Robust, Distributed References and Acyclic Garbage Collection

Marc Shapiro
Peter Dickman
David Plainfossé

Projet SOR, INRIA, BP 105, 78153 Roquencourt Cédex, France

email: marc.shapiro@inria.fr

Abstract

We propose efficient, programming language-independent, location-transparent references as a substitute for pointers in distributed applications. These references provide the semantics of normal pointers for both local and distributed, transient and persistent objects. They may be passed in messages between and within nodes using a low-overhead presentation-layer protocol. The programmer remains free to create, delete or migrate objects at will. Sending references (or migrating objects) may cause references to be chained together across any number of spaces; we provide a short-circuit protocol to optimize access through such chains.

Integrated with these references, we provide efficient, distributed, garbage collection of acyclic data structures. Even in the presence of network failures such as lost messages, duplicated messages, out of order messages and site failures, the correctness of GC is guaranteed. The protocol assumes the existence of local garbage collectors of the tracing family. The protocol combines: (i) local tracing (from a conservative root); (ii) conservative distributed reference counting; (iii) periodic tightening of the counts; and (iv) allowance for messages in transit during GC. The protocol uses only information local to each site, or exchanged between pairs of sites; no global mechanism is necessary. It is parallel and should scale to very large systems, e.g. tens of thousands of nodes connected using both local and wide-area networks.

1 Introduction

Distributed object-based systems are of ever increasing importance, providing a powerful means of exploiting contemporary hardware. Writing distributed applications is, however, rarely as straightforward as writing local ones. A key reason is that local and remote objects are generally handled differently: a plain pointer may be used when a local object is created; a special handle must be used for remote objects. Code must therefore be aware of which kind of object it manipulates, and may need to be replicated to handle both kinds of objects. The solution to this problem is to provide a uniform programming model for both kinds of objects, that is, to provide a location-transparent reference mechanism.

We propose *references* as a substitute for pointers to objects in distributed applications. References provide uniform identification and access to local and remote objects. Methods of the target object of a reference can be invoked; references can be passed as arguments in invocations, both within and across spaces. Invocation of a referenced local object turns into a procedure call through a pointer. For remote objects, the invocation causes the parameters to be marshalled and sent to the object in remote procedure call (RPC) messages.

Intimately associated with our references is a fault-tolerant protocol for the detection of acyclic distributed garbage.

PoDC '92-8/92/B.C.
© 1992 ACM 0-89791-496-1/92/0008/0135...$1.50

Some programming languages provide garbage collection (GC) to automatically deallocate inaccessible objects. GC is extremely useful, as it simplifies the programming model, therefore freeing valuable programmer time, while avoiding bugs and memory leaks which are notoriously hard to diagnose. However, there has been relatively little work on GC in systems supporting distribution. System designers often dismiss GC, viewed as too complex and/or too costly or just not useful for general applications. They are wrong on all counts. The bulk of distributed GC can be implemented in a generic, language-independent, low-overhead, fault-tolerant manner.

Our GC protocol is inexpensive in that it requires no extra foreground messages or system activity; furthermore, neither additional copying nor additional interpretation of message contents is required. The mechanism is safe in that as long as one reference to an object exists somewhere in the distributed system, the object's storage will not be reclaimed. We minimize the administrative message overhead by piggy-backing and batching. The protocol is fully parallel and scales well: the space complexity is proportional to the number of remote references; it communicates only between pairs of spaces; and third-party dependencies are avoided. As the only information used is local, or exchanged between pairs-of-sites, no global state or synchronization is necessary and neither multicast nor ordered protocols are required.

The underlying assumptions are weak and reasonable. Messages that arrive are delivered in finite, non-zero time; they may be lost, delivered out of order, or duplicated. The network may become partitioned. Nodes may crash silently. Clocks need not be synchronized. Local activities are not synchronized together.

When application code (the *mutator* processes), local garbage collectors, and a distributed collector all execute in parallel, it becomes difficult to guarantee consistency. Some published distributed GC algorithms assume strong consistency. In contrast, our design is based on weakening these assumptions. Strict consistency comes at a high cost; allowing "safe" inconsistency has the potential of greater efficiency, reliability and availability. For example, our protocol permits a space to infer that a remote reference to a local object exists when, in fact, there is no such reference. Other apparent inconsistencies may arise due to messages being in transit. Inconsistencies which do not violate the safety invariants of GC are harmless. Our mechanism relies on this fact, relaxing the conditions that usually guarantee liveness whilst maintaining safety invariants. This relaxation (as opposed to weakening) permits garbage to accumulate in the system. Tightening the conditions then directly results in the reclamation of the garbage. This process of relaxing and retightening is embodied in a straightforward windowing algorithm based on unsynchronised timestamps.

This paper presents our protocol abstractly. Section 2 introduces our model and Section 3 then provides a detailed description of the algorithms and the underlying data structures. After that, Section 4 analyzes the complexity of the protocol in terms of messages, time and space overhead. We compare this protocol to several others in Section 5.

In this presentation certain key features of the mechanism are highlighted and discussed in some detail. We omit the protocol for migrating objects, support for the deletion of non-garbage objects, and persistence mechanisms; these are described elsewhere [25]. We do not address the collection of cyclic distributed garbage here; see however Section 5.

