise the distributed nature of the system, we shall refer to capabilities for remote objects as remote names (RN) and assume the existence of some method for locating objects efficiently, given their RNs. 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 RN for the object as one of the arguments. Below we introduce some terminology, and illustrate it with the help of Fig. 1, which shows an object O_1 at node B holding an RN for an object O_2 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_j 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 RNs for other objects (such as O_k holding an RN for O_i). One such mechanism is illustrated in Fig. 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 RNs of that node. Specific details of how objects come to hold RNs for other objects are not directly relevant for our discussion, so will be glossed over. We will, however, assume that objects are capable of copying their RNs to other objects.

Figure 1. O_i holds an RN for O_j.

A distributed computation is performed by client and server processes. The invocation of an operation on O_i by O_j will be carried out as follows: a client process at node B obtains RN_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 RN_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_i, the client process, the export and import lists of B and therefore the RN for O_i at node B vanish and the server on A will become an orphan. Assuming that only O_k held an RN for O_i, then O_k 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 RNs 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 RN for O_j 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 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 RN 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 may cause some remote public objects to become garbage. Consider the system shown in Fig. 2. Suppose that O_k deletes its RN 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. There is also the case where node crashes may cause dangling references. For example, a crash of node D in Fig. 2 will cause O_i and O_j to hold RNs for O_k, which no longer exists. As such, an invocation of some operation on O_k by O_j or O_i may well cause a run-time exception whose treatment, in our view, should be orthogonal to the functioning of the garbage-collection system. Such dangling references can be detected simply by recording the local clock time at which a node is initialised. This time is made part of all RNs to volatile objects owned by that node. When an RN is used, a check is made to see if the time recorded in the RN is the same as the time when the node was initialised. If not, the node must have crashed and so the RN is invalidated. The following sections will assume the use of this technique for dealing with dangling references, and will concentrate in particular on developing a distributed garbage collector capable of dealing with the first undesirable situation.

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. Secondly, the collection mechanism should be able to cope with the copying of RNs 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 minimise 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 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) The majority of garbage collectors for non-distributed systems employ the mark-scan method. 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, 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.

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 localised; 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 cyclically 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.

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. 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.

All describes a number of algorithms for use in a distributed system. The most advanced of his algorithms adopts a copying technique, and does not require any sort of synchronised 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.

An optimised weighted reference count technique has been recently proposed. Here each reference held is assigned a weight. The reference-count associated with an object is constrained to be equal to the sum of the weights of all the references held on that object. One of the main advantages of this scheme is that an object, Z, holding an RN to some object, B, can pass that reference (copy it) to some other object, C, with relative ease, without involving B. Z passes a copy of the RN to C with a part of its weight assigned to this copy (say half); this ensures that the sum of the weights maintained at B remains the same. This certainly reduces the number of messages, but introduces problems when fault-tolerance is required (see below).

None of the methods discussed so far has addressed the problem of fault-tolerant garbage collection in distributed systems. This topic, although important and hard, has not received much attention. To illustrate the problems posed by failures, consider the weighted-reference-count technique. Suppose Z is a volatile object (and B knows this) and the node containing Z crashes. Even if B's node detects this crash, it cannot adjust the weight maintained at B till Z's node recovers, since B does not know the current weight at Z; what's more, the weights associated with references must be kept on stable storage, even for volatile objects.

A scheme has been presented in Ref. 15 which exploits a reliable central service for storing 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 Ref. 24, 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 synchronisation 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 algorithms uses a parallel mark-scan collector based on the algorithm presented in Ref. 9. It resembles the solution in Ref. 1, 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 synchronised clock or special failure atomic procedures.

4. RPCs AND ORPHAN-DETECTION AND KILLING

Orphans are unwanted executions that can manifest themselves due to communication or node failures. 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 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. Fig. 3 depicts a particular example of interference where calls are nested.

