A Survey of Distributed Garbage Collection Techniques*

David Plainfossé[1] and Marc Shapiro[2]

[1] ILOG, 2 av Galliéni, B.P. 85, 94253 Gentilly Cedex, France
e-mail: David.Plainfosse@ilog.fr
[2] INRIA, Projet SOR, B.P. 105, 78153 Le Chesnay Cedex, France
e-mail: Marc.Shapiro@inria.fr

Abstract

We present the spectrum of distributed garbage collection techniques. We first describe those *reference counting-based* techniques and compare them, in particular with respect to resilience to message failures. Reference counting-based techniques are *acyclic* since they are unable to collect cyclic data structures. We then describe a number of *hybrid* schemes improving over distributed reference counting algorithms in order to collect cyclic garbage. We then describe *tracing-based* techniques derived from uniprocessor tracing-based techniques. Finally, we discuss the pros and cons of each technique.

1 Introduction

Many programming languages [Weis, 1990, Cardelli *et al.*, 1988] provide garbage collection to automatically deallocate inaccessible objects. Garbage collection is extremely useful, as it simplifies the programming model, therefore freeing valuable programmer time, while avoiding bugs and memory leaks which are notoriously hard to track and repair. As any C [Kernighan and Ritchie, 1978] or C++ [Stroustrup, 1991] programmer can witness, the manual management of a dynamic heap is complex and error-prone. If the heap is shared by many applications written by different programmers, if it is accessed in parallel, if it also includes disk storage (as in persistent object systems) and distributed access (as in distributed systems), then manual resource management is simply out of the question.

Garbage collection has recently become of increasing interest in distributed systems [Lang *et al.*, 1992, Hughes, 1985]. The motivations for such a service

*Published in: International Workshop on Memory Management, Kinross, Scotland (UK), September 1995.

are numerous. First, transparency: just as modern distributed systems support transparent, uniform placement and invocation of both local and remote objects [Shapiro *et al.*, 1992], so should they also support transparent object management, including reclamation. Second, storage management is a complex task, not to be handled by users.

Distributed garbage collection is even harder than local garbage collection because the local collectors must be coordinated, to consistently keep track of changing references between address spaces. This consistency problem is further complicated by the common failures of distributed systems such as lost, duplicated, or late messages, and crashes of individual spaces.

Distributed garbage collection poses a challenging problem: reclaiming all kind of data structures while achieving efficiency, scalability and fault-tolerance. A number of proposals have attempted to design a distributed GC that fulfills all these requirements. The great number of incomplete proposals reflects how difficult the challenge is.

Most techniques only addresse part of the issues. The key reason is that adapting uniprocessor algorithms to a distributed environment is not straightforward. Tracing requires costly termination mechanisms, whereas reference counting is defeated by common message failures.

This paper is organised as follows. Section 2 first introduces our object model. Section 3 describes the reference count-based approach. In particular, we compare those techniques according to their resilience to message failures. Such counting-based techniques are unable to collect cycles of garbage and must assume that they are rare enough to minimize memory leakage. A number of hybrid proposals are explained in Section 5 which combine counting-based techniques with a global (tracing-based) technique. Section 6 surveys some enhanced techniques well suited to distributed settings. Section 7 sums up our conclusions and proposes taxonomy of the reviewed techniques.

2 Model

Spaces.

We consider a distributed system partitioned into disjoint *spaces*. We use the abstract term "space" to avoid committing to a particular implementation. For instance, at the lowest level 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 the Internet.

In our model, spaces cannot directly access object through virtual memory address due to protection mechanisms. Such model raises new issues which are not addressed in this survey (see [Ferreira and Shapiro, 1994] for a novel GC algorithm dedicated to distributed shared memory).

Spaces interact with each other by message passing using potentially unreliable communication channels. Consequently, messages may *fail* due to: loss, duplication, delayed or out of order delivery.

Spaces may *fail* due to software or hardware problems. We only consider fail-stop spaces: failed space does not sends messages. Spaces may also *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.

Exit and Entry Items.