2 Overview

The distributed universe of objects is subdivided into disjoint *spaces*.[1] It is assumed that a space can be identified unambiguously, e.g. by a UID[2]. A space has two possible states. It may be operating and communicating normally; or it may terminate. If a space terminates it does not reappear and its name is never reused. If the hardware it resides on crashes, a space may either persist (recover) or terminate. A space may also appear to cease communicating (*disconnect*), due to network problems, for example, or during temporary overload or recovery after a crash; eventually, however, such a space either recovers (*reconnects*) or terminates. In the case of a crash, our model does not specify whether the affected space(s) recover or terminate, nor how such termination is notified; one could postulate the existence of an external "oracle".

We assume that each space executes a local garbage collector (LGC) of the tracing family[3]. An LGC executes independently of the activity of other LGCs and of distributed cleanup.

Each space A carries a timestamp generator $stamp_A()$. The timestamp generators need not be synchronized across spaces.

Finally, each space also carries an array $threshold_A$

[1] We use the abstract term "space", rather than, for instance, "host", "node" or "process", to avoid committing to a particular implementation or lifetime.

[2] A higher level of distributed GC might be able to ensure that a space name is not reused until no further references to the old space exist; this is beyond the scope of this paper. Therefore we assume that all space names are unique.

[3] Such LGCs are standard in Lisp, Smalltalk, Eiffel and similar environments. LGCs are also being developed which are either language independent [5] or adapted to C and C++ [1, 9]. Tracing GC is fault tolerant, in that each execution of the collector is independent of all previous ones.

of timestamp values received from other spaces and limiting acceptable messages.

2.1 Objects and References

Spaces contain passive objects consisting of instance data and associated methods. An object's instance data may include any number of references to other objects. A reference is a location-transparent handle, through which methods of the target object may be invoked. A reference may be passed as an argument and thus copied between spaces. Our model is illustrated by Figure 1; which all the examples in this text refer to.

Inter-space references are identified in special data structures in the source and destination spaces. A space maintains a table of *stubs* for its outgoing remote references. There is never more than one stub in a space for a given object, even if that space contains many references to that object.

Complementing the stubs, each space maintains a table of *scions*,[4] which track incoming remote references. Every stub that points to a given space has a corresponding scion in that space. A scion may either point to a local object or a further stub; this permits chaining of remote references. Scions are conservative in that the stub corresponding to a scion may no longer exist. This inconsistency does not lead to errors, but may temporarily prevent some garbage from being reclaimed. Note that we use a distinct scion for each stub, unlike other systems in which multiple 'outgoing' pointers will merge into a single 'incoming' pointer.

A stub contains a *locator* composed of two parts, called strong and weak. Each part indicates a scion and consists of an identifier of a space containing the scion, and the scion's name valid within that space. The strong part identifies the scion which matches the containing stub. Distributed garbage collection relies on the invariant that (in the absence of failures) there is always an uninterrupted chain of stubs' strong parts and scions (hereafter called a "strong chain") between the source and target of a remote reference. The weak part identifies a scion which, while part of the same chain of strong indicators, may be closer to the target object[5]. Weak parts are used for communication and location, which rely on the invariant that the indicated scion will not be collected, being protected by the strong chain.

Mutators send and receive messages using a low-overhead "presentation-layer" protocol, with scions and stubs created or updated automatically as needed. The marshalled form of a reference is a locator.

[4] **Scion** *n.* 1. A descendant or heir. 2. A detached shoot or twig containing buds from a woody plant and used in grafting [16].

[5] Indeed, if the target object does not migrate, the weak part is guaranteed to point to its space of residence.

2.2 Garbage Collection

In discussing garbage collection we distinguish the *mutator* and *collector* rôles [8]. Garbage collection poses three distinct problems: distinguishing references from other data in objects; given these references, detecting garbage objects; and disposing of garbage objects, according to their semantics. The former and latter problem are language-dependent and are delegated to the local garbage collectors (LGCs), as is the detection of local garbage. Distributed detection is independent of object structure or semantics and is performed by our protocols. During local garbage collection of space A, the local collector's root set is augmented to consist of the local roots (noted R_A) and the local scions.

As noted in the introduction, a scion may exist for which the corresponding stub no longer exists. This may cause the local garbage collector to believe that there is a remote reference to a local object, whereas, in fact, none exists. For this reason, garbage collection is somewhat conservative. The protocols ensure that eventually the unreferenced scion will be reclaimed. Then, the objects reachable only from that scion become garbage and may be reclaimed by the local garbage collector. In our scheme, all unreferenced scions (i.e. unreachable and not on a distributed cycle of garbage) will eventually be reclaimed (to the extent that the LGC is itself exhaustive).

3 Specification of the Protocol

We distinguish four aspects of the mechanism and highlight novel features for each. Together, these offer both robust distributed references and the automated collection of acyclic distributed garbage:

- The Transport Protocol describes the way in which messages are handled (under what circumstances is a message discarded, for example).
- The Presentation Protocol describes the way in which references are marshalled into, and unmarshalled from, messages.
- The Invocation Protocol details the way in which a reference is used. That is, how locating of the target object interacts with the activity of invoking its methods.
- The Cleanup Protocol covers the elimination of data structures associated with garbage remote references.

When a space terminates, rather complex recovery behaviour may be required of other spaces. Rather than including details of this in each of the four protocols listed above, a separate section addresses Termination Recovery.

