SSP chains : robust, distributed references supporting acyclic garbage collection

Marc Shapiro, Peter Dickman, David Plainfossé

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: +33(1)39 63 55 11

Rapports de Recherche

N° 1799

Programme 1

Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

SSP CHAINS: ROBUST, DISTRIBUTED REFERENCES SUPPORTING ACYCLIC GARBAGE COLLECTION

LES CHAÎNES DE PSS, DES RÉFÉRENCES RÉPARTIES ROBUSTES PERMETTANT LE RAMASSE-MIETTES ACYCLIQUE

Marc Shapiro
Peter Dickman
David Plainfossé

Novembre 1992

SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection*

Les chaînes de PSS, des références réparties robustes permettant le ramasse-miettes acyclique

Rapport de Recherche INRIA no. 1799

Broadcast Technical Report no. 1

Marc Shapiro
Peter Dickman†
David Plainfossé‡

Novembre 1992

*This is a revised and extended version of a paper presented at the Symposium on Principles of Distributed Computing, Vancouver (Canada), August 1992.

†ERCIM fellow; current address: GMD, Institut I5.RS, Schloß Birlinghoven, 5205 Sankt Augustin 1, Germany

‡Also with Laboratoire d'Informatique Théorique et de Programmation, Université Pierre et Marie Curie, Paris

Abstract

SSP chains are a novel technique for referencing objects in a distributed system. To client software, any object reference appears to be a local pointer; when the target is remote, an SSP chain adds an indeterminate number of levels of indirection. Copying a reference across the distributed system extends an SSP chain at one end; migrating the target object extends it at the other end. Invocation through an SSP chain is efficient: each stage of an SSP chain contains location information and long chains are short-cut at invocation time. These actions require (almost) no extra messages in addition to those of the client application. The rules for creating, using, modifying and deleting SSP chains are stated precisely and maintain well-defined invariants. The invariants hold even in the presence of message failures (loss, duplication, late delivery); after a crash, the existence invariants must be re-established. SSP chains support distributed garbage collection (GC); we present a robust distributed variant of reference counting. The techniques presented here are cheap, robust, widely applicable, and scalable.

Résumé

Les chaînes de paires souche-scion (PSS) sont une nouvelle technique pour référencer les objets dans un système réparti. Pour un logiciel client, toute référence apparaît comme un pointeur local ; lorsque sa cible est distante, une chaîne de PSS ajoute un nombre indéterminé de niveaux d'indirection. Lorsqu'on copie une référence à travers le système réparti, la chaîne est étendue à l'une de ses extrémités ; lorsque l'objet cible migre, c'est à l'autre extrémité que la chaîne est étendue. L'invocation est efficace à travers une chaîne de PSS ; ef effet, chaque élément de la chaîne contient l'adresse de l'élément suivant, et les chaînes trop longues sont shuntées au moment d'une invocation. Ces actions ne nécessitent (quasiment) aucun message en plus de ceux de l'application cliente. Nous spécifions de façon précise les règles de création, d'utilisation, de modification et de destruction des chaînes de PSS, règles qui maintiennent des invariants bien définis. Ceux-ci restent vrais même en cas de panne de message (perte, duplication ou livraison tardive) ; après un crash, les invariants d'existence doivent être rétablis. Les chaînes de PSS permettent le ramasse-miettes réparti ; nous présentons une variante robuste et répartie du comptage de références. Les techniques présentées ici sont bon marché, robustes, d'application universelle, et s'adaptent bien aux systèmes de grande échelle.

1 Introduction

We address the well-known problem of identifying objects in a distributed system. The identification of an object may be copied between the *spaces* of the distributed system, allowing other "source" objects to form *references* to the identified "target" object.

Our design, illustrated in Figure 1, is a novel combination of techniques and rules. The main features of this work are the following:

- A remote reference is represented as an arbitrary-length *chain* of *stub-scion pairs* (SSPs).

  Each outgoing *stub* contains the location of two incoming *scions*. The *strong* location indicates the next scion in the chain; the *weak* one indicates some scion of the same strong chain, typically closer to the target.

  SSP chains are an elegant, efficient and simple solution to the problem of distributed object identification.

- The rules or *protocols* for passing and using references, i.e. for constructing, modifying, and destroying SSP chains, are stated precisely. The rules ensure that a set of invariants hold.

  The protocols are cheap, in that (almost) no additional foreground messages are needed, other than those sent by the application, and no application data needs to be re-interpreted, copied or modified by some lower level.

- From the SSP structure and the invariants, one can infer the existence (or lack) of remote references to any particular object; therefore they support distributed garbage collection (GC).

- Our particular distributed GC algorithm is a conservative variant of reference counting. It uses the semantics of SSP chains and of garbage collection to tolerate inconsistencies and failures cheaply.

  Inconsistencies never violate the invariants; this ensures GC is safe. The usual liveness conditions are relaxed, permitting garbage to accumulate in the system; but they are periodically tightened, direcly resulting in (local) reclamation of garbage.

- The failure assumptions are weak; in other words, our techniques have wide applicability.

  Messages can be arbitrarily lost, delayed, duplicated, or delivered out of order; sites may crash in a fail-stop manner; clocks need not be synchronized. We do assume an oracle detecting the termination of spaces. In general this is a strong assumption. However, even if such an oracle does not exist, the invariants remain true, and GC remains safe.

- Our techniques are scalable, since there is no global information, knowledge, data structures, or synchronization. Information flows only between pairs of spaces.

1.1 Overview and Definitions

An object is any identified entity. Its shape, size, contents, and semantics are arbitrary. An object may contain arbitrary references, represented as local

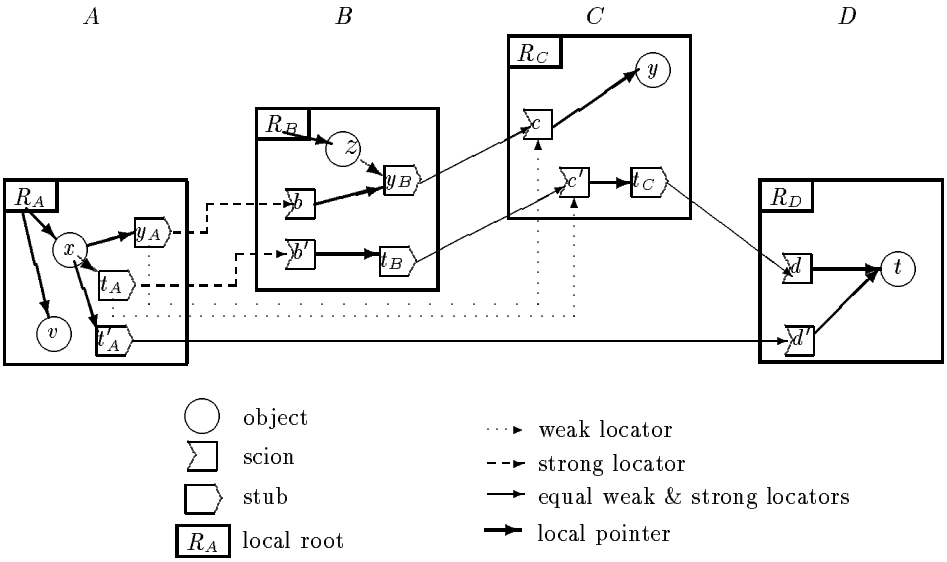

Figure 1: Object and reference model

pointers, to others. An object has an arbitrary but well-defined procedural interface; some known procedure arguments or results may be references which can be assigned to variables of the receiver, modifying the reference graph dynamically.

To send a reference across a space boundary, it must be "marshalled" into a message, i.e. the pointer value is translated into a standard external representation by a *Presentation Protocol*. On reception, it is "unmarshalled" into another pointer. Marshalling and unmarshalling append a stub-scion pair (SSP) at the tail of the reference known to the sender. For instance, in Figure 1, sending the reference to y from space B to space C extended the pre-existing pair composed of stub y_B in space B locating scion c in space C.

Objects may move or *migrate* between spaces. When an object migrates, it leaves behind an SSP that is appended at the tip of any existing chain targetting that object. This ensures that the object remains accessible. (It also suggests a migration protocol [4]: freeze the state of the object; copy it to the new space; set up the final SSP; commit the migration in the source space by atomically setting all pointers to the object to point to the new stub; unfreeze the state.) In the figure, object t has migrated from space C to D, adding the stub-scion pair $t_C \rightarrow d$ to the tip of the chain ending at c'.

Sending a reference and migrating an object are the only two ways of creating or extending an SSP chain.

The *Invocation Protocol* allows the source of a chain to invoke some procedure of the target. If the chain is too long, it is short-cut as a side-effect of invocation. If the source's weak location was exact, but the strong chain had indirections, a direct SSP is found or allocated and piggy-backed on the response. If the weak location was inexact (i.e. the target has migrated), then the target returns an exception with an exact weak location. This brings us back to the previous case. There is no message or message hop overhead for short-cutting, except when the target has migrated. (As a possible optimization, long chains may also be short-cut by background activity before strictly necessary.)