As a side effect of marshalling results and arguments of invocations, spaces may exchange references to objects. Consequently, an object involved in several remote invocations may be referenced from a number of remote spaces. Such remote references are created when the reference to an object crosses the space boundaries.

A space contains passive objects. Objects carry references to other objects, possibly across spaces boundaries. In the remainder of this chapter, a *reference* always means a reference to a remote object, whereas the word *pointer* is used when the reference is local.

Representation of Remote References.

A reference is composed of a local pointer, an *exit item* and an *entry item*. The local pointer points to an exit item which in turn remotely refers to an entry item. The exit item embodies at least one *locator* which refers to a remote entry item. The entry item holds a local pointer to the *public* object. Such public objects can be remotely invoked through the remote reference, as opposed to *local* objects which are only pointed locally. References are created as a side effect of sending messages.

The space that contains a public object is called its *owner*. Other spaces, known as *clients*, have references to that public object. The rôles of a client and an owner are specific to a particular object: the owner of an object may well be the client of another one. Clients and owner of an object run in different address spaces on the same machine or on different machines, and communicate solely by message passing.

2.1 Operations on References

As a result of program activity, the distributed reference graph changes dynamically. Here is the list of valid operations on references.

Reference Creation.

An owner space sends to a *receiver* space a reference to an object. Upon receiving the message containing that reference, the receiver space becomes a client of the remote public object. If the object was local it becomes public. A reference creation is not an unique operation. In other words, a reference creation may involve an already public object. Object allocation and reference creation are independent operations. Figure 1 illustrates creation of a reference to a local object v, owned by space B. Upon sending the message containing that reference, space B creates a local entry item b to prevent v from being reclaimed. Upon reception of the message, space A installs a exit item v_A initialised with the locator found in the message.

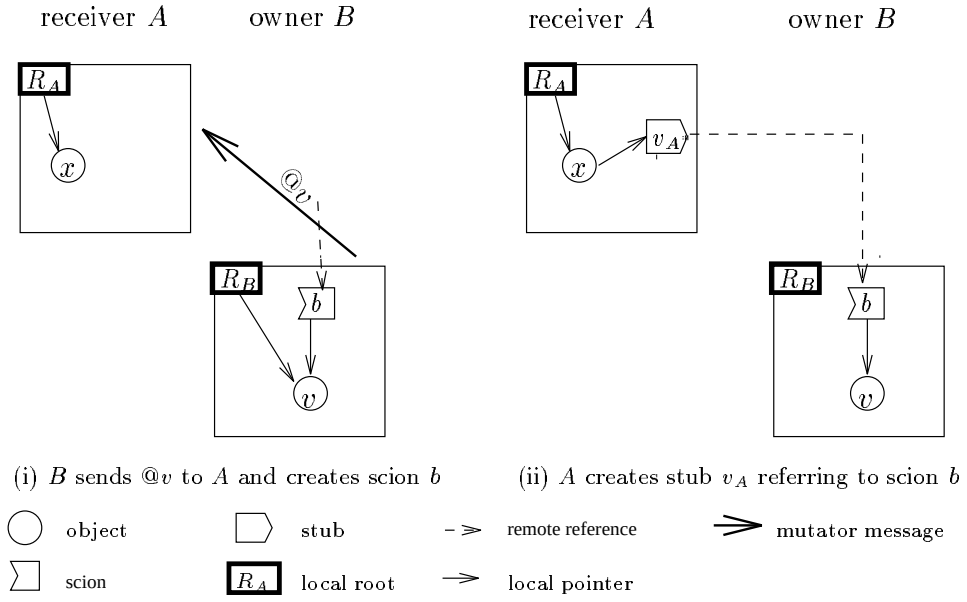
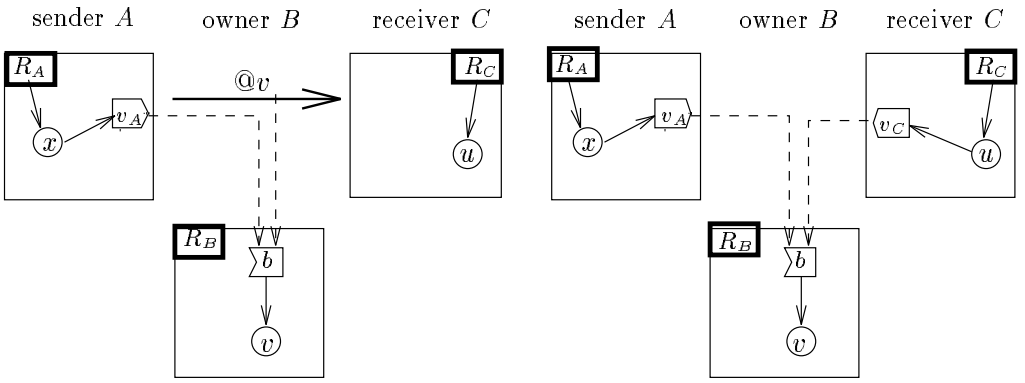