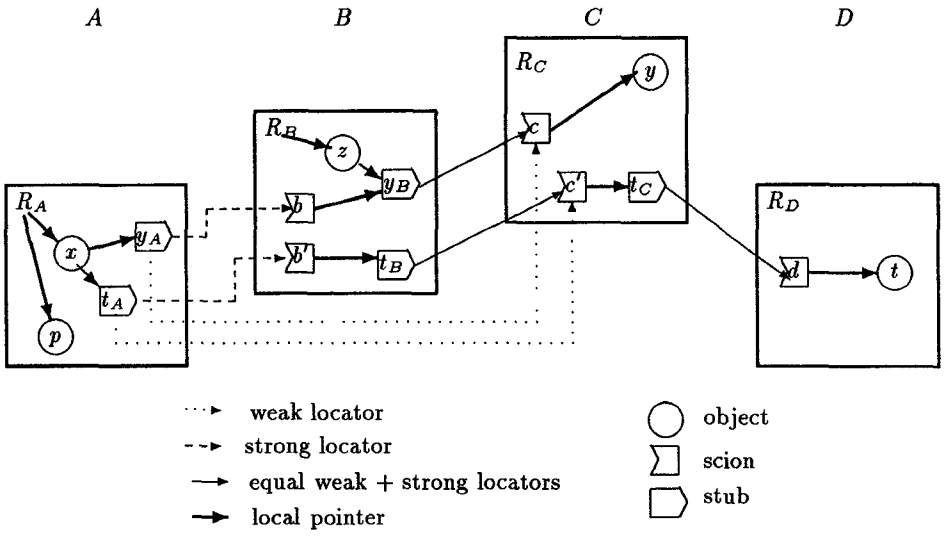

Figure 1: Object and reference model

3.1 Data structures

These mechanisms manipulate data structures of three key forms: scions, locators and stubs. Suppose y is located in space C; when used in B, a reference to y relies on a stub y_B in B and a scion c in C. The stub contains a *locator* $\{C, c, C, c\}$ with *strong* and *weak* parts, both of which, in this simple case, indicate scion c on space C. Scion c itself holds a pointer to y.

Scions contain a space-identifier (indicating the single remote space which potentially contains the matching stub), a locally generated timestamp (produced when a reference which relies on this scion was last locally marshalled), a pointer to the object (or to a stub if the object is remote). Scions are accessed in four modes: (i) for invocation and location, an individual scion is accessed directly, via a scion name included in the weak location of a stub; (ii) by enumeration of the local scions that are associated with a given remote space; (iii) by enumeration of local scions that point at some particular local object or stub; or (iv) by enumeration of all local scions. The enumeration modes are used, respectively, by the Cleanup Protocol, by reference marshalling, and by the local garbage collector. The scion data structure is documented in Table 1.

Locators are the marshalled form of references and are also the primary components of stubs. Locators are documented in Table 2.

Stubs contain a locator and a timestamp. Stubs are accessed in three ways: (i) invocation proceeds directly, through a local pointer to the stub; (ii) when a reference is unmarshalled from a message, it is compared against existing stubs (for unicity, and in case update of the weak part is needed) by strong locator; (iii) the Cleanup Protocol enumerates all local stubs containing a strong part that point at a given remote space. Stubs are documented in Table 3.

Table 1: Scion Data Structure

source_space: space_name	the name of the space holding matching stub.
target_object: pointer	a pointer to the object (or a stub if the target is not local).
stamp: timestamp	a locally-generated timestamp to protect in-transit references

Table 2: Locator Data Structure

strong_space: space_name	Space where next scion in chain resides
strong_scion: scion_name	Scion's name
weak_space: space_name	A space closer to where the object resides
weak_scion: scion_name	Scion's name

A locator contains both strong and weak parts. Unlike most similar mechanisms, both parts do lead (possibly indirectly) to the target of the reference, without any global search. There is always an uninterrupted chain, embodied in the strong locator parts, of stubs and scions from primary source to ultimate destination. The proof that the garbage collector is safe relies on this invariant.

Finally, each space A contains a table of received timestamps, $threshold_A$, indexed by space identifier, ex-

Table 3: Stub Data Structure

location: locator	Locator for the target object
stamp: timestamp	Timestamp to guard against race conditions

plained in the next section.

3.2 Transport Protocol

Communication between mutators in different spaces occurs via messages that are timestamped using the timestamp generator, $stamp_B()$, of the message's source B.

At the destination A, the message timestamp is compared with the B timestamp held in $threshold_A[B]$: only messages containing timestamps generated later than the threshold are accepted. This eliminates a race condition explained in Section 3.5.1.

Our approach usually permits messages to be processed out-of-order. Whenever message ordering could be critical, the required synchronization is explicitly supported using either call-response message pairs or the threshold table. Some delayed messages might be treated as lost, but only in circumstances in which it is possible that acting upon them would violate the GC invariants.

As our mechanisms are designed to tolerate message loss, reordering and duplication, it is acceptable for the message-passing protocol to use cheap, unreliable transport protocols. If an application uses a more reliable protocol, this causes no difficulties as the necessary conditions for this scheme are retained. The background messages required by the cleanup protocol may also use an unreliable protocol, even when the application itself requires a more reliable mechanism.