Unreferenced SSPs are automatically freed by garbage collection (GC). Distributed GC has two components, local GC and a distributed *Cleanup Protocol*. In each space, an (assumed) local garbage collector collects locally-inaccessible objects. The Cleanup Protocol propagates accessibility information from one local garbage collector to the next, in occasional background messages.

As in most modern systems, our references are location transparent, i.e. a client program will access an object using the same kind of reference, whatever the target and its current actual location. References are also network transparent, i.e. a program cannot distinguish between target locations, because a stub presents a procedural interface identical to its target's. However, contrary to common practice, our references are not necessarily location independent, i.e. the contents of our data structures may depend on the current or past location of the target object. It is essential for performance that a stub contains the actual address of its target. Similarly, stubs do not need to contain some universally-unique quantity. A scion name is context-dependent, hence its scalability is not a problem.

Note how, in Figure 1, the two scions d and d' of space D both refer to the same local target t. This is in contrast with similar systems (such as Emerald) where multiple outgoing references (stubs) merge into a single incoming structure (scion). Our multiple scions per target, one per source space, are essential for fault tolerance.

Note also that space A has received two references t_A and t'_A from different sources: the former from B and the latter from D. From within A, they are not directly comparable, even though in this case both refer ultimately to the same target t. After short-cutting, the chain through $t_A$will disappear, and both references will be via t'_A, and hence will become equal.

1.2 Structure of this paper

This paper presents our design abstractly. The prose is supplemented with data structures and pseudo-code. These illustrate our intent and are written for clarity, not performance.

Section 1 is this introduction. Section 2 introduces our design and notation. Section 3 then provides a detailed description of the data structures and the four protocols (Transport, Presentation, Invocation, and Collector Protocols) that act upon them. After that, Section 4 analyzes the complexity of the protocol in terms of messages, time and space overhead, then considers the effects of strengthening some of the assumptions; it also reports briefly on the performance of a prototype implementation. We compare our design to related ones in Section 5. The conclusion, Section 6, briefly reviews the properties of this work.

Some omitted features are described elsewhere: manual deletion [30], support for persistent objects [31], implementation experience [23], implementation design [17, 22], objects in shared virtual memory [10].

We are aware of some limitations of this work. First, we do not address the problem of cyclic distributed garbage here, although we do analyze some possible solutions in Section 5. Second, the recovery protocol is based on exhaustive search. This does not scale; although we hint at a solution (divide the universe such that search is feasible in any single partition, partitions being connected by non-terminating gateways), this needs more work.

A full-scale implementation and a proof of this design are currently in preparation.

2 Overview

Our model is illustrated by Figure 1, which the examples below will refer to. We consider a distributed system partitioned into disjoint spaces. A space contains objects. Objects carry references to other objects, possibly across space boundaries; a remote reference is represented as an SSP chain. As a result of program activity, the reference graph changes dynamically. An object which is no more accessible via some path of references is *garbage*; its space is wasted and should be reclaimed.

2.1 Spaces

The distributed universe of objects is subdivided into disjoint *spaces*, the large squares labeled A through D in Figure 1. A space can be identified unambiguously, e.g. by a UID.[1]

Each space A carries a timestamp generator $stamp_A()$. The timestamp generators need not be synchronized across spaces. Every reading of a stamp generator yields a different increasing value. When a reference is sent in a message, the same timestamp value is used to stamp the message, the stub and the scion of the associated SSP.

Finally, each space also carries an array $threshold_A$ of timestamp values received from other spaces and limiting acceptable messages.

We use the abstract term "space" to avoid committing to a particular implementation. For instance, at the lowest level of our current implementation, spaces are address spaces, the scope of space names being the processor; at the next level up, each processor and each disk partition is a space, and the scope is the local net; at the top level, each local net is a space of an internet. In a separate implementation [10], the spaces are instead regions of shared distributed virtual memory.

While in operation, a space communicates with others via messages obeying the protocols defined herewith. Eventually, it terminates. Thereafter it does not reappear and its name is never reused.

A space may *disconnect*, i.e. appear to cease communicating due to various problems such as network overload or partition, or during a reboot. Disconnection need not be complete nor symmetric. A disconnected space cannot modify the distributed reference graph; disconnection is therefore safe. Eventually, a disconnected space either reconnects (e.g. recovers) or terminates.

After a crash, our model does not specify whether the affected space(s) recover or terminate. Abrupt termination of a space due to a crash poses a recovery problem, covered in Section 3.7. We postulate an external oracle capable of detecting and notifying termination; however in its absence, terminated spaces just appear permanently disconnected and our protocols remain safe.

[1] A higher level of distributed GC can 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 here that space names are unique.

2.2 Objects and References

Spaces contain passive objects (the circles labeled x, y, z, t, v in Figure 1) with a procedural (or method) interface. An object may hold, as a *source*, references to other objects *(targets)*, through which methods of the targets may be invoked.

A reference may be passed as an argument and thus copied between spaces. Local references, i.e. within a space, are assumed to be much more common than remote ones, i.e. across spaces (locality).

Stubs represent outgoing remote references; *scions*[2] represent incoming ones. Stubs and scions are used, managed, updated, and deleted by our protocols; client software is not aware of their existence.

For client software, all references are represented by pointers. A local reference is a pointer to the target. A remote reference is a pointer to a local stub, representing the remote target locally, and presenting the same procedural interface.

A stub contains a *locator* subdivided in a *strong* and a *weak* part. Each part contains the location of a scion, as an identifier of a space containing the scion, and the scion's name valid within that space. The strong part identifies the *matching* scion, i.e. the next scion in this SSP chain; we will sometimes use the expression *strong chain* to emphasize this fact. The matching scion itself points to the target, either directly or via a further stub-scion pair. The weak part identifies some scion on the same strong chain, either the same as the strong part, or one closer to the target.

There is never more than one stub in a space for a given scion, even if that space contains many references to the same target. Conversely, every stub that points to a given space has a corresponding scion in that space. We use a distinct scion for each stub, for instance scions d and d' in space D both point to local target t; the former on behalf of C, the latter on behalf of A. This is unlike other systems [4, 15, among others] in which multiple outgoing references will merge into a single entry item.

A scion may exist without a corresponding stub (scions are conservative). This inconsistency is safe, but may temporarily prevent some garbage from being reclaimed.

Distributed garbage collection relies on the invariant that (in the absence of crashes) there is an uninterrupted strong chain between the source and target. Weak parts are used for communication and location, which rely on the invariant that the weakly indicated scion will not be collected, being protected by the strong chain.

2.3 Stub-Scion Pairs

A remote reference is represented as a chain of stub-scion pairs (SSPs). A chain starts its existence as a single SSP, either when sending the reference of a local object to some other space, or when migrating an object to some other space. An existing SSP chain is extended in similar circumstances. Each stub's strong locator indicates the next scion in the chain. Its weak locator indicates some better path to the target, if one can be known without exchanging extra messages.

[2] **Scion** *n.* 1. A descendant or heir. 2. A detached shoot or twig containing buds from a woody plant and used in grafting [19].

We will illustrate this by two examples, shown in Figure 1. First, consider copying a reference. The remote reference to object y located in space C, held by object z in space B, is passed to x in space A; the strong chain will pass indirectly through space B. The weak locator will, however, point directly to a scion c in space C. Note that the stub matching that scion lies on the strong chain and is in B, not A. The invariants below ensure however that this scion will not be collected as long as a weak locator uses it.

Now consider migration of object t located in space C. B holds a (direct) reference to t through a stub t_B and a scion c'; A holds an indirect one, stub t_A containing strong locator $\{B, b'\}$ and weak location $\{C, c'\}$. Now t migrates to space D; it leaves behind a trail supporting continued access to it: in space C, it replaces itself with stub t_C, containing locator $\{D, d; D, d\}$ to a scion that itself points to t's new location. This use of chains is similar to Emerald [8, 12].

The liberal use of indirect SSP chains makes it cheap to pass references in messages, while retaining useful invariants. The Invocation Protocol short-cuts indirection chains at first use.

Note that following the paths indicated by either the strong and the weak locators leads exactly to the target, without use of hints, global names, or global search (in the absence of crashes).

Two stubs received from different destinations are not directly comparable, even though they may refer to the same ultimate target. For instance, in Figure 1, two stubs t_A and t'_A ultimately refer to t in D. They cannot be compared or merged because the former's strong locator identifies space B whereas the latter identifies D. However, when t_A's long chain will be eventually short-cut, both references will use t'_A (hence will compare equal).

2.4 SSP Chain Invariants

The protocols presented herewith preserve a number of invariants:

1. There is an uninterrupted SSP chain from primary source to ultimate destination.
2. Every stub has a *matching* scion (i.e. identified in its strong locator).
3. A scion matches at most a single stub.
4. The weak location part of a stub names a scion which is also reached by following the strong locator.

Invariant 1 is the garbage collection safety invariant; correctness of the collector protocols relies on it.