We assume the following semantics for an RPC: a normal termination (the client receives a reply from the called server) implies exactly one execution. An abnormal termination on the other hand can mean zero, partial or one execution at the called server. When we consider a sequence of calls made by a client, it is necessary to ensure that the same sequencing is also preserved at the called servers. We express this requirement more precisely below.

Let C, denote a call made by a client and W, represent the corresponding computation invoked at the called server. Let C, and C, be any two calls made by a client such that: (i) C, happens after C, (denoted by C, then C,); and (ii) computations W, and W, share some data such that W, and/or W, 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:

CR: C, then C, implies W, then W,.

The criterion 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 CR will be satisfied. However, failures can create orphans that do require
special measures in order to meet 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 Fig. 3 should be regarded as undesirable, since it is expected that the execution of a sequential program should give rise to a sequential computation characterised 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 CR.

Based on the client–server model briefly described in Section 2, we have developed an efficient scheme for orphan treatment for general-purpose RPCs.¹⁹ There are three mechanisms used for treating orphans.

(i) The first mechanism copes with the situations similar to the one depicted in Fig. 3, where a client call terminates abnormally, leaving open the possibility of orphans on other nodes (these nodes may even be unknown to the client). A simple timer-based solution has been adopted. A client makes a call with a deadline, d, the value of which is also put in the call message indicating to the server the maximum time available for execution. Upon receiving the call, the server calculates its real deadline to be d−d', where d' is based on an estimate of the network latency. If the deadline expires at the server, the server aborts the execution; at about the same time the client's timeout will also expire, resulting in an abnormal termination. The deadline mechanism can be replaced (or even augmented) by 'probe'-based mechanisms, where clients and servers periodically send messages to each other to detect possible crashes.

(ii) The second mechanism deals with the case where a client, after recovering from a crash, makes a call to some node C: it is necessary to ensure that any orphans on C are terminated before post-crash calls are served. 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. Every node is therefore required to maintain a list of servers it has created to service calls from the clients: this list, called the client list, is used for identifying the orphans (more details pertaining to the client list will be presented in a subsequent section).

(iii) Every node has a terminator process that occasionally checks the crashcount values of other nodes, by sending messages to them and receiving local crashcount values as replies from those that are up; it aborts any orphans when it detects that a client has crashed (either no reply has been received or there is a difference in the crashcount values). This mechanism copes effectively with the case where a node remains crashed for a long time.

These mechanisms have been optimised to provide a 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 synchronised. Further, it is not necessary for each and every call to be checked by a server for crashcount comparison. The terminator-based mechanism has been optimised 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).²⁵ although schemes have been developed for systems supporting atomic actions.¹⁶,¹⁷ The method presented in Ref. 17 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 synchronised, although non-synchronised clocks do not produce inconsistencies.

In the subsequent sections of the paper we shall describe enhancements made to mechanisms (ii) and (iii) above to provide garbage collection.

5. FAULT-TOLERANT GARBAGE COLLECTION

The main features of a distributed fault-tolerant reference-counting scheme exploiting the above orphan-treatment system will be presented in the following sections. It will be shown that the scheme satisfies the following two correctness requirements.

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

LIV: if there are zero capabilities in the system for an object, that object will eventually have its reference-count equal to zero.

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 (namely 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.

In Section 5.1 we present a simple fault-tolerant scheme for volatile objects to be used when RNs cannot be copied. In Section 5.2 the refinements required to cope with the copying of RNs will be discussed. We shall exploit only mechanisms (ii) and (iii) of the orphan-detection scheme discussed in the previous section, in that 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. As observed before, since the scheme discussed here is based on the reference-count technique, it suffers from the well-known limitation that it is incapable of collecting objects if inter-node references form a cycle. However, it will be shown in Section 5.4 that this limitation can be overcome by extending the design.