3.3 Presentation Protocol

A side-effect of marshalling a reference into a message is to produce a scion. There can only be a single scion per target object and per referring space. The marshalling code first searches for such a scion; if none exists it is created; it is timestamped with the current timestamp.

The marshalled form of a reference is a locator, the *strong* part of which names a scion in the sender's space. The *weak* part names a scion closer to the object (but on the same strong chain), if the sender knows one; otherwise it is equal to the strong part. These parts both permit the corresponding remote scion to be unambiguously identified. The message is timestamped with the same timestamp as used to create the scion(s) it refers to.

At the receiver, unmarshalling the reference produces a pointer to a single local stub per matching scion. The actions are: search for a stub with the same strong locator; if one exists and its timestamp is less than the message's, then copy the weak part and the timestamp from the message (if none exists, create one from the weak part and the timestamp in the message); pass up the address of this stub to the application.

Since marshalling and unmarshalling are necessary for remote communication, our approach adds negligible overhead other than creation of stubs and scions, while ensuring no duplicates. Care is taken to ensure uniqueness of the stub-scion pair referring to a particular object between two spaces. This has a small associated cost (discussed in Section 4.2.2), because it requires additional indexing mechanisms and searches, but it renders scion and stub creation idempotent. Hence, deletion of a stub permits the corresponding scion to be discarded without fear that another stub may depend on that scion.

The construction of both stubs and scions is conservative. The endpoints of the remote reference are created without knowing whether they will be useful, and stubs are always created after their matching scions. For instance, it may occur that a message containing a reference is lost; in this case, a scion has been created without a corresponding stub. It may also occur that a received reference is actually ignored by the mutator; in this case the whole reference chain is useless. The stub and scion code can only add new stubs and create more scions. This is consistent with the view that mutators only allocate objects, whereas deallocation is performed transparently by the collector. Here the LGCs remove unreferenced stubs, and the cleanup protocol removes unreferenced scions.

3.4 Invocations and Short-Circuiting Indirect References

A reference is typically used to invoke some procedure (or *method*) of the target object. Remote invocation uses a call-response protocol[6].

3.4.1 Indirect References

Liberal use of indirect reference chains allows a reference to be passed cheaply in messages, while retaining useful invariants (i.e. the guarantee of reachability). But, when considering communication, they are harmful: not only because of the overhead and poor locality, but more fundamentally because sending a reference along an indirect chain creates yet another indirect chain.

[6] Our initial specification [25] was based on one-way messages. A call-response protocol considerably simplifies short-circuiting indirect references.

Consider, for example, an invocation on y, made by x, passing a reference to y itself as an argument. On receipt of the invocation at C, assuming the strong locator chain was used, the argument will be indicated by a chain of stub-scion pairs starting in C and running through B, A, B again and back to C. If this same reference is returned as a result, matters deteriorate further.

3.4.2 Short-Circuiting an Indirect Reference

For these reasons, the weak locator chain is used for invocations[7] and the strong chain is lazily short-circuited in a safe fashion as a side-effect of such invocations. Obsolete indirect stubs and scions will be cleaned up later by the garbage collector. Furthermore, an indirect chain is short-circuited, at the latest before the harm indicated above may occur, i.e. before allowing execution of an invocation carrying reference arguments.

There are two sub-cases to consider. The easiest case is when the caller's weak locator is exact, indicating the scion closest to the target. The other case is when the harmful effect indicated above could occur (the weak locator is inexact, and at least one argument is a reference). The intermediate case (inexact weak locator but no reference argument) can be treated in either way.

3.4.3 Weak Locator Exact

In Figure 1, suppose x calls y.f(), i.e. invokes some method of y with no argument. The call message is sent to weak location $\{C, c\}$. Upon receipt of a call, at scion c, from space A other than the space containing the matching stub (B), a new scion c'' (which does not appear in the figure) is created. Locator $\{C, c'', C, c''\}$ is piggybacked on the invocation results, and the stub y_A at the invoker's space A, through which the invocation was made, is updated to locator $\{C, c'', C, c''\}$. As the original chain: $y_A \rightarrow \{b, B\} \rightarrow y_B \rightarrow \{c, C\} \rightarrow y$ remains in place until the y_A stub contents are changed, the GC invariants are maintained. The superseded indirect chain (the scion $\{B, b\}$ and, possibly[8], the stub y_B and the scion $\{C, c\}$) becomes garbage and will be collected at some later time.

3.4.4 Weak Locator Inexact

Consider now x invoking $t.\mathbf{g}(p)$, some method $\mathbf{g}$ of t with a reference argument (for which a scion a is allocated). Since the weak locator indicates C as above, the call message is similar.

Scion c', receiving this message, detects that x is not local because it points to a stub. The message is passed (without unmarshalling) to t_C, which forwards it on to its own weak locator, i.e. $\{D, d\}$. Here we notice that the target is local but that the caller's weak locator was inexact and the argument is a reference. Therefore the call is aborted and a location_exception is signalled back to A with an up-to-date location. For this, a new scion $\{D, d'\}$, pointing to t, is allocated for use by A.

Upon receiving the exception, t_A updates its locator to point to $\{D, d'\}$. It then retries the invocation (a new scion a' pointing to p on behalf of D must be allocated). Now scion $\{A, a\}$ can be collected, as well as the old indirection chain $b' \rightarrow t_B \rightarrow c' \rightarrow t_C \rightarrow d$.