The fault-tolerant character of our protocols is based on invariant 2.

Invariant 3 is the relaxed liveness invariant. In a strictly consistent system, each scion would match a stub at all times and acyclic garbage would be collected "instantaneously." By relaxing the invariant we allow some distributed acyclic garbage to be retained; the Cleanup Protocol periodically tightens the inequality, thereby collecting the retained garbage.

Invariant 4 is the communication invariant. It implies that following the weak locator will get to the target, and that this path is shorter than (or in the worse case, the same length as) following the strong locator. Furthermore, invariant 1 implies that the weak location is immune to garbage collection; the

conjunction of invariants 1 and 4 is the safety invariant for communication. The Invocation Protocol depends on the fact that the scion pointed by a weak locator will not be collected.

The existence invariants 1 and 2 only hold in the absence of crashes; when the oracle dectects a crash, they are re-established by the recovery protocols of Section 3.7.

2.5 Garbage Collection

By definition, any object reachable from a predetermined *root*, via the transitive closure of the "refers to" relation, is "live"; an object not live is garbage. Unreachability is a stable property: once an object becomes garbage, it remains so forever.

Reclamation of unused memory can be manual (as the last reference to an object is removed) or automatic *garbage collection* (detecting which objects have become unreachable). Notoriously prone to bugs and memory leaks, manual reclamation is a programmer's nightmare, that can only get worse as large, shared, distributed, and persistent stores become prevalent. SSP chains support distributed garbage collection.

We assume that each space executes a local garbage collector (LGC). During local garbage collection of space A, the LGC's root set consists of the local roots (noted R_A) and the local scions.

Ideally, the LGC should use a tracing algorithm such as mark-and-sweep or copy-collect (see [34] for a survey of modern LGC techniques). However, reference counting and even manual reclamation are acceptable. The distributed Cleanup Protocol does not interfere with LGC activity; it only propagates reachability information from one LGC to the next. LGCs execute independently of one another.

The GC literature distinguishes the *mutator* and *collector* rôles [7]. Collection includes three distinct problems: distinguishing references from other data; given the 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 LGCs, as is the detection of local garbage. Distributed detection is independent of object structure and is performed by the Cleanup Protocol.

A scion may exist without the corresponding stub. For this reason, garbage collection is somewhat conservative. The Cleanup Protocol ensures that eventually the unreferenced scion will be reclaimed; objects reachable only from that scion become garbage and may be reclaimed by their LGC. 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.

2.6 Problems of Distributed GC and Some Solutions

We will now look at the problems of distributed GC and some solutions found in the literature. Compared to centralized GC, new problems appear because the universe is partitioned into separate spaces communicating through messages. Each space executes in parallel with the others. In an idealized distributed system, messages might be considered instantaneous or reliable, and crashes and disconnections might be ignored. But in fact these assumptions are not realistic.

2.6.1 The Distributed Reference Counting Problem

The naïve extension of reference counting keeps the reference counter with the target object. Every copy or deletion of a reference to some object sends a control message to the target. This is costly, especially since the control message must be delivered reliably, exactly once.

Furthermore, messages should be delivered to their destinations in causal order (as defined in [14]). As an example of what can happen otherwise, consider site B, holding a reference to object y located on site C. It sends that reference to site A and immediately deletes its own reference to y. The decrement message from B may arrive to C before the increment message from A, incorrectly causing object y to be discarded.

Some variations such as Weighted References [1, 9] do not have this particular problem, but do remain sensitive to lost messages and crashed sites.

2.6.2 The Distributed Tracing Problem

A standard approach to distributed tracing is to combine independent local, per-space collectors, with a global inter-space collector. The two types of collector interface to each other through data structures similar to our stubs and scions.

The problem of fault-tolerant distributed tracing is to maintain the consistency of scions with stubs, in the face of message and site failures, and of race conditions. In fact, if LGCs, mutators, and the inter-space collector all operate in parallel with each other and messages are not instantaneous, strict consistency is not achievable. A LGC bases its decisions on local, necessarily *inconsistent* information.

Here is an example illustrating this consistency problem. Consider a system (not illustrated in any of the figures) composed of only two spaces, A and B.

1. Space A, holding a reference to object x, accessible from its local root, sends a reference to x to space B, then destroys its own reference to x.
2. Space A performs a local collection and concludes that x is not locally accessible.
3. Space B sends the reference to x back to A, then destroys its own reference to x.
4. Space B performs a local collection; similarly, it concludes that x is not locally accessible.
5. The global collector summarizes the information received from the LGCs: x is not accessible in either A or B. It wrongly concludes that it is garbage (in fact it is now accessible from A).

A message containing a reference may be in transit at the time of a LGC; this can be a problem, as illustrated in the next scenario:

1. Space A holds a reference to object x in space B.
2. A's reference to x is destroyed.
3. The global protocol informs B that A no longer holds a reference to x.

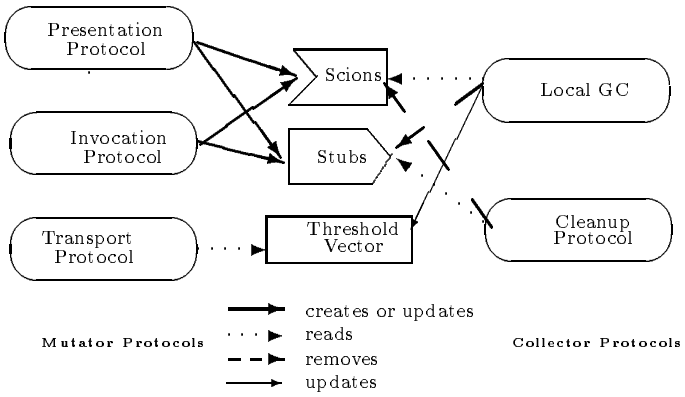

Figure 2: Relationships between the protocols and main data structures.

4. While the destruction message is in transit, B again sends the reference to x to A, then makes x unreachable from B's local root.
5. The destruction message arrives to B; x is collected.
6. A receives the reference to x; x is now reachable from A's root, although it has been collected.

A standard response is to fight inconsistency by using strong protocols such a global barrier to synchronize the end of all the local mark phases [15]. Our approach instead is to allow *safe inconsistencies*, i.e. which do not violate the safety invariants of GC.

3 Specification of the Protocol

The rules for managing SSP chains are presented as a set of protocols: three layered mutator protocols, i.e. protocols for exchanging application messages, and a collector protocol, subdivided into local and remote components. These protocols all mantain the invariants of Section 2.4.

- The Transport Protocol specifies which messages are accepted or discarded.
- The Presentation Protocol says how to marshall a reference into, and unmarshall it from, a message.
- The Invocation Protocol details how, when invoking an object, it is located and any indirect SSP chain is short-cut.
- The Cleanup Protocol eliminates scions associated with garbage remote references.

Section 3.7 separately addresses recovery after a space has terminated.

In Figure 2 the relationships between the four protocols and the local garbage collector are shown in terms of the data structures provided at each space.

3.1 Data structures

These mechanisms manipulate three key data structures: 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.

A scion (see Table 1) contains 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), and a pointer to the object (or to a forwarding stub if the object is remote). The following functions access scions:

- For invocation and location, function find_scion_by_name (scion_name) accesses an individual scion via a scion name taken from the weak location of a stub.
- Function scion_from_space_set (space_name) returns the set of the local scion names that are associated with a given remote space. The Cleanup Protocol compares the result of this function against the data in a live message to determine unreachable scions (see Section 3.6.2).
- scion_to_object (object, space_name) returns the local scion that points at some particular local object or stub on behalf of a particular space, or null if it doesn't exist. This is used when marshalling a reference to ensure scion uniqueness.
- The local garbage collector traces from the local root and from the set of local scions, returned by function all_scions_set().

Auxiliary functions used in the pseudo-code include name_of_scion (scion) that returns the name of the local scion passed as its argument. Figure 4 lists pseudo-code for the function that builds a locator for a local reference (an object or a stub), possibly allocating a scion.

Table 1: Scion Data Structure

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

Locators (Table 2) are the marshalled form of references and are also the primary components of stubs.

A locator contains both strong and weak parts. Unlike most similar mechanisms, both parts do lead (possibly indirectly) to the target of the SSP chain, without any global search.

Stubs (Table 3) contain a locator and a timestamp. Stubs are accessed in three ways:

- Invocation proceeds directly, through a local pointer to the stub.

Table 2: Locator Data Structure

strong_space: space_name	Space where next scion in chain resides.
strong_scion: scion_name	Name of next scion within its space.
weak_space: space_name	A space closer to where the object resides.
weak_scion: scion_name	Name of closer scion in its space.

- Function stub_matching_scion (space_name, scion_name) returns the stub whose strong locator matches the arguments, or null if none such exists. The Presentation Protocol calls this function to unmarshall a locator received in a message, returning a pointer.
- Function stubs_to_space_scion_set (space_name) enumerates the local stubs that point in their strong part to a given space and returns the set of (strong) scion names they contain. This is used by the Cleanup Protocol to send a live message containing a list of valid scions.