Figure 1:
Remote reference creation.

Reference Duplication.

A *sender* space, client of a remote public object, sends to a *receiver* space a reference to that object. The receiver space becomes client of the (remote) public object when it has processed the message containing that reference. In contrast to reference creation, the owner (space) is not involved in the reference duplication and therefore is not aware of it. Figure 2 illustrates the duplication of a reference to public object v, between spaces A and C. Upon receiving the message containing that reference, the receiver space C installs a exit item v_C initialised with the locator to entry item b found in the message.



(i) A duplicates reference @v to C (ii) C receives reference @v and installs stub v_C

Figure 2:
Remote reference duplication.

Reference Deletion.

Locally a client space deletes a reference it owns. This operation can be manual or automatic. Usually, the local exit item part of that reference is automatically collected by local garbage collector. Reference deletion may potentially lead to creating garbage. Thus, the rôle of the distributed garbage collector is to occasionally inform the owner which public objects are no longer remotely referenced. Figure 3 illustrates the deletion of the reference in C to public object v located on space B. First, space C deletes its local exit item v_C and afterwards sends a *delete* message to owner space B. Upon processing that *delete* message, B reclaims the garbage entry item b. Afterwards, B triggers a local GC and reclaims object v.

3 Techniques Based on Distributed Reference Counting

We now look at the problems of reference-counting distributed GC and some solutions found in the literature. Compared to uniprocessor GC, new problems appear because the universe is partitioned into separate spaces communicating through messages. If messages are not delivered reliably in their causal order then maintaining the reference counting invariant is problematic as explained in Section 3.1.

3.1 The Distributed Reference Counting Problem

The naïve extension of reference counting to distributed systems keeps a reference count with each public object. The corresponding entry item contains that reference count, which is updated on each reference duplication or creation. Upon

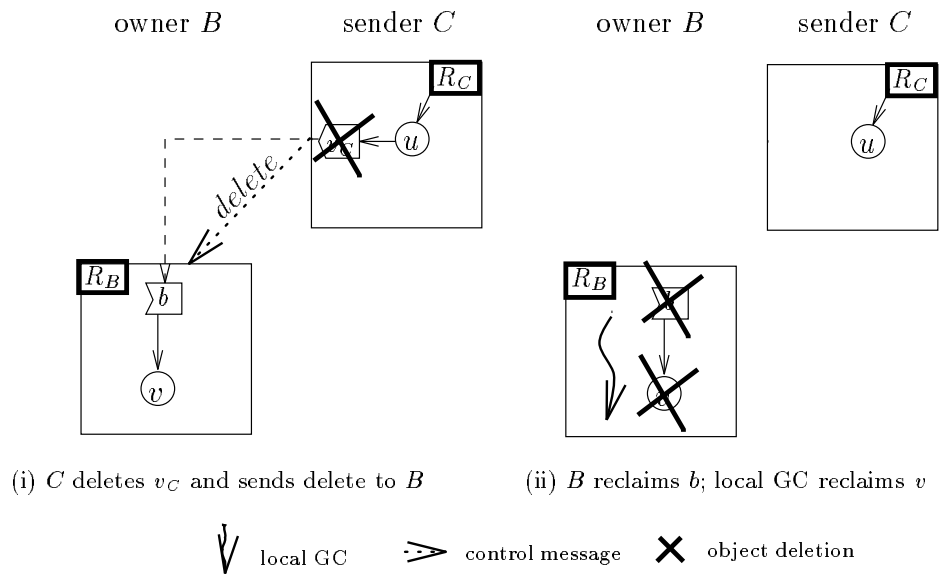


Figure 3:
Remote reference deletion.

duplication or deletion of a reference, a client space informs the owner's entry item to update the counter, by sending an extra *control* message to the owner.