3.5 Collector Protocol

Above we have specified the *mutator protocol*. Now we will specify the *collector*, i.e. actions performed independently of the mutator's execution, in order to collect garbage. This involves two independent activities: local garbage collection and the distributed cleanup protocol. Reclamation of unreachable stubs and scions is tricky, because of the possibility of lost messages, and of race conditions.

3.5.1 Local Garbage Collection

The LGC traces references from the local root and the set of all local scions. An unreachable stub is garbage, and can be collected. However there is a possible race condition with messages, containing the same reference, arriving late.

The race condition is eliminated by the following rule. Before discarding a stub in A, the strong locator of which points to space B, $threshold_A[B]$ is increased to the value in the stub's timestamp, causing the transport protocol at A to drop earlier messages from B.

3.5.2 Cleanup Protocol

As was noted in the introduction, the mutator protocol relaxes the GC liveness condition, and the cleanup protocol occasionally strengthens it: the strongest form is that every scion has a single matching stub, the weakened form permits some scions to have no matching stub.

Signaling to a scion that the matching stub has been collected is complicated by two potential problems. First, the "deletion" message could be lost. To tolerate message loss, lists of stubs which were still live at some time are sent, rather than sending deletion messages. Second, messages are asynchronous, leading to possible race conditions between scion update and deletion. To avoid the race condition, a scion is removed only if it is both unreachable and there is no message in transit which may make it reachable again. (This race condition is different from the one in the previous section.)

[7] If objects do not migrate, the weak locator is guaranteed to indicate the space containing the target object.

[8] Depending on whether they also form part of other, extant, references at B.

Thus a space A will periodically send to some other space B a message (called a live message) containing: (i) the list of scion names, taken from the strong locators of all extant stubs at A that point at scions at B, and (ii) the value $threshold_A[B]$.

This permits the receiver B to deduce what scions in B are unreachable: precisely those for which there is no matching stub. An unreachable B scion can be removed, if and only if the following condition holds for it at B:

$$scion.\mathsf{stamp} \leq message.\mathsf{threshold}$$

i.e. there are no recent messages in transit carrying its location, which could make it reachable again.

Essentially, we have made stub and scion deletion an idempotent operation and eliminated the race conditions by ignoring messages arriving "too late" and by never discarding scions that have recently updated timestamps. Scion timestamps protect against deletion of scions for which a usable reference may be in transit, whereas stub timestamps protect against re-creation of stubs for which scions have been discarded.

To ensure that progress is made, one space may prompt another to report on the stubs it holds. A space B may send a background prompt message to another space A. On receipt of such a message the live message is sent and processed as indicated above.

3.6 Termination Recovery

In this section the problems arising when a space terminates, are addressed. We define space termination to mean that its local root is deleted, as well as all objects it contains. (References to the deleted objects are detectably dangling; an attempt to invoke the target will raise an exception.) Indirection chains through the terminating space must first be resolved and short-circuited. These rules are easy to enforce when a space voluntarily terminates itself; in other cases a protocol is needed to achieve the same observable effect. There are two aspects: ensuring communication, and re-establishing the invariants.

3.6.1 Recovering Communication

Invocation uses the weak locator of the sender's stub, and therefore is not impaired by a break in the strong chain only. In fact, if the target does not migrate, the weak locator holds its actual location.

If objects do migrate, then the weak locator may point to an intermediate scion, such as t_A to c' in Figure 1. If the weakly located scion is lost, this also breaks the strong locator chain. Here the only possibility is to search exhaustively for the object. Such a search is expensive and may be prohibitive in large systems, so a structuring of the system into collections of spaces with intervening non-terminating gateways would be required.

Furthermore, global search assumes that the holder of a stub knows some unique feature of the sought-after object. Since we don't assume UIDs, the unique feature will be the weak locator part. Thus when an object migrates, its new scion must carry with it the list of scion names under which it had been previously known. This list will be discarded as a side-effect of discarding the scion in the short-circuit protocol of Section 3.4.2.

3.6.2 Re-Establishing the Invariants

Initially it might appear that termination of a space that contains a stub is of little consequence. Unfortunately, it is an error to simply discard the matching scion.

The safety of garbage collection depends on the invariant that an uninterrupted strong chain exists between the source and target of a reference. In turn, the invocation protocol depends on the fact that the scion pointed by a weak locator will not be collected.

We are contemplating a number of possible solutions. The simplest is to retain forever all scions whose source_space has not reliably short-circuited indirections through it.

Our preferred solution improves over the above, by relying on the existence of a global garbage collector (which is necessary anyway to collect distributed cycles of garbage, since they are not removed by the protocol presented in this paper) to detect and remove garbage scions retained in this way. This is the approach taken by Dickman [7] and means that the algorithm is no longer live, but remains efficient and effective.

An alternative would be to apply a rule, similar to the rule for objects, to broken chains: any reference chain indirecting through a terminated space is deemed dangling. Such an approach is only correct, however, if scion identifiers are never reused, as otherwise a different problem of erroneous chain following is introduced. It has the drawback that an object may become unreachable by some path, which happened to go through a terminated space, and remain reachable by others: such inconsistencies are undesirable.