5.1 Treatment of node failures

Nodes are responsible for local garbage collection. Only private objects are candidates for local 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 export 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 at some node A 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 RNs 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 reference exists). The objects listed in the export list may be a superset of those actually used by other nodes. For example, referring to Fig. 1, suppose that O_k at node B holds the only RN for O_i at node A, and that O_k is deleted at B. Object O_i 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 RNs 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 Fig. 4), is 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. An entry of type RNlist is required for garbage-detection purposes. This is a list of elements, each containing two entries: an entry for an RN and a Boolean flag 'unused'. The RNs in the RNlist with the unused flag set to false are the capabilities that have been used by the client whose name is recorded in the clientNode (the remaining capabilities in the list indicate that they have not yet been used by the client). 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 initialised to be empty at the node startup time. Whenever a server receives an invocation, it makes sure that the unused flag in the RNlist for the relevant capability is set to false. 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, its reference-count value is incremented by one, otherwise a new element is added to the export list, with the reference-count field initialised to one.

2. The capability being exported is inserted in the

```
type ClientElem = struct {
  Name cli
  % client node address
  Real crashCount % crash count value of the clientNode
  % list of servers created for the clientNode
  ServerList serverList
  % list of RNs offered to the clientNode
  RNlist rnList
};
```

Figure 4. Client list data structure.

508  THE COMPUTER JOURNAL, VOL. 34, NO. 6, 1991
When 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 RNs recorded in the rnList 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 RN in the del list sent by node B. (1) It checks if the RN is in the rnList field of the entry for node B in the client list, and if so, (2) it deletes the RN from rnList 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 RNs 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 Ref. 19.

It is worth noting that the scheme presented so far ensures SAF and LIV requirements when RNs are not copied. SAF, which requires that only those objects for which no RNs exist (i.e. 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, is decremented, this is because either some node holding an RN for O crashed causing the RN to vanish, or some node sent a del list containing the RN for O. 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 copying of RNs. In this case the scheme presented so far does not ensure that only objects without RNs for them will have reference-counts equal to zero. In the following section this and other issues will be discussed.

5.2 Reliable copying of RNs

One additional mechanism is required to copy RNs reliably while preserving SAF and LIV. Consider the following example. Node A is the owner of a public object and node B holds an RN for that object. B now copies this RN to some node C as a result of a request by C. Inconsistencies can arise if B crashes (causing its RN to vanish) after sending its RN 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 RN for a non-existing object. In order to satisfy SAF, the RN copying 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 has to copy an RN as a part of its RPC reply message, it first makes an RPC (inform–done message pair, see Fig. 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 RN. Referring to the example, if the call by C to acquire the RN 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.

Figure 5. An RN transfer.

Let us now consider situations where the LIV requirement can be violated. With reference to Fig. 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 remain positive for ever. 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 node makes sure that if no calls are received from client nodes for a long time, enquiry messages will be sent to them to detect crashes. This mechanism can be enhanced to take care of unused RNs (RNs listed in the client list that have their unused flags set to true). In the situation presented above, eventually the terminator at A will send an enquiry message to C and will be able to adjust its relevant entries, since C does not hold the RN for the object at A. It is worth while to note that the protocol discussed can cope with crashes of B as well as C: in the latter case, the terminator at A will not get a reply from C, thus will conclude that C has crashed. Care is required in order to avoid a potential race condition: suppose that the enquiry to C from the terminator at A comes before the transfer message from B; then SAF could be violated. This situation can be avoided if node C behaves safely by regarding RNs (possibly) in transit — these are the outstanding getRN calls — as in its possession. Thus, in the situation just discussed, the reference-count for the capability at A will remain unaffected (and the entry in rnList will remain flagged as unused). The terminator at A will be periodically checking the status of unused RNs, so LIV is still guaranteed.
To summarise, nodes are periodically required to exchange three types of information: (i) lists of potential orphans, (ii) lists of unused RNs, and (iii) del lists. The first type of information is required for orphan detection, and the remaining two for garbage detection. A simple optimisation 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 that 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 book-keeping 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 copying RNs between nodes, as the following example illustrates. Suppose that the protocol of Section 5.2 is not employed, then the following situation is possible (Fig. 5): B deletes its RN after sending it to C and then crashes before informing A about the copy. 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 RN in transit are not), the object referred by the RN 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.15