Sending a control message upon each reference operation increases communication overhead significantly. Furthermore, those control messages must be delivered reliably, without loss or duplication to preserve the reference counting invariant. Finally, control messages should be delivered to their destinations in causal order (as defined by Lamport [1978] to prevent race conditions.

Figures 4 and 5 show two examples of such race conditions between increment and decrement messages. Figure 4 sketches a decrement/increment race, and Figure 5 an increment/decrement race. The former can happen when the receiver of a reference is responsible for sending the control message, whereas the latter can occur when sending the control message is the the sender's responsibility.

The decrement/increment race can occur when a sender space duplicates a reference and deletes it immediately after. For instance, in Figure 4, space A holds a reference to object v located in space B. Space A sends that reference $@v$, to C and immediately deletes its own pointer to v. Upon receiving the message containing the reference $@v$, space C installs it and consequently sends an increment message to B. Upon deleting its own reference to v, space A sends decrement message to B. If the decrement message from space A arrives to B before the increment message from C, then the corresponding counter of exit item v exit item will drop to zero. Since the object v is not reachable from the local root R_B, then the object v will be reclaimed prematurely.

The increment/decrement race can take place when a sender sends a reference to a receiver that immediately deletes it. In Figure 5, space A sends a reference $@v$

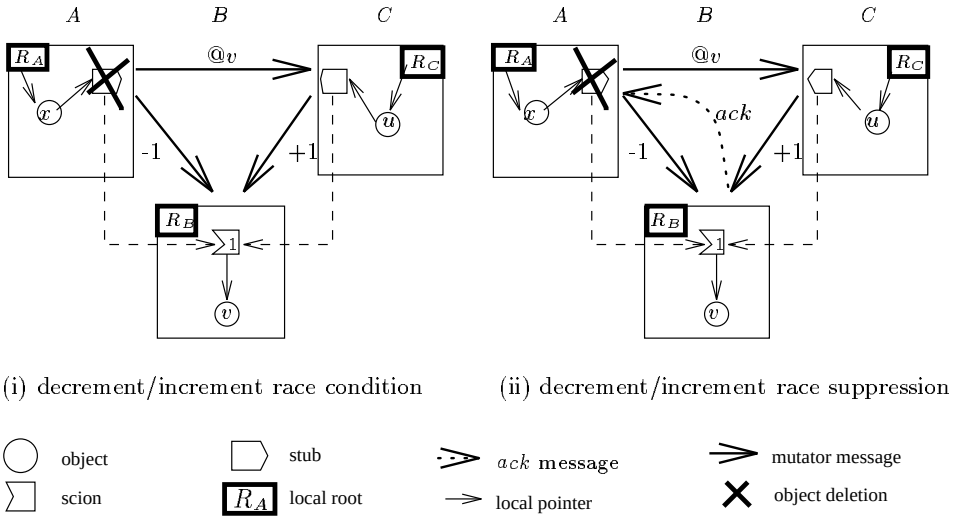


Figure 4:
Race conditions between *decrement and increment* messages.