Yet another solution maintains liveness and avoids further errors, but is expensive, involving a large-scale search. On discovering such a stub-less scion a message can be passed down the remaining chain to the object concerned[9]. The message has inserted in it the space and scion identifiers for every scion encountered during the message's journey. Having thus collected a list of all scions that may be indicated by detached sections of the

[9] If the chain is broken in two places the mid-section will be recovered first and this will then recover the association with the most detached part.

chain, an exhaustive search is performed. Each space in turn is presented with the list of intermediate scions and requested to update any and all relevent locators. Again the cost of global searches should be limited by a hierarchic structuring of spaces.

Whatever solution is chosen, the window of vulnerability can be narrowed by aggressively short-circuiting indirections. When a strong and weak locator disagree, a dummy invocation is made, thereby updating the locators. This immediately solves the problem if objects cannot migrate, at a cost in additional messages. The dummy invocation can be performed either immediately upon receiving a reference, or after a time-out, or by a bakcground dæmon.

4 Analysis

A proof of the safety of the algorithm, and a discussion of the circumstances under which it exhibits liveness, are in preparation. Essentially, it is shown that the algorithm can never collect non-garbage objects since the scions form a superset of the existing stubs and act as local roots for the LGC. The timestamp windows, as represented by the local threshold vectors, are fundamental to the proof, given the possibility of messages being in-transit or duplicated. Furthermore, for example, if disconnections do not occur after some initial interval, it is shown that all acyclic garbage is eventually collected by the algorithm (note that this is stronger than claiming that the algorithm is live in the absence of disconnections).

Our references require no foreground messages other than the ones sent by the application. Local processing and memory costs appear acceptable. In addition to tolerating non-byzantine failures, the protocols described do not require any form of global synchronization or snapshot, nor are third parties depended upon in any way. As all of the costs incurred are associated with the handling of references, and the background activities need only commence once a reference is used, no overhead is imposed on applications which choose not to use our mechanism.

4.1 Failures

The failure model used in this work is slightly richer than in most comparable material. Messages may be duplicated as well as lost or delivered out-of-order. Processor pairs are subject to periods during which communication between them may be impossible; however, it is not assumed that this failure is either symmetric or transitive. Processors are fail-stop. It is assumed that messages are not undetectably corrupted and that each timestamp generator produces increasing values.

A particular emphasis has been placed on message-related failures in this presentation, as they are most naturally and coherently integrated into our protocols. The support for recovery when some space terminates is more complex and was presented separately. Overall, these mechanisms permit the collection of acyclic garbage in an environment that is rather more demanding than those postulated by most other approaches.

4.2 Costs

The costs of the protocol are considered according to three different measures: in terms of messages, CPU time and memory space. To provide a baseline against which comparisons can be made, consider the state-of-the art implementation, based on UIDs or capabilities, supporting network transparency, but not garbage collection. This system would support messages and timestamps. Data structures would require marshalling and unmarshalling.

This analysis omits the cost of recovery after a crash.

4.2.1 Messages

An important feature of this algorithm is that it requires no additional foreground messages. Additional messages do, however, arise in the background as a consequence of the cleanup protocol.

The marshalled form of a reference, as held in messages, consists of a locator. If stock hardware is used, a marshalled reference is therefore around 16 bytes long. This is comparable to the size of UIDs in many systems. In both our system and the minimal system, messages are timestamped.

Since a UID is location-independent, locating its target entails a distributed search algorithm. In the worse case, a reliable global search is needed. Maintaining a location cache for recently-used UIDs allows to amortize the cost of the search. There is no such cost with locators.

4.2.2 Local CPU Time

Reference marshalling and unmarshalling require searches for existing scions and stubs, prior to creating new ones. A similar cost arises when short-circuiting indirect references. Passing a UID is typically much simpler, involving a simple copy into the message.

A UID system bears a cost searching through its cache for the location of a message's destination. The processing involved is somewhat simpler than our marshalling and the cost is borne only once per message.

Finally, we use additional CPU time in executing the local garbage collector, and in interpreting the live messages of the cleanup protocol. We perform these ac-

tivities in the background, however, so the impact on application performance is minimal.

All the costs listed above are local. A space never needs to wait for another any more than required by the mutator.

4.2.3 Memory

The per-space memory costs of this approach depend on the degree of locality exhibited by the applications executed in the system. Three major data structure types are required: the threshold vector, stubs and scions.

The following estimates of memory usage can be made. Assuming for simplicity of analysis that timestamps, space-identifiers, scion names and local pointers each occupy four bytes, and that all indirection chains and garbage have been eliminated:

- The threshold vector requires one entry, of 8 bytes, for each known remote space.
- There is a single stub for each remote object that is locally referenced, requiring 24 bytes.
- There is a single scion per object for each remote space that contains references to that object; it occupies 20 bytes.

In addition, hash tables or the like are required to implement the accesses modes listed in Section 3.1. These costs are not unreasonable: a maximum of 8+24+ 20 = 52 bytes per remote reference, across the system as a whole, compares unfavourably, but not appallingly, with the cost of 16-byte UIDs supported by a location cache. Some hotspots may arise, however, if particular well-known objects are referenced from a great many remote spaces, due to the accumulation of scions.

4.3 Measured Performance

We have prototyped an earlier version of our protocol, called SGP [25], on the distributed Lisp Transpive [18]. This version lacks weak locators, uses a message protocol rather than call-reply and uses an extra timestamp vector instead of timestamping stubs. A detailed account and analysis of this experiment may be found in [19].