The scheme for the garbage collection of a stable heap discussed above will still satisfy both SAF and LIV properties. SAF is satisfied, since objects are collected only if all their stable RNs have been explicitly deleted. LIV is preserved, since an object will be eventually informed about deleted stable RNs because the del lists are kept stable by the ex-holders; thus the object reference-count will eventually become consistent with the number of RNs 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 RNs for the same object are permitted (e.g. RNs to O; in Fig. 2). An example of such an environment could be a network of nodes, some of which are discless workstations. In such a system RNs held by discless 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 RNs a node holds; this can be performed in the client list (see Fig. 4) by requiring each element of the RN list to contain an additional field: a flag indicating whether the RN is stable or not. The book-keeping 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 organisation, the garbage-collection schemes presented for volatile and stable objects can coexist. For example, whenever orphan servers are deleted at a node, reference-counts of only those RNs which are recorded as volatile in the client list are decremented in the relevant export lists. This mixed approach will continue to satisfy both SAF and LIV properties. For example, if C crashes (refer to Fig. 2) the reference-count of O; in the stable export list will be decremented by one when that crash is detected at A; however, O; will not be deleted since there still exists a stable RN naming O;.

The scheme presented here has similar functionality to that given in Ref. 15, with the following differences: (i) there is no need to keep in-transit references on stable storage: any inconsistencies caused by crashes during the copying of an RN 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 Inter-node cycles

If inter-node references form an acyclic graph, then, when an object of that acyclic graph is collected, all the garbage objects reachable from that object will eventually be deleted. However, if some inaccessible inter-node references form a cycle, inaccessible objects will never be deleted in the scheme proposed so far, as indeed in any pure reference-counting scheme. For example, suppose object x at node A has a reference to object y at node B and y has a reference to object x, as shown in Fig. 6.

![Figure 6. An inter-node cycle.](http://example.com/f6.png)

The inter-node references for x and y form a cycle that spans node boundaries. Even though x and y are both locally inaccessible, they appear to be globally accessible and therefore are not reclaimed by the local garbage collector at their nodes. They are also not recognised as inaccessible with the scheme presented in the preceding sections. In the following, two possible approaches to remove this limitation will be discussed.

One way to devise a cycle-tracing scheme is to employ mark-scan garbage collection. In order to perform a global marking of the storage, each node holds a table which records the marking information for the public objects which are remotely referenced by other nodes. The mark field can take the values: not found, found and scanned. Initially all objects are marked as not found.
When an object is first found to be accessible, its mark is changed to found. Once all the references reachable from the object marked as found have been examined, the object is marked as scanned. The global marking phase terminates once no found objects remain. At this point, all the objects which are marked as scanned must be kept, but the objects marked as not found are known to be inaccessible and so can be collected. This collection takes place during the final scanning phase, which is local to the nodes. It should be noted that the objects marked as not found include those forming inter-node cycles, for example objects \( x \) and \( y \) in Fig. 6 will still be marked as not found at the end of the global marking phase.

In order to adopt the mark-scan scheme to achieve a fault-tolerant cycle-tracing algorithm, the problem of detecting the termination of the marking phase must be solved. Consider the situation depicted in Fig. 7, where the marking information has been included in the entry of the export list at each node. In such a situation, if node A crashes, remote objects \( y \) and \( w \) at node B will remain marked as not found until A recovers. During such time, B could wrongly collect object \( w \). In general, B cannot know whether an object marked as not found is still needed. For example, although objects \( w \) and \( y \) are marked as not found, object \( w \) is still accessible, while object \( y \) is part of an inaccessible inter-node cycle. The solution chosen for this problem is to make as found all the objects being referred to by a crashed node. As previously stated, crashes can be detected by sending enquiry messages with crashcount values kept by the RPC mechanism. To complete the marking phase each node should look up its local client list and mark as found all the objects used by the crashed nodes. In the example of Fig. 7, when node A crashes, objects \( w \) and \( y \) will be marked as found. After the marking phase terminates, the scanning phase will collect the inter-node cycles through all non-faulty nodes.