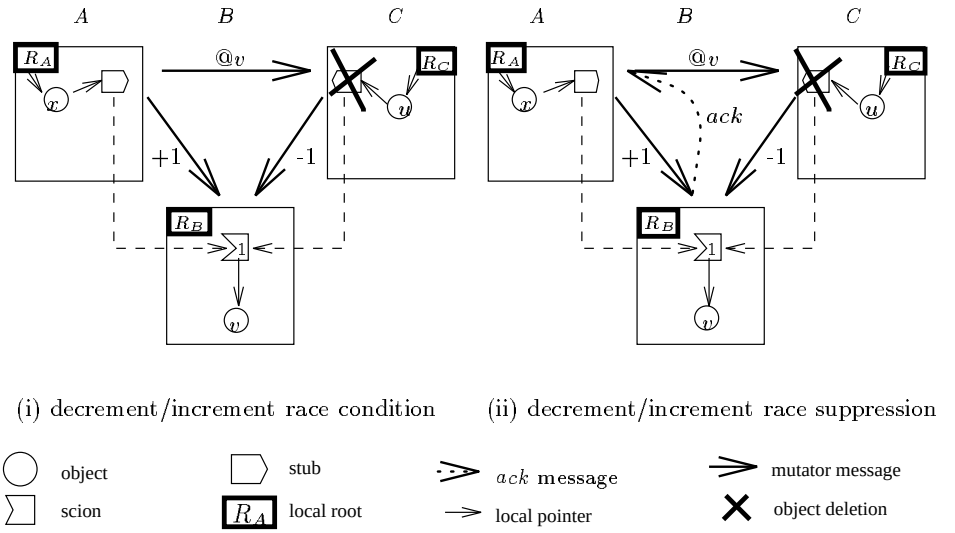


Figure 5:
Race conditions between *increment and decrement* messages.

to space C and consequently an increment message to space B. Upon receiving the message containing the reference $@v$, the target space does not install it and therefore sends a decrement message to C. If that decrement message arrives first at C then the corresponding counter drops to zero causing the unsafe reclamation of object v.

One way to suppress those potential race conditions is to acknowledge each increment message before sending a reference. For instance, in Figure 5-(ii), the increment message +1 is first sent to owner space B. Upon receiving this increment message, space B updates the corresponding $counter_v$ and then sends back an acknowledgement message *ack* to space A. Space A waits for this *ack* message before sending the reference $@v$ to B. The *ack* message acknowledges that the increment message has been received and computed on space B. Therefore, this first increment message send cannot be involved in a race with any subsequent decrement message from space C.

In spite of acknowledged messages, the naïve extension of reference counting is still not resilient to message failures because increment and decrement messages are not idempotent. Race conditions can be eliminated thanks to acknowledgement messages at the expense of a significant overhead in message traffic for each reference duplication. A number of adaptations improve resilience to either message failures or race conditions. Among others, we describe two major enhanced reference counting techniques (explained in Section 3.5) which improve message failures or race conditions (explained in Section 3.2).

3.2 Weighted Reference Counting

Bevan [1987] and Watson and Watson [1987] independently proposed the use of Weighted Reference Counting (WRC) as an efficient alternative to naïve reference counting technique. Their technique eliminates increment messages and therefore the potential race conditions.

Each reference has two weights: a *partial weight* and a *total weight*. A exit item contains a partial weight whereas the corresponding entry item contains both a *partial* and a total weight. The total weight is kept unchanged upon reference duplication or creation. It is decreased upon reference deletion.

Similarly, upon reference creation, an entry item is allocated and its total weight is initialised with an even value greater than zero and the partial weight contains in the entry item is initialised with half of the total weight. Each time a new reference to the same object is created, the entry item's partial weight is halved and the remaining half is sent along with the message.

Upon duplicating a reference, the partial weight contained in the exit item is halved and the remainder is used as an initial value of the new partial weight. Upon reception of a reference in a message, the corresponding exit item is installed with the partial weight found in the message.

Each time an exit item is discarded the partial weight contained in the remote

exit item is sent in a control message to the owner and decremented from the partial weight contained in the entry item. Weight management ensures that in the absence of message failures, the following invariant is maintained for any public object v:

$$total\_weight_v = \sum partial\_weight_v \qquad \text{(a)}$$

Conversely, an object becomes local again (*i.e.,* it is unreachable from any remote space) when the total weight and the partial weight contained in a single entry item are equal.