Table 3: Stub Data Structure

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

Finally, each space A contains a table of received timestamps, $threshold_A$, indexed by space identifier, explained in the next section.

3.2 Transport Protocol

The specification of the Transport Protocol is deliberately weak. Although we do assume messages are not corrupted, we do not attempt to detect lost, late or duplicate messages. Embellishments to that effect are perfectly acceptable (in fact, they might be essential to the applications) but are not necessary for correctness of our mechanisms, and provide only minor benefits (see Section 4.3).

Communication between mutators in different spaces occurs via messages that are timestamped using the timestamp generator, $stamp_B()$, of the message's source B. Table 4 shows the message structure.

Table 4: Standard Message Stucture

sender_space: space_name	Identify sender to receiver.
sender_timestamp: timestamp	Sender's timestamp.
type: message_type	Message types will be introduced as needed.
...	Specific to message type.

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 race conditions explained in Sections 3.6.1 and 3.6.2.

If a message is received from a previously unknown space, a new entry can be made in the timestamp vector, with the timestamp found in the message as the initial value. Once an entry has been made in the local timestamp vector, for a given remote space, it must be maintained even if all stubs indicating that space are discarded. Only if it can be guaranteed that there are also no messages in transit from that space, is it safe to remove the entry.

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 dropped (i.e. treated as lost), but only when acting upon them could violate the invariants.

As our mechanisms are designed to tolerate message loss, reordering and duplication, it is acceptable for messages to be carried by cheap, unreliable underlying 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.

Figure 3 lists in pseudo-code some functions which are involved in Transport Protocol. The low-level function transport_send() timestamps and sends messages using an transport layer. Function transport_receive() compares the message's timestamp with threshold for that sender. Under normal circumstances the message is accepted and passed to the upper layer. Function call_and_receive() provides blocking RPC; for simplicity, this version omits timeout and retry.

3.3 Presentation Protocol

We assume that a "stub generator" (similar to Birrell and Nelson's [2] or to RPCgen [32]) makes remote invocation appear to be a local operation. A stub generator generates code for the stub (their "client stub") and for the scion (their "server stub") from an interface specification. The stub's interface is identical to the target's, but a generated operation simply "marshalls" the procedure name and arguments into a message, sends it, and later "unmarshalls" the response message. Symmetrically, a scion unmarshalls a call message, dispatches it to the appropriate target procedure, and marshalls the result into a response message. Marshalling and unmarshalling are the Presentation-layer Protocol; they create or update the stubs and scions corresponding to any references passed as arguments. Marshalling of non-reference data types does not concern us here.

Figure 4 lists pseudo-code for the function ptr_to_locator() that finds a scion for a local reference (an object or stub) and returns a locator to it. The locator is the marshalled form of a reference. There can only be a single scion per local target (object or stub) and per referring space. The marshalling function ptr_to_locator() first searches for such a scion; if none exists it is created. In both cases, the scion is timestamped with the timestamp of the message being marshalled.

The locator's *weak* part identifies a scion closer to the object, but on the same strong chain, if the sender knows one; i.e. if the sender's reference points to a stub, then the weak part is taken from that stub. Otherwise the weak part is equal to the strong part.

```
procedure transport_send(target_space: space_name, ts: timestamp, msg: message)
    msg.sender_space := this_space()
    msg.sender_timestamp := ts
    low_send(target_space, msg)
end procedure

function transport_receive(msg: message)
     % ignore late messages when acting upon might create inconsistencies
     if msg.sender_timestamp < threshold[msg.sender_space] then
          exit                                                          10
     end if
     ... forward message according to its type ...
end function

function call_and_receive(target_space: space_name,
                          ts: timestamp,
                          call_msg: call_message): message
    msg: message
    call_msg.sequence_nb := stamp()
    transport_send(target_space, ts, call_msg)                          20
    msg := ... wait for a reply or exception message with same sequence_nb ...
     return msg
end function
```

Figure 3: Transport Protocol: transport_send() fills standard message fields; transport_receive() checks for timestamp value before passing up the message; call_and_receive() implements a synchronous call above transport functions.

At the receiver, unmarshalling the reference produces a pointer to a single local object or stub per matching scion. Figure 5 shows function locator_to_ptr(), responsible for translating a locator to a local pointer. A special case must be first handled: an SSP chain might loop back to the caller. Therefore, if either weak or strong part of a locator refers to a local scion;, the function returns the local pointer found in the target_object field of that scion.In the normal case, the function first searches for a matching stub. If it exists, its timestamp is increased and its weak part is updated with the weak location found in the locator. If none exists it creates one

Table 5: Message Containing Marshalled Reference

sender_space $= B$	From B to A.
sender_timestamp $= stamp_B()$	
type $= \ldots$	Either 'call' or 'reply' message.
$\ldots$	Whatever is needed by the message type.
data $= \{B, b; C, c\}$	Locator for reference to y.

As an example, consider sending the reference to y from B to A in Figure 1. The contents of the corresponding message will be found in Table 5. Space B creates a scion b, pointing to y_B (its local representation of y) on behalf of A, or updates it if it already exists, setting its timestamp to the message timestamp. The marshalled form of the reference to y is locator $\{B, b; C, c\}$. When A unmarshalls the reference from the message, it will create (or update)

```
function ptr_to_locator(ref: object_or_stub,   % pointer to object or stub
                        src: space_name,       % on behalf of this space
                        ts: timestamp          % timestamp of current call message
                        ): locator
requires:: ts is timestamp of current message
ensures:: returned locator indicates a scion which points to a object

    sc: scion
    critical section
        sc := scion_to_object(object, src)                                  10
        if sc = null then
            sc := scion.allocate(ref, src, ts)   % see below
        else
            sc.stamp := max(sc.timestamp, ts)
        end if
    end critical section

    % initialize locator weak and strong parts with accurate known location
    l: locator
    if is_a_stub(ref) then                                                  20
        l := {this_space(), scion_name(sc);
              object.location.weak_space, object.location.weak_scion}
    else
        l := {this_space(), scion_name(sc);
              this_space(), scion_name(sc)}
    end if
    return l
end function

function scion.allocate(ref: object_or_stub,                                30
                        src: space_name,
                        ts: timestamp): scion
    ... allocate memory for self ...
    self.target_object := ref
    self.source_space := src
    self.stamp  := ts
    return self
end function
```

Figure 4: Presentation Protocol. ptr_to_locator() finds or allocates a scion pointing to a local object or stub, for use by some remote space, and returns a locator to it. scion.allocate() allocates a fresh scion, denoted **self**.

```
function locator_to_ptr(l: locator,          % received locator
                        ts: timestamp,       % timestamp sent along with locator
                        ): object_or_stub    % returns a reference to local object or stub
requires:: timestamp and locator extracted from same message
requires:: timestamp greater or equal to threshold
ensures:: returned reference refers to target of locator

    % special cases: if strong or weak locator loops back to this space,
    % then return reference of object or stub pointed by corresponding scion
    if l.weak_space = this_space() then                                          10
        return find_scion_by_name(l.weak_scion).target_object
    end if
    if l.strong_space = this_space() then
        return find_scion_by_name(l.strong_scion).target_object
    end if
    st:stub
    critical section
        st := stub_matching_scion(l.strong_space, l.strong_scion)
        if st = null then
            st := stub.allocate(l, ts)        % otherwise allocate a new stub     20
        else
            if st.timestamp < ts then         % update only if better weak
              st.timestamp :=  ts
              st.location.weak_space  := l.weak_space
              st.location.weak_scion  := l.weak_scion
            end if
        end if
    end critical section
    return st
end function                                                                     30

function stub.allocate(l:locator, ts: timestamp): stub
    ... allocate memory for self ...
    self.location := l
    self.stamp := ts
    return self
end function
```

Figure 5: Presentation Protocol. Unmarshalling a reference, locator_to_ptr() takes a locator and returns a local pointer. It first looks for a matching stub to update and return; if it doesn't exist, it creates a new one. It does not create or update stubs in two cases: when the weak part of locator names a local scion or when a call message loops back through calling space. stub.allocate() allocates a fresh stub denoted **self**.

a stub, copying the weak locator from the message data and the timestamp from sender_timestamp.

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 SSP chain 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-Cutting SSP Chains

An SSP chain is typically used to invoke some procedure (or *method*) of the target object. Remote invocation uses a call-response protocol[3], making no particular assumptions of reliability or ordering of messages.

This specification ignores time-outs, re-sends, etc., that might be essential to application software but are unimportant to our mechanisms. It would not be difficult to add them to the retry loop of Section 3.4.5.

3.4.1 Basic Call-Response Invocation Protocol

Invocation proceeds by sending a call message from the source of a reference to its target; the latter executes the corresponding procedure, and answers with a reply message. Tables 6 and 7 show the format of the call and reply messages. A reply is sometimes replaced by an exception; we will mention location_exception in Section 3.4.5 and deleted in Section 3.7.