This distributed cycle-tracing scheme will therefore collect only part of the inaccessible inter-node cycles, and there is the question of whether this can cause the entire system to stop, because of shortage of storage. For example, with reference to Fig. 8, considering the case where only node A crashes, the inter-node cycle spanning node B and C will be collected (that is objects \( o \) and \( q \)), while the inter-node cycle between A and B will not be collected, at least while A remains crashed. However, it is worth noting that the number of inaccessible inter-node cycles cannot increase. A crashed node which prevents the collection of a cycle also prevents the creation of an additional cycle: since the node has crashed, another cycle cannot be made through it. The cycle-tracing algorithm presented above can be optimised when inter-node cycles include volatile objects. For example, if object \( x \) at node A in Fig. 7 is volatile, and A crashes, object \( y \) at node B can be collected without waiting until the end of the cycle-tracing algorithm. This speeds up termination of the cycle-tracing algorithm and allows early collection of volatile objects forming inter-node cycles spanning through crashed nodes.

An alternative approach to a global mark-scan algorithm is to perform a local scan for deleting inaccessible objects when it is believed that a cycle has been formed. Various algorithms of this kind have been proposed in the literature.7-24 For example, Vestal proposes an algorithm that, when started at an inaccessible object lying in a cycle, will collect the entire cycle. The only problem with Vestal’s solution is that it requires an effective heuristic for selecting the starting object, otherwise the collection of the cycle cannot be guaranteed. In our scheme the starting node can be found in the following way.

As discussed previously, every node has a terminator process such that when an object remains unused for a long time that process sends enquiry messages to the client nodes to make sure they are still running. The terminator process can be enhanced to provide an effective heuristic for choosing the starting node of the Vestal’s algorithm by allowing the propagation of an enquiry among the terminators. If an unused object, say \( x \) at node A in Fig. 7, is lying in a cycle, then a message, sent by the terminator process of node A to enquire about the remote clients of object \( x \), will come back to node A after propagation through the nodes in the cycle (a broadly similar scheme has been proposed for the ANSA architecture).25 Therefore object \( x \) can be chosen to start executing Vestal’s algorithm. It should be noted that the terminator process, on receiving back its enquiry message for object \( x \), can only deduce that \( x \) lies in a cycle, but not that \( x \) is an inaccessible object, since one can also have accessible cyclic structures.

Both the schemes discussed above require a crashed node to recover in order to collect inter-node cycles going through that node. This should not be a cause of any increase in the number of cycles, because, as stated before, the number of cycles through a crashed node remains fixed during the down-time of that node.

---

**Figure 7. Cycle-tracing initialisation.**

**Figure 8. Marking after node A crashes.**
5.5 Performances

A trial version of the design presented in Sections 5.1 and 5.2 was implemented on a network of Flex object-based systems running on ICL Perq workstations connected by an Ethernet. Measurements of the overhead caused by the scheme were 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 were carried out for procedure calls performed by a client process to a remote existing server, which returns a new RN 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 RN, plus the time the server spends on inserting a new RN in the export list and client list. Comparing the average time per call (averaged over 1000 calls) with the same call without any provisions for fault-tolerance and garbage collection, it has been found that the performance degradation due to this scheme is of the order of 17%.

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 copying of an RN, 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 have presented a practical solution which actions can be both cheap and efficient. The performance figures presented bear out this observation to some extent. By realising that orphan treatment has much in common with collecting garbage in the presence of faults, we have been able to present an integrated scheme. 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 synchronised clocks.

Although the scheme described here has been developed as an integral part of an RPC system, there is no reason why the reference-count service could not be implemented on its own. Indeed, at the time of writing the revised version of this paper (April 1991), we came across a technical report describing just such a design. It uses import and export lists (called respectively ERT and ODT) in a manner roughly similar to ours. Although there are several subtle differences, on the whole this design can be seen to provide an independent confirmation of the approach presented here. Any reference-count-based scheme suffers from the limitation that it is incapable of collecting objects with inter-node references to each other. Possible approaches to remedy this were discussed in the paper.