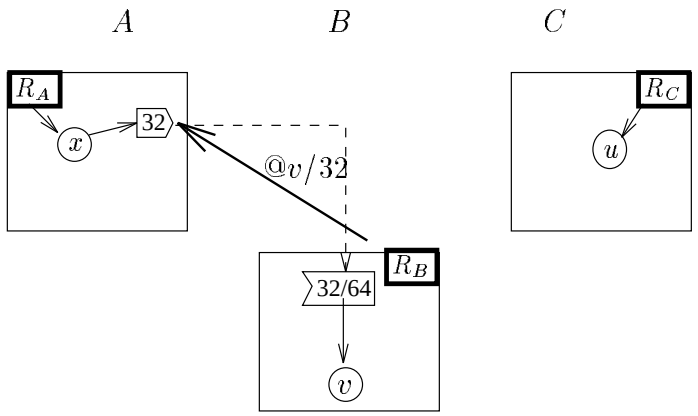
Figure 6 represents a public object v located on space B. Figure 6-(i) illustrates sending a reference $@v$ to space A. The corresponding entry item is initialised with a total weight equal to 64. This total weight is halved and one half is stored in the entry item and remainder half is sent along with message. Upon receiving the message a new exit item is created and initialised with the weight found in the message, *i.e.* 32. Figure 6-(ii) illustrates duplication of a reference to v between spaces A and C. Upon sending the reference, the corresponding exit items weight is halved, down to 16, and the other half is sent along with the message. Upon receiving the message containing $@v$, space C installs an exit item with a partial weight initialised to 16. Figure 6-(iii) shows the decrement message sent to space B when an exit item is discarded on space C. The decrement message contains the partial weight found in the discarded exit item, *i.e.* 16. Upon receiving that decrement message the partial weight found in the message is used to decrease the total weight stored in the entry item. In Figures 6-(iii), the total weight is equal to 28 after being decreased by 16. Figure 7-(iv) shows the state of the partial weight and total weight after the last reference to v has been discarded. Notice that the partial weight and total weight are now equal and object v is now garbage.

Bevan [1987] proposes to use partial weights that are powers of two, in order to store only the exponent in exit items. For instance, if all initial weights are ≤ 128, then only 3 bits are necessary to store partial weights. Total weights remain in standard binary representation in order to support arbitrary decrement messages.

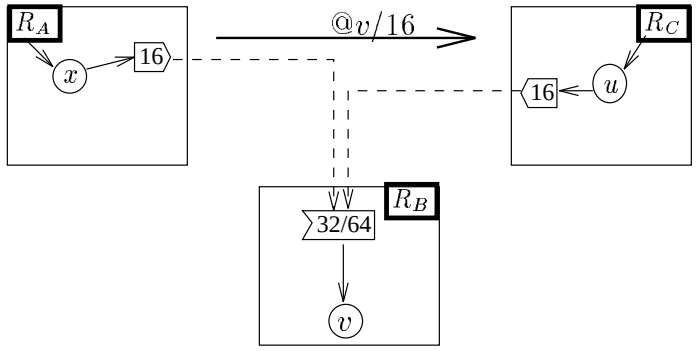
3.3 Shortcomings of Weighted Reference Counting

The main drawback of Weighted Reference Counting is that an initial weight of 2^k can only be duplicated k times before it falls to 2^0 and cannot be split any further.

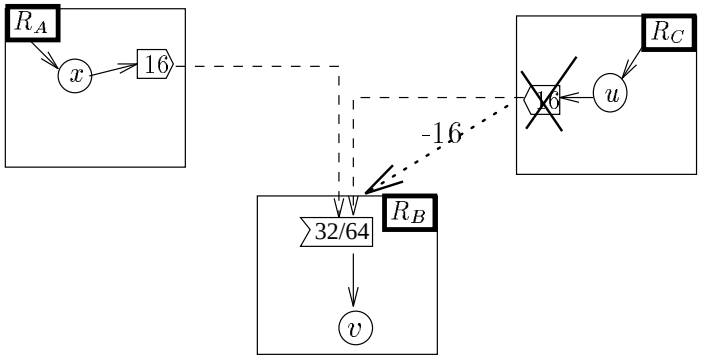
Several techniques have been proposed to overcome this problem. A trivial solution consists in adding a same amount to the total weight and the partial weight. This way, a sender sends a control message to the owner space containing that amount. Upon receiving this message the owner adds its amount to the corresponding total weight and acknowledges the control message, which allows the sender to increase its partial weight by the same amount.