For our evaluation, we replaced Piquer's original distributed Indirect Reference Count (IRC) collector. Our protocol provides the same functionality as IRC, and is furthermore scalable and resilient to message and space failures.

In this section, we compare the measured performance of our prototype with IRC, in terms of communication and CPU overhead. Our measurements of two applications (merge sort and matrix multiplication) were taken on a Parsytec board composed of four T800 Transputers with one megabyte of memory each, hosted in a Sun. Each application is timed twice in a row; the figures are better the second time because of Transpive's caching policy. The measurements, repeated dozens of times, have shown extremely low variance. Our experiments were able to test resilience to message loss but not to termination, due to lack of a fault-tolerant application. Furthermore, we were not able to quantify how conservative or how scalable our protocol is.

Table 5: Message Overhead

Application	Control Messages					
	IRC		SGP		IRC – SGP	
(sort 100)	31	28	10	8	21	20
(sort 200)	41	39	10	8	31	31
(mult 20 20)	101	96	20	18	81	78

Table 4 shows local execution times. The overhead is due to management of (the Transpive equivalent of) stubs and scions. Our implementation is on average 10% slower than IRC and 20% slower than with distributed collection turned off. This result is encouraging: our implementation is not optimized and retained some obsolete data structures and processing from Piquer's implementation. Furthermore our protocol does more than IRC.

Table 5 measures message overhead. IRC sends "delete" messages, whereas we periodically send live messages. This buffering reduces dramatically the number of control messages.

Although our object model does not take replication into account, it was necessary for Transpive; it proved quite easy to add. But Lisp's extremely fine granularity of objects is very demanding, requiring a huge number of stub and scions which consume a lot of space, increasing the garbage collection overhead.

5 Related Work

This section compares our proposal with related work, in the two areas of location-independent references and distributed garbage collection.

5.1 Location-Independent References

Many distributed systems [17, 21, 24] rely on fixed-length, location-independent Universal IDentifiers (UIDs) to designate and locate objects throughout the network. UIDs do not scale well. Uniqueness can be guaranteed only within some domain; cross-domain references require a separate mechanism. A UID does not carry location information; locating its target entails a global search in the general case. Furthermore, UIDs are not pointers, forcing programmers to use two very different mechanisms.

Table 4: Execution Times

Application	CPU time in seconds No DGC		IRC		SGP		Overhead SGP/IRC	
(sort 100)	3.8	3.2	4.7	3.9	5.5	4.1	17%	5%
(sort 200)	5.6	4.4	6.7	5.2	8.1	5.9	20%	12%
(mult 20 20)	11.1	7.8	12.1	8.7	13.5	9.8	11.9%	12.3%

Our reference mechanism owes much to the links of Demos/MP [20] and the forwarders of Emerald and Hermes [3, 4]. In contrast to their proposals, our references are intimately associated with GC. This is made possible by the invariants maintained by our protocol.

Fowler [10] proposes chaining forwarders to provide continuous access to highly mobile objects. Fowler analyzes three alternative location protocols (distinguished in how they short-circuit indirection chains), demonstrating that the cost decreases dramatically when the number of accesses increases faster than the number of moves. His Jacc protocol bears some similarities with ours. Our stubs carry more information than Fowler's forwarders; for example, our weak locator accesses a non mobile object in a single hop, whereas a forwarder is, in effect, a strong locator. In Fowler's design, a highly mobile object may inform others of its current location, requiring something similar to the source space information in our scions.

5.2 Distributed Garbage Collection

One important problem of distributed garbage collection is maintaining the consistency of scions with stubs in the face of failures. A common approach is to use reliable mechanisms to enforce strong consistency, which is expensive. A more recent approach is to relax traditional GC invariants.

Our scheme is based on the latter alternative and bears similarities to some proposals based on reference counting [6, 18]. Unlike those approaches, however, a scion is maintained per source space, which permits us to tolerate message loss while avoiding the dangers of duplicated delete messages.

Dickman [6] proposes an optimised weighted reference counting (oWRC) algorithm. In order to deal with unreliable communication protocols, oWRC preserves a weak invariant enforcing that each object weight (total weight) is always greater or equal to the sum of all remote reference weights (partial weight). The use of a weak invariant allows the algorithm to tolerate message loss but duplicated messages remain problematic.

Mancini and Shrivastava [15] present an efficient and fault-tolerant reference-counting distributed garbage collector. A reliable RPC mechanism, extended to detect and kill orphans, provides resilience to failures. A special protocol copes with duplication of remote references, by making an early short-cut of potential indirections even if they are never used.

In the future we expect to add to our protocol a separate mechanism to deal with distributed cycles of garbage, which are not currently handled. There are several proposals in the literature, e.g. Bishop's migration technique [2] or Schelvis' cycle-detection technique [22]. We will discuss below Liskov's logically centralized algorithm, Hughes' timestamp algorithm, and Lang's dynamic grouping technique.

Lang *et al.* [13] propose to combine a distributed reference count with the dynamic grouping of nodes with distributed mark-and-sweep within each group. The reference counts must be accurate, hence message failures are not tolerated. The mark-and-sweep algorithm relies heavily on termination protocols, which are not scalable. A distributed garbage cycle that crosses group boundaries is not collected until another group is formed, enclosing the whole cycle; therefore liveness is not guaranteed. A failure during a collection causes group reorganisation excluding the failed node, restarting the group GC.