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. The work of the first author has been supported by grants from the Royal Signals and Radar Establishment of the U.K. Ministry of Defence, ESPRIT-BRA project no. 3070 (FIDE: Formally Integrated Data Environment), and the Italian C.N.R. 'Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo'. The work of the second author was partially supported by grants from the U.K. Science and Engineering Research Council and ESPRIT project ISA (project no. 2267), through a subcontract administered by APM Ltd, Cambridge.

REFERENCES

Correspondence

A Note on the Optimality of a Reve Algorithm

Sir, In recent issues of The Computer Journal, several authors 1-4 have labourted on a generalised Towers of Hanoi problem called the Reve's puzzle 5 which was proposed and solved by B. M. Stewart 6 some 50 years ago. We follow Rohl and Gedeon 7 to call it the Reve's puzzle 5, not because he was the first one to attempt to solve it, but because he was the first one to describe it in the context of stools and cheeses—in fact he did not present a complete and general solution to it. As some readers of The Computer Journal may not have access to the original publications, it is worth reproducing the original problem statement 8 below.

'Given a block in which are fixed k pegs and a set of n washers, no two alike in size, and arranged on one peg so that no washer is above a smaller washer, what is the minimum number of moves in which the n washers can be placed on another peg, if the washers must be moved one at a time, subject always to the condition that no washer be placed above a smaller washer?'

In the course of deriving a mathematical equation which characterises the minimum number of moves, Stewart 6 described an algorithm for solving this generalised problem. 'Move the n uppermost washers to another peg, using all k pegs; move the remaining washers to a second peg, using the available k -1 pegs; and finally move the n washers to the second peg, once again using k pegs.'

(Note: where n = n1 + n2.)

For a long time, this generalised problem had been solved by Stewart, 6 though another solution was described by J. S. Frame 4 at the same time. Rohl and Gedeon 7 in their first algorithm, did not specify the values of n1 and n2. In their second algorithm 8 the calculations for n1 and n2 are given, but their algorithm does not always generate the minimum number of moves for solving the generalised problem.

In what follows, an algorithm for calculating n1 and hence n2 is presented. But first a minor correction to Rohl's algorithm Reve is made. Note that the last parameter of the algorithm Reve is a value parameter; thus the last swap for exchanging the contents of s[1] and s[m] in the body of the procedure Reve is unnecessary. It is possible to improve the efficiency of the algorithm Reve by deleting the last swap.

procedure Reve(m: nstools; n: ncheeses; s: stools);
begin
if n = 1 then writeln('Move a cheese from stool', s[1], 'to stool', s[2:1]);
else
swap(s[1], s[m]);
Reve(m, n - F(m - 1, n), s);
swap(s[2], s[m]);
Reve(m - 1, F(m - 1, n), s);
Reve(m, n - F(m - 1, n), s);
end;
end; (of procedure 'Reve')

The optimality of this algorithm relies critically on the function F, which computes the value of n2 and is now given below.

function F(m: nstools; n: ncheeses): integer; var i, a, b: integer;
begin
i := m - 1;
while C(i, m - 1) < n do i := i + 1;
a := C(i - 1, m - 1);
b := C(i - 2, m - 2);
if a + b > n then
F := b
else
F := n - a
end; (F)
where C(n, r) computes the binomial coefficient \( \binom{n}{r} \).

Finally, it is worth pointing out that the combination of n1 and n2 for producing the minimum number of moves is not always unique, and thus the above algorithm generates only one of the optimal solutions under these circumstances. For a more elaborate discussion, the reader is referred to Ref. 3.

Yours faithfully,
M. C. ER
Department of Computer Science,
St Patrick's College,
Maynooth,
County Kildare,
Ireland

References