3.4.2 Indirect SSP Chains

We now examine indirect SSP chains, that use multiple, redundant hops through intermediate spaces. Liberal use of indirection chains supports cheap passing of references in messages, while retaining useful invariants (i.e. the guarantee of reachability). But, when considering communication, long chains are harmful: not only because of the overhead and poor locality, but more fundamentally because sending a reference along an indirect SSP chain creates yet another indirect SSP chain. In Figure 1, consider an invocation via t_A, made by x, passing a reference to t itself as an argument. On receipt of the invocation at D, the argument will be indicated by an SSP chain starting in D and running

[3]Our initial specification [30] was based on one-way messages. A call-response protocol simplifies short-cutting indirect SSP chains.

Table 6: Mutator Call Message

sender_space: space_name	Sender space for this hop.
sender_timestamp: timestamp	Sender's timestamp.
type = 'call'	Message type is 'call'.
sequence_nb: timestamp	To pair response with this call.
callee_scion: scion_name	Name of current target.
caller_space: space_name	Space of initial caller.
procedure: opcode	Distinguishes between called procedures. .
data: bytes	Call arguments, marshalled according to Section 3.3.

Table 7: Mutator Reply Message

sender_space: space_name	Sender space for this hop.
sender_timestamp: timestamp	Sender's timestamp.
type = 'reply'	Indicates this is a mutator reply message.
sequence_nb: timestamp	Equal to that of matching call.
callee: locator	Up-to-date locator for called target.
data: bytes	The results, marshalled according to Section 3.3.

through C, A, C again and back to D. If this same reference is returned as a result, matters deteriorate further.

3.4.3 Short-Cutting an SSP Chain

For these reasons, the weak locator chain is used for invocations[4] and the strong chain is lazily short-cut 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-cut, 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; then the call message proceeds directly to the target, and a direct SSP *overrides* the long chain as a side-effect of the reply.

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. Here, the call message is forwarded on to the target, which returns an exception containing the exact location. The source stub adjusts its weak locator (which is now exact) and tries again, which brings us back to the previous sub-case.

The intermediate case (inexact weak locator but no reference argument) can be treated in either way.

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

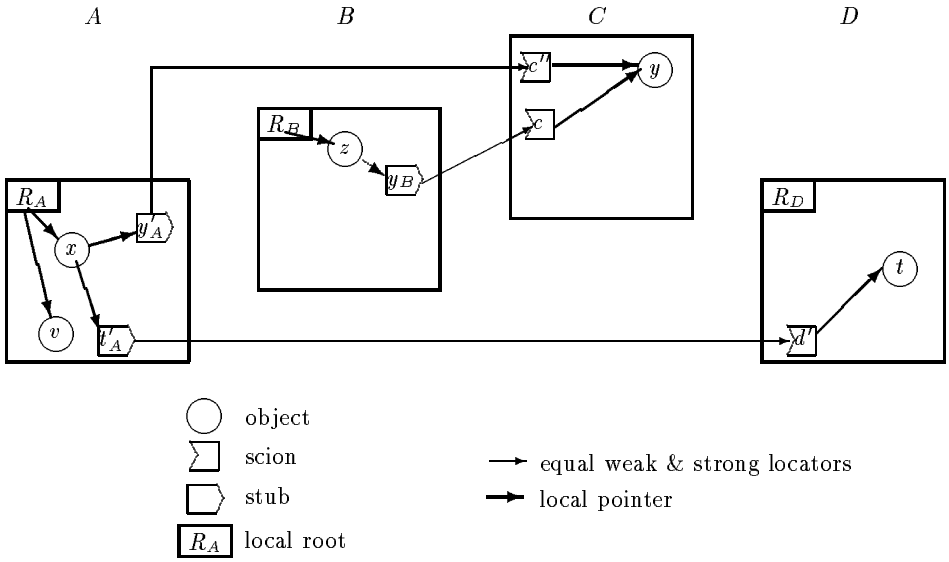

Figure 6: After short-cutting the indirections, and garbage collection.

3.4.4 Short-Cutting Strong Chain

In Figure 1, suppose x calls $y.\mathsf{f}(v)$, i.e. invokes some method of y with argument reference to v. The call message, shown in Table 8, contains a marshalled reference to v, specifically, a locator for scion a that points to v for space C (not shown in the figures). This call 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'' is created. Scion name c'' is piggybacked on the invocation results, as shown in Table 9. A new stub y'_A is allocated at the invoker's space A, containing locator $\{C, c''; C, c''\}$, overriding the original stub y_A, through which the invocation was made. This caller's pointer to y_A is changed to point to y'_A. Other references to y_A (especially possible pointers from scions to y_A) are not affected.[5] (The way in which the new pointer is reflected back to the caller is language dependent and will not be considered here; our implementation is documented elsewhere [17].)

Because of the possibility of overriding, the Invocation Protocol implementation must be careful to deliver a response to the same stub that sent the call, not to one overriding it.

As the original chain: $y_A \rightarrow \{b, B\} \rightarrow y_B \rightarrow \{c, C\} \rightarrow y$ remains in place until y_A is eventually collected, the invariants are maintained. The superseded indirect chain (the scion $\{B, b\}$ and, possibly,[6] the stub y_B and the scion $\{C, c\}$) becomes garbage and will be collected at some later time. This is illustrated in Figure 6.

[5] In the PODC-92 version of this paper [28], we stated that y_A could be updated to contain $\{C, c''; C, c''\}$. This is incorrect because $\{B, b\}$ might be used by a weak locator, which could become dangling. We thank Pierre-Guillaume Raverdy for pointing out this bug to us.

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

Table 8: Example Call Message

sender_space = A	Call message from A.
sender_timestamp = $stamp_A()$	
type = 'call'	
sequence_nb = $stamp_A()$	
callee_scion = c	
caller_space = A	Initial caller is A.
procedure = f	Call method f.
data = $\{A, a; A, a\}$	Locator for scion to v for C.

Table 9: Return Message, Weak Locator Exact

sender_space = C	Reply from C to A.
sender_timestamp = $stamp_C()$	
type = 'reply'	
sequence_nb = call.sequence_nb	Match this reply with call.
callee_scion = c''	Up-to-date scion name for y for use by A.
data = ...	

3.4.5 Short-Cutting Weak Locator

In Figure 6, consider now invoking t.f(v), method f of t with a reference argument, through t_A. Since its weak locator indicates C as above, the call message is similar; only the timestamp and the callee_scion field are different (for the argument, the same scion a is used).

Table 10: Location Exception Message

sender_space = D	Location exception C to A.
sender_timestamp = $stamp_D()$	
type = 'location exception'	
sequence_nb = call.sequence_nb	Match this exception with call.
better_weak = d	Scion in sender space.

Scion c' knows that t is not local, since c' points to a stub t_C. Upon receiving this message, c' forwards it to the weak location target of t_C (i.e. $\{D, d\}$), without unmarshalling it, but updating the callee_scion and stamp fields, and resetting the message type to 'forwarded call'.

The destination scion $\{D, d\}$ notices that the message has been forwarded (by examining the message type) and that 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 (see Table 10). For this, scion name d is sent back to caller in the better_weak field of the exception message.

Upon receiving the exception, the weak locator of t_A is updated with the scion name found in the exception message. The caller then retries the invocation as shown in Table 11. Assuming the target hasn't migrated again in the

Table 11: Retry Call Message (differences only)

callee_scion $= d$	t is now in D.
data $= \{A, a'; A, a'\}$	Single argument: locator for v.

meantime, this brings us back to the case where the weak locator is exact but the strong chain is indirect; hence Section 3.4.4 applies. As explained there, a side effect of the call is to shortcut the strong SSP chain.[7]

Note also that since a scion d' to t was already allocated by D for A, a new scion need not be allocated. After short-cutting, both references from A to t are via the SSP $t'_A \rightarrow d'$. Although the two references were previously not comparable, they do reduce to the same stub after short-cutting.

3.5 Invocation Pseudocode

Figure 7 lists the caller part of the Invocation and the Presentation Protocols. In this example, stub.f() (derived from the interface of the target's procedure f), takes two reference arguments and returns a reference.

It is important to note that a single timestamp is used throughout the whole marshalling of each call message. The Invocation Protocol loops, sending a call message, until receiving a reply message, i.e. until a direct invocation succeeds. Several calls may be attempted if the target object is highly mobile and moves between retries. Each retry uses a new timestamp, and the reference arguments are marshalled again for the new target space. The location exception message carries a better weak locator which is used to update the stub's weak locator and retry. When receiving a reply message the stub function f() exits from the retry loop. If a shortcut has occurred, it creates a new, overriding stub. In the pseudo-code, the overriding stub is propagated to the caller by assigning to self.

Figure 8 shows pseudo-code for the invoke() procedure, on the callee side. If the target object is not local, the message is forwarded along the weak chain without unmarshalling. Otherwise the message is processed locally. If the object is local, but the message has been forwarded, then a location exception message is sent back to the caller with an up-to-date scion name. This scion will be used to retry the call (see Fig. 7). If the call is direct, then the scion invokes the object and returns the results to the caller. As a side effect, a short-cut may be performed. In this case, ptr_to_locator() may create a scion. The name of the scion is sent along the reply message in the callee_scion field.