Hughes [11] uses a global clock. A collector, starting from some local root at time t, marks all objects it reaches with the value t. The marking on a reachable object will advance periodically; on an unreachable object the mark will not change. Objects marked with a date less than some global minimum are collected. Determining the minimum requires repeated execution of a global termination algorithm. Furthermore, if even a single processor is disconnected, it is impossible to advance the minimum.

Liskov and Ladin [14] describe a fault tolerant distributed garbage detector based on their highly available logically-centralised service. Each local collector informs the centralised service of incoming and outgoing references, and about the paths between incoming and outgoing references. The path computation is expensive but necessary for reclamation of distributed garbage cycles. Based on the paths transmitted, the centralised service builds the graph of inter-site references, and detects garbage (includin dead cycles) with a standard tracing algorithm. The centralised service informs LGCs of accessibility of objects.

In a later paper [12] Ladin and Liskov simplify, and correct the deficiencies of, the above proposal, adopting Hughes' algorithm and loosely synchronised local clocks. Hughes' algorithm eliminates inter-space cycles of garbage, thereby eliminating the need for for an accu-

rate computation of the paths and for the central service to maintain an image of the global references. Furthermore, the centralized service determines the garbage threshold date, making a termination protocol unnecessary.

6 Conclusion

We have presented scalable location-transparent references to objects in a distributed system, with well-defined failure semantics. Integrated into the approach is fault-tolerant automatically collection of acyclic distributed garbage, which can be combined with any reasonable local garbage collection algorithm. The mechanism is effective, inexpensive, straightforward and is based on a novel combination of well-known techniques. Our mechanism requires no global search or synchronization and uses a very cheap transport protocol (not requiring multicast communications or any particular ordering on messages). The key enabling concepts are the scions, i.e. inverse reference lists (as opposed to reference counts), and the use of timestamps and windowing protocols to support idempotent deletion. In conjunction a the conservative creation policy, these provide a fault-tolerant and efficient mechanism.

The current specification suffers from some limitations. First, only acyclic garbage is collected; it will be necessary to extend the mechanisms to collect distributed cyclic garbage. Second, recovery from space termination is incompletely specified. Third, although the main-line protocol is scalable, the recovery protocol entails global search; to limit the cost of search, we pointed at the need to structure the universe into small partitions (in which exhaustive search remains realistic) connected by gateways, but this needs more work.

A first version of the garbage collection protocol has been prototyped; its measured performance is similar to an existing, non fault-tolerant, non scalable, distributed collector. We are currently in the process of implementing the specifications of this paper, as a system level facility in the Soul object-support layer [23].

Acknowledgments

We particularly wish to thank Daniel Edelson for his helpful comments and discussions, especially those concerning the presentation of this material. We would also like to thank Olivier Gruber, Bernard Lang and Robert Cooper for their comments on earlier versions of this work.

References

[1] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Western Research Laboratory, Palo Alto, CA (USA), February 1988.

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

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, October 1986.

[4] Andrew P. Black and Yeshayahu Artsy. Implementing location independant invocation. In *Proceedings of the 9th Int. Conf. on Distributed Computing Systems*, pages 550–559, Newport Beach, CA USA, June 1989. IEEE.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9):807–820, September 1988.

[6] Peter Dickman. Optimising weighted reference counts for scalable fault-tolerant distributed object-support systems. Submitted for publication, 1992.

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

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

[9] Daniel R. Edelson. A mark-and-sweep collector for C++. In *Principles of Programming Languages*, pages 51–57, Albuquerque, NM (USA), January 1992.

[10] Robert Joseph Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proc. 5th Annual ACM Symp. on Principles of Distributed Computing*, pages 108–120, Alberta, Canada, August 1986.

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

[12] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Int. Conf. on Distributed Computing Sys.*, Yokohama (Japan), 1992.

[13] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, Albuquerque, New Mexico (USA), January 1992.

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

[15] L. Mancini and S. K. Shrivastava. Fault-tolerant reference counting for garbage collection in distributed systems. *The Computer Jounal*, 34(6):503–513, December 1991.

[16] William Morris, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, 1980.

[17] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[18] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, volume I of *Lecture Notes in Computer Science*, pages 150–165, Eindhoven (the Netherlands), June 1991. Springer-Verlag.

[19] David Plainfossé and Marc Shapiro. Experience with a fault-tolerant garbage collector in a distributed lisp system. Submitted for publication, April 1992.

[20] M.L. Powell and B.P Miller. Process migration in Demos/MP. In *9th ACM Symposium on Operating System Principles*, volume 17, pages 110–119, October 1983.

[21] Marc Rozier, Vadim Abrossimov, Franccois Armand, Ivan Boule, Frédéric Herrmann, Michel Gien, Marc Guillemont, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Willi Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–370, December 1988.

[22] Marcel Schelvis. Incremental distribution of timestamp packets: a new approach to distributed garbage collection. In Norman Meyrowitz, editor, *OOPSLA'89 Conf. Proc.*, volume 24 of *SIGPLAN Notices*, pages 37–48, New Orleans, LA (USA), October 1989. ACM Sigplan, ACM.

[23] Marc Shapiro. Soul: An object-oriented OS framework for object support. In *Workshop on Operating Systems for the Nineties and Beyond*, pages 251–255, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

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

[25] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.