Figure 9 lists the pseudo-code of a scion's invoke() function. This example has three functions, corresponding to the target object's interface. The invoke() function is responsible for unmarshalling arguments, dispatching to the appropriate target procedure and marshalling the result. Marshalling and unmarshalling of references call ptr_to_locator() and locator_to_ptr() respectively.

3.6 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

[7] We previously would shortcut SSP chain at once when receiving the location exception. We thank Dennis Shasha for this more elegant solution.

```
function stub.f(arg1: object_or_stub, % arguments taken from target's specification
                arg2: object_or_stub
                ): object_or_stub       % same for return
requires:: object is remote
ensures  : remote invocation carried out

    new_locator: locator
    receive_msg : message
    % retry loop over call_and_receive; marhsall references
    % with a new timestamp at each iteration                                    10
    while true do
        ts: timestamp := stamp()                    % a single timestamp for each call
        target: space_name := self.location.weak_space
        % marshall reference at each iteration
        call_msg.data[0] := ptr_to_locator(arg1, target, ts) % marhsall first argument
        call_msg.data[1] := ptr_to_locator(arg2, target, ts) % marshall second argument
        receive_msg := call_and_receive(target, ts, call_msg)
        new_locator := {receive_msg.sender_space, receive_msg.callee_scion;
                        receive_msg.sender_space, receive_msg.callee_scion}
        switch (receive_msg.type)                                              20
            case 'reply':    % call succeeded
                exit while   % exit loop

            case 'deleted' : % target is deleted
                ... propagate back to caller ...

            case 'location exception': % retry invocation
                self.location.weak_space := receive_msg.source_space()
                self.location.weak_scion := receive_msg.better_weak
        end switch                                                             30
    end while
    % after short-cut: override stub, propagate back to caller
    if self.location.weak_space ≠ self.location.strong_space then
        self := locator_to_ptr(new_locator, receive_msg.sender_timestamp)
    end if
    % unmarshall result
    return locator_to_ptr(receive_msg.data[0], receive_msg.timestamp)
end function
```

Figure 7: Invocation and Presentation Protocol, sending an invocation. The stub function for a remote function f(), stub.f() implements the location exception retry loop and does argument marshalling by calling the presentation layer functions. It applies to a particular function stub denoted **self**. If a shorcut has happened, it propagates the accurate address of overriding stub back to caller by assigning to **self**.

```
procedure scion.receive_call(call_msg: call_message)
requires:: message type is either 'call' or 'forwarded call'
ensures:: message delivered or forwarded

    l: locator
    ts: timestamp := stamp()

    % weak inexact: forward call message along weak chain without unmarshalling
    if self.object is a stub then
        call_msg.sender_space := this_space()                                  10
        call_msg.callee_scion := self.object.weak_scion
        call_msg.type := 'forwarded call'
        transport_send(self.object.location.weak_space, ts, call_msg)
        return
    end if

    caller: space_name := call_msg.caller_space
    % weak exact but message forwarded: reply with an exception message
    % containing accurate weak locator
    if call_msg.type = 'forwarded call' then                                   20
        except_msg: location_exception_message
        except_msg.better_weak := scion_name(self)
        transport_send(caller, ts, except_msg)
        return
    end if

    % normal case: weak exact and message not forwarded: perform invocation
    % and reply possibly with up-to-date locator
    reply_msg: reply_message
    % perform invocation, prepare reply                                        30
    reply_msg.data := self.invoke(caller, call_msg.sender_timestamp,
                                  ts, call_msg.proc, call_msg.data)
    reply_msg.sequence_nb := call_msg.sequence_nb

    if self.source_space ≠ caller then      % if indirect SSP chain then shortcut
        reply_msg.callee_scion := ptr_to_locator(self.object, caller, ts).strong_scion
    else
        reply_msg.callee_scion := call_msg.callee_scion
    end if
    transport_send(caller, ts, reply_msg)                                      40
end procedure
```

Figure 8: Invocation Protocol, handling and receiving invocations. scion.receive_call() is responsible for processing call message for a particular scion (denoted **self**); replies with a reply or an exception.

```
function scion.invoke(source_space: space_name,   % initial caller space
                      sender_ts: timestamp,       % timestamp sent along with call message
                      msg_ts: timestamp,          % timestamp used for mashalling references
                      proc: opcode,                % code for invoked function
                      args: bytes                 % marhsalled arguments
                      ): bytes                    % marshalled results
requires:: self.object is not a stub
   l: locator
   ret: bytes
                                                                                    10
   switch(proc)

     case 'f'
        % function 'f' takes two reference arguments and returns a reference
        ret1: object_or_stub
        arg1 := locator_to_ptr(data[0], sender_ts)  % marshall first argument
        arg2 := locator_to_ptr(data[1], sender_ts)  % marshall second argument
        ret1 := (self.object).f(arg1, arg2))        % call function f()
        ret[0] := ptr_to_locator(ret1, source_space, msg_ts) % marshall result
                                                                                    20
     case 'g'
        % function 'g' takes no arguments and returns a reference
        ret1: object_or_stub
        ret1 :=  (self.object).g()  % call function g()
        ret[0] := prt_to_locator(ret1, source_space, msg_ts)  % marshall result

     case 'h'
        % function 'h' takes two arguments (1 integer, 1 reference) and returns an integer
        ret1:  integer
        arg1 := unmarshall_integer(args[0])      % marshall first argument            30
        arg2 := locator_to_ptr(data[1], sender_ts)      % marshall second argument
        ret1 := (self.object).h(arg1, arg2)      % call local function h()
        ret[0] := marshall_integer(ret1)        % marshall result
  end switch

  return ret
end function
```

Figure 9: Presentation Protocol, unmarshalling and dispatching an invocation. Scion's function invoke() unmarshals arguments, dispatches call to local functions and marshalls results. In this example, the target object exports three functions called f(), g() and h().

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.

3.6.1 Local Garbage Collection and Create-Create Race

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.

For instance, consider y_A in Figure 1. Its strong locator is $\{B, b\}$, meaning that y_A was created in order to unmarshall a reference to y contained in a message received from B. Suppose that y_A becomes unreachable (e.g. because the pointer in x is set to null): it is collected; a message to this effect is sent to B; in turn scion b is collected. Suppose that a duplicate of the original message (or another, delayed message also carrying the reference to y) is now received at A: y_A will be created again, but with no corresponding scion. This is an error.

We call this the Create-Create Race; the following rule ensures it cannot occur. Before collecting 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 (see Section 3.2).

Late and duplicate messages are permitted by the Transport Protocol, but only if they cannot invalidate the invariants. A stronger transport protocol delivering messages in FIFO order with no duplication would not have a create-create race.

3.6.2 Cleanup Protocol and Create-Delete Race

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

Signaling to a scion that the matching stub has been collected is complicated by a set of potential problems. First, a "deletion" message could be lost. Second, if application messages can be lost, the stub may have never been created at all. To tolerate both cases of message loss, lists of stubs which are live at some time are sent repeatedly (this is idempotent), rather than sending deletion messages. Finally, messages are asynchronous, leading to possible race conditions between scion update and deletion. We call this a Create-Delete Race; it is different from the previous one. To avoid the Create-Delete Race, a scion is removed only if it is both unreachable and there is no message in transit which may make it reachable again.

Thus a space A will periodically send to some other space B a live message containing: (i) the set of scion names, taken from the strong locators of all extant stubs at A that point at scions at B, as given by stubs_to_space_scion_set (defined in Section 3.1), and (ii) the value $threshold_A[B]$ (see Table 12).

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:

$$scion.\mathsf{stamp} \leq live\_message.\mathsf{threshold}$$

Table 12: live Message

sender_space = A	From A to B.
sender_timestamp = $stamp_A()$	
type = 'live'	List of live stubs.
threshold = $threshold_A[B]$	Sender's threshold for receiver.
live = (b, b')	Set of scions of receiver, in use by sender.

i.e. there may be no recent messages in transit carrying its location, which would make it reachable again (recall that message in transit, such that its stamp is greater than the threshold, will be dropped by the Transport Protocol; see Sections 3.6.1 and 3.2).

Essentially, we have made stub and scion deletion idempotent operations, and eliminated the race conditions, by ignoring messages arriving "too late" (Create-Create Race) and by never discarding scions that have recently updated timestamps (Create-Delete Race). 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. Any non-instantaneous transport protocol will generate the Create-Delete Race.

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.

Figure 10 lists some pseudo-code associated to the local GC and Cleanup Protocol. Function live_daemon() periodically sends live message to the other spaces. Received live messages are processed by cleanup(), using the threshold to avoid Create-Delete races. As stated in Section 3.3, stubs are collected by the local GC. Actually, stubs must be finalized before being reclaimed: the finalization function delete_stub(), applied to garbage stubs, is responsible for increasing the threshold to the unreachable stub's timestamp.

3.7 Termination Recovery

In this section we address the problems arising when a space terminates. We define space termination to mean that its local root is deleted, as well as all objects it contains. (References to deleted objects are detectably dangling; an attempt to invoke the target will raise an exception 'deleted'.) Indirect SSP chains through the terminating space must first be resolved and short-cut.

For instance, in Figure 1, suppose that space D terminates. Object t is deleted, i.e. x's reference to t becomes (detectably) dangling: if x attempts to invoke t an exception will be raised.

Suppose instead that A terminates. Then both scions b and b' have become unreachable. Eventually t will be garbage-collected.

Suppose instead that C terminates. By our rules, y is deleted but t is still reachable from x; the SSP chain from x to t must be short-cut before termination is complete.

These rules are easy to enforce when a space voluntarily terminates itself. In the case of an abrupt termination caused by a crash a protocol is needed to

```
procedure live_daemon()
   live_msg: live_message
   periodically
      foreach sp: space_name in all_spaces_set() do
         live_msg.live := stubs_to_space_scion_set(sp)
         live_msg.threshold := threshold[sp]
         transport_send(sp, stamp(), live_msg)
      end foreach
   end periodically
end procedure                                                          10

procedure cleanup(source_space: space_name, threshold: timestamp, live: scion_name_set)
requires:: threshold and live received in a live message
ensures:: all non_live scions have been deleted
   % dead_scion_list set contains scions not included in live message
   dead_scion_list: scion_name_set :=
       scion_from_space_set(source_space) − live % set difference
   % remove dead scions from scion_list
   foreach s: scion in dead_scion_list do
      if s.stamp ≤ threshold then                                      20
         delete_scion(sc)
      end if
   end foreach
end procedure

procedure delete_stub(st: stub)
requires:: st is unreachable
ensures:: st collected, threshold increased
   critical section
      threshold [st.location.strong_space] :=                          30
            max (threshold [st.location.strong_space], st.stamp)
      remove_stub(st)
   end critical section
end procedure
```

Figure 10: Cleanup Protocol: function live_daemon() sends live messages. Received live messages are processed by the cleanup() function at each target space. Function delete_stub() finalizes stubs; called by the local GC, it increases the threshold with the timestamp found in the deleted stub.

achieve the same observable effect. There are two aspects: ensuring communication, and re-establishing the existence invariants.

3.7.1 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 (in the example, if C terminates, A does a global search for $\{C, c'\}$ in order to invoke t). Thus when an object migrates, its new scion must carry with it the list of scion names under which it had been previously known: here, $\{D, d\}$ also recognizes the name $\{C, c'\}$ in response to a global search. This list will be discarded as a side-effect of collecting the scion after the short-cut protocol of Section 3.4.3.[8]

An alternative would be to assume reliable migration that ensures that all stubs leading to the object are correctly updated before committing. Scions contain enough information to propagate the new location back along all strong locator chains. This provides a solution to the problem, whilst avoiding global searches, scion name lists, and the use of unique identifiers. The backward propagation, and especially the commit, are prohibitively expensive.

3.7.2 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. If a weak locator indicates this scion, perhaps because the terminated space passed a reference to this object elsewhere prior to termination, the scion should be retained to allow recovery by scion-less stubs in the same chain.

We are contemplating a number of possible solutions. The simplest is to retain forever all scions whose source_space has not reliably short-cut 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 [6] and means that the algorithm is no longer live, but remains efficient and effective.

An alternative would make the rule for broken chains be the same as for objects: any SSP chain indirecting through a terminated space would be deemed dangling. Care must be taken not to reuse the name of a scion which might have been referenced from the terminated space (and hence might be the target of

[8] Care must be taken in space C not to reuse name c' as long as $\{D, d\}$ still recognizes it.

a weak locator elsewhere). This solution 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 carries the space and scion identifiers for every scion encountered during the message's journey. The target then performs an exhaustive search, presenting this list to each space, requesting it to update any relevant 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-cutting indirections. When a strong and weak locator disagree, a null invocation is made, thereby updating the locators. This immediately solves the problem if objects cannot migrate, at a cost in additional messages. The null invocation can be performed either immediately upon receiving a reference, or after a time-out, or by a background dæmon.

4 Analysis

A proof of the safety of the algorithms, and a discussion of the circumstances under which it exhibits liveness, are in preparation. It is shown that the protocols do maintain the invariants of Section 2.4, and that GC never collects non-garbage objects since the scions form a superset of the existing stubs and act as local roots for the LGC. Furthermore we show that all acyclic garbage is eventually collected.

The rest of this section analyzes the costs of our protocols and their tolerance to a the fault conditions that are common in actual distributed systems. Memory usage assumes stock 32-bit hardware.

4.1 Failures

The failure assumptions are weaker than in most comparable material [4, 5, 6, 15, 16, 21]. Messages may be duplicated, 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. By minimizing the assumptions we ensure our mechanisms are applicable to a wide range of systems, and we allow the use of cheap, lightweight, lossy protocols. We do not impose the use of reliable or causal protocols, of clock synchronization, nor of any global algorithm.

Processors are fail-stop. It is assumed that messages are not corrupted and that each timestamp generator produces increasing values.

4.2 Costs

We consider the costs of our protocols in terms of messages, of CPU time and of memory space. This analysis omits the cost of recovery after a crash. A

[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.

later section (Section 4.3) considers some savings allowed by strengthening the assumptions.

As a comparison baseline, 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.

4.2.1 Number and Size of Messages

Most importantly, SSP Chains require almost no foreground messages in addition to those of the application. The only extra foreground message exchange is the location-exception/invocation-retry when a target has migrated.

Background messages arise in the Cleanup Protocol. Also, the Transport Protocol avoids race conditions by occasionally dropping an application message. It is up to higher, unspecified levels of protocol to detect message loss and to resend if necessary.

Re-establishing the invariants after a failure may be done entirely in the background; re-establishing communication may require an foreground exhaustive search.

The marshalled form of a reference, as held in messages, consists of a locator, approximately 16 bytes long, i.e. 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 reduces the average cost of a search. There is no such cost with locators.

As a possible optimization, chain short-cutting may be performed early, without awaiting invocations. A background dæmon can scan the stubs looking for differences between strong and weak locators. When it finds such a difference a null invocation can be performed, triggering the actions described above. This has a cost both in local CPU time and in background messages.

4.2.2 CPU Time

The small CPU and memory overhead is more than justified by the added functionality and by the message savings.

Reference marshalling and unmarshalling require searches for existing scions and stubs, prior to creating new ones. A similar cost arises when short-cutting indirect references. Auxiliary hash tables and lists, reduces these overheads whilst increasing space usage. Passing a UID typically consists of a simple bit-wise 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 activities 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.

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 (including the auxiliary lists).
- There is a single scion per object for each remote space that contains references to that object; it occupies 20 bytes (including the auxiliary lists).

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, causing an accumulation of scions.

Another tradeoff is possible in the implementation of the message threshold. One could replace the timestamp-per-stub with a single, per-space highest received timestamp vector hts_A. Such a change might save space at the expense of a coarser granularity of windowing on the threshold contents. On discarding a stub, rather than closing the window only on messages generated prior to the last action involving this stub, all earlier messages from the indicated space would be discarded.

The threshold mechanism can be tuned so that applications perceive different effects if so desired. An application requiring ordered channels can simply update the threshold vector whenever a message is passed up to the application. A different application which tolerates out-of-order messages can rely solely on the thresholding provided by our protocols.

4.3 Possible Simplifications

SSP chains are a cheap, efficient and robust mechanism for object identification and location. The mechanism was explained in the context of weak assumptions, in order to emphasize its broad applicability. We now strengthen the assumptions and show some possible simplifications, which retain the general flavor of SSP chains.

The mechanisms that appear costly, and are therefore prime targets for savings are: location exceptions; live messages; the timestamps and threshold vector; and the multiple scions per target.

If one adds the assumption that objects do not migrate, the extra message exchange of the location_exception message and invocation retry is no longer necessary in the invocation protocol.

Let us now assume a stronger, reliable transport protocol (like TCP or ISO-TP) such that messages are not lost or duplicated, and arrive in FIFO order.

Table 13: Execution Times

Application	CPU time in seconds						Ratio	
	No DGC		IRC		SGP		SGP/IRC	
(sort 100)	3.8	3.2	4.7	3.9	5.5	4.1	1.17	1.05
(sort 200)	5.6	4.4	6.7	5.2	8.1	5.9	1.20	1.12
(mult 20 20)	11.1	7.8	12.1	8.7	13.5	9.8	1.12	1.12

In this case, the Create-Create Race does not occur (Section 3.6.1); therefore the threshold mechanism can be eliminated from our transport protocol (see Section 3.2). Also, the periodic background live message of the Cleanup Protocol (see Section 3.6.2) is not necessary. Instead, whenever a stub is collected by its LGC, a single-shot delete message is sent to its matching scion. The Create-Delete Race (see Section 3.6.2) remains possible; however a simpler solution will now work. Each scion contains an integer count of the number of times the corresponding reference has been sent to the source space for this scion. The sender of a copy of the reference increase its scion's count. The receiver of a delete message decreases the count; deallocate a scion when its count reaches zero.

It is tempting to get rid of the multiple scions standing for a single target (one for each source space) and replace them with a single scion, containing a common reference count. However, without the source of an incoming reference, the recovery protocol after a crash is seriously complicated, because it is not known which references are actually affected by the crash. We advise this simplification only among processes of a single machine, the only case where the no-crash assumption is reasonable. At this point, our cleanup protocol is reduced to reference counting.

4.4 Measured Performance

We have prototyped an earlier version of our protocol, called SGP [30], on the distributed Lisp Transpive [21]. This version lacks weak locators, uses a message protocol rather than call-reply and uses an *hts* timestamp vector instead of timestamping stubs. A detailed account and analysis of this experiment may be found elsewhere [23].

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, show extremely low variance. Our experiments were able to test resilience to message loss but not to crashes, due to lack of a fault-tolerant application. Furthermore, we were not able to quantify how conservative or how scalable our protocol is.

Table 13 shows local execution times. Our implementation is on average 10% slower locally than IRC and 20% slower than with distributed collection

Table 14: Message Overhead

Application	# Control Messages IRC		SGP		Ratio SGP/IRC	
(sort 100)	31	28	10	8	0.32	0.28
(sort 200)	41	39	10	8	0.24	0.20
(mult 20 20)	101	96	20	18	0.20	0.19

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 14 compares message costs. As can be seen, SGP is considerably more message-efficient than IRC. This is essentially because IRC sends individual delete messages, whereas we batch deletions into periodic live 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 [20, 25, 29] 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.

Our reference mechanism owes much to the links of Demos/MP [24] and the forwarders of Emerald and Hermes [4, 5]. In contrast to their proposals, our references are intimately associated with GC. This is made possible by the invariants maintained by our protocol.

Fowler [8] chains forwarders to provide continuous access to highly mobile objects. Fowler analyzes three alternative location protocols (distinguished in how they short-cut 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. 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.[10] Unlike those approaches, however, we maintain one different scion per source space, which permits us to tolerate message loss and crashes while avoiding the dangers of duplicated delete messages.

Dickman's [6] optimised WRC (oWRC) is based on the invariant that an object's own weight ("total weight") is greater or equal to the sum of all remote reference weights (partial weight). This weak invariant allows oWRC to tolerate message loss; but duplicated messages remain problematic. The safety property of oWRC is maintained in spite of possible loss of weight-update messages. As a consequence, the algorithm is not live, since an object can only be reclaimed if its total weight is strictly equal to the sum of all partial weights. The oWRC algorithm is used in conjunction with a global mark-and-sweep to collect garbage objects not reclaimed by oWRC.

Piquer's Indirect Reference Counts (IRC) [21] also improve on WRC. IRC eliminates increment messages (making it resilient to out-of-order messages or third party concurrency and eliminating underflow problems), by managing creation and duplication of remote pointers locally. Remote pointers have a field called indirect pointer, which is used only by the distributed GC, and references either the target object, or another (indirect) remote pointer. Mutators never use indirect pointers but direct pointers to access objects in a single hop. The use of two kinds of pointers allows the creation of chains of remote pointers without penalising access to objects. In contrast to Piquer, we use weak locators not only to optimise access, but also to short-cut redundant indirections.

Mancini and Shrivastava [18] 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 [3] or Schelvis' cycle-detection technique [26]. We will discuss below only Ladin's logically centralized algorithm, Hughes' timestamp algorithm, and Lang's dynamic grouping technique.

Lang *et al.* [15] combine a distributed reference count and mark-and-sweep within groups of nodes. These groups are created and disbanded dynamically. 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. A failure during a collection causes group re-organisation 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; on an unreachable object the mark will not change. Objects

[10] More specifically on Weighted Reference Counts or WRC [1, 33], which distribute part of the counting to the pointers themselves, avoiding some messaging and some race conditions.

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 [16] describe a fault tolerant distributed garbage detector based on their highly available logically-centralised service. This service is physically replicated, hence achieving high availability and fault-tolerance. All objects and tables are assumed to be backed up in stable storage. Clocks are synchronised and message delivery delay is bounded. These assumptions allow the centralised service to build a consistent view of the distributed system. 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 (including dead cycles) with a standard tracing algorithm. The centralised service informs LGCs of accessibility of objects.

In a later paper [13] Ladin and Liskov simplify, and correct the deficiencies of, the above proposal, by using Hughes' algorithm and loosely synchronised local clocks. Hughes' algorithm eliminates inter-space cycles of garbage, thereby eliminating the need for for an accurate 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 SSP chains, a scalable, location-transparent mechanism for referencing and invoking objects in a distributed system. SSP chains have well-defined semantics, captured by a set of invariants, that hold (or can be re-established) even in the presence of message failures and crashes. We also presented a distributed GC protocol, a variant of reference counting, tolerating the inconsistencies allowed by the SSP chain invariants. The global cleanup protocol can be combined with any reasonable local garbage collection algorithm.

The mechanisms presented here are effective, inexpensive, and straightforward; indeed, they are based on a novel combination of well-known techniques. They require no global search or synchronization and put very weak requirements on the underlying messaging protocols: cheap, unordered, unreliable messages are consistent with our approach. The key enabling concepts are the chaining of stub-scion pairs, the combination of weak and strong locators, scions or inverse reference lists (as opposed to strict reference counts), and the use of timestamps to support idempotent deletion. In conjunction a conservative creation policy, these provide a fault-tolerant and efficient mechanism.

The current specification suffers from some limitations. Only acyclic garbage is collected; it will be necessary to extend the mechanisms to collect distributed cyclic garbage. Although the main-line protocol is scalable, recovery after a crash entails global search. In order to limit the cost of search, we advocate a structure of the the universe into small groups of terminating spaces (in which exhaustive search remains realistic) connected by non-terminating gateways. However, this particular aspect 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 [17, 22, 27]. A proof that the protocols maintain the invariants is in preparation.

Acknowledgments

We warmly thank Olivier Gruber, Dennis Shasha and Pierre-Guillaume Raverdy for their helpful remarks and suggestions. We also extend our thanks to Daniel Edelson, Bernard Lang and Robert Cooper for their comments on earlier versions of this work.

References

[1] D. I. Bevan. Distributed garbage collection using reference counting. In *Parallel Arch. and Lang. Europe*, pages 117–187, Eindhoven, The Netherlands, June 1987. Spring-Verlag Lecture Notes in Computer Science 259.

[2] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), February 1984.

[3] 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.

[4] 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.

[5] 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.

[6] Peter William Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures.* PhD thesis, Darwin College, U. of Cambridge, Cambridge, England (GB), March 1992.

[7] 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.

[8] 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.

[9] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Programming Languages Design and Implementation*, number 24(7) in SIGPLAN Notices, pages 313–321, Portland OR (USA), June 1989. SIGPLAN, ACM Press.

[10] Olivier Gruber and Laurent Amsaleg. Object grouping in Eos. In Tamer Özsu, Umeshwar Dayal, and Patrick Valduriez, editors, *Distributed Object Management (Int. Wkshp. on)*, Edmonton (Canada), August 1992. Morgan Kaufmann.

[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] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

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

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[15] 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.

[16] 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.

[17] Julien Maisonneuve, Marc Shapiro, and Pierre Collet. Implementing references as chains of links. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 236–234, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Computer Society Press.

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

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

[20] 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.

[21] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 150–165, Eindhoven (the Netherlands), June 1991. Springer-Verlag.

[22] David Plainfossé and Marc Shapiro. A distributed GC in an object-support operating system. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 221–229, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Computer Society Press.

[23] David Plainfossé and Marc Shapiro. Experience with a fault-tolerant garbage collector in a distributed Lisp system. In *Proc. 1992 International Workshop on Memory Management*, pages 116–133, Saint-Malo (France), September 1992.

[24] 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.

[25] 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.

[26] Marcel Schelvis. Incremental distribution of timestamp packets: a new approach to distributed garbage collection. In Norman Meyrowitz, editor, *OOPSLA'89 Conf. Proc.*, number 24(10) in SIGPLAN Notices, pages 37–48, New Orleans, LA (USA), October 1989. ACM Sigplan, ACM.

[27] 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.

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

[29] 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.

[30] 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.

[31] Hervé Soulard and Mesaac Makpangou. A generic FO-structured framework for persistence support in distributed settings. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 57–65, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Computer Society Press.

[32] Sun Microsystems. *Network Programming*, May 1988. Part Number 800-1779-10.

[33] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, Eindhoven (the Netherlands), June 1987. Springer-Verlag.

[34] Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management, Int. Workshop IWMM*, volume 637 of *LNCS*, pages 1–42, Saint-Malo (France), September 1992. Springer-Verlag.