(i) B creates a reference to v with a partial weight equal to 32



(ii) A duplicates to C a reference to v with a partial weight equal to 16



(iii) C discards its own reference to v

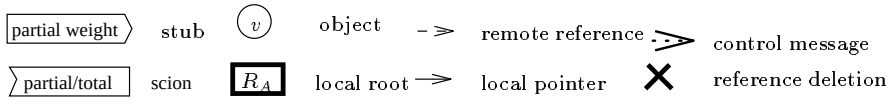
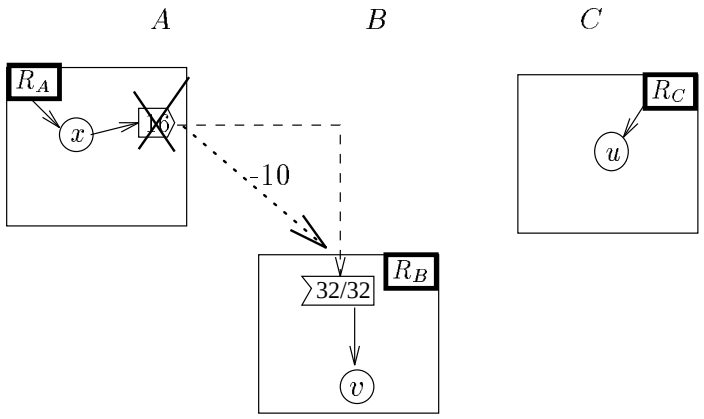
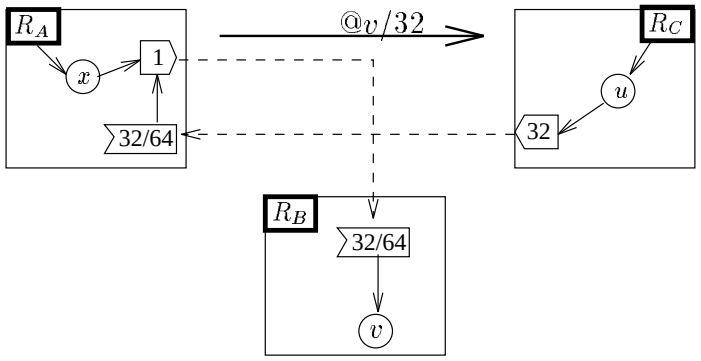


Figure 6:
Weighted Reference Counting.



(iv) A discards last reference to v; total weight is decreased from 64 to 20



(i) A creates an indirect reference upon duplication of @v

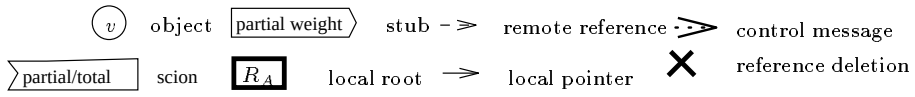


Figure 7:
Indirect reference creation with Weighted Reference Counting.

Another solution, suggested by Watson and Watson [1987] avoids the control message by creating an *indirect entry item* in the sender space. This allows to locally duplicate new references with a partial weight that matches the indirect entry item's total weight. Such references do not refer directly the object but to the indirect entry item. Figure 7 shows such a situation, where the partial weight on space A has dropped to one. In order to duplicate a reference to v between spaces A and C, an indirect entry item is created on space A, initialised to a total weight of 64. Upon receiving the reference an exit item on space C is allocated and initialised to refer indirectly to object v through space A. Hence, each access to t from u will necessitate two hops, doubling the number of messages sent. Furthermore, any failure of space A will prevent access to object v from space C.

Rudalics [1990] exhibits a domino effect with indirection entry items, due to the absence of short-cutting indirections. Once an indirection is created it remains forever. Therefore, an object t may acquire a reference on a public object through an indirection entry item even though there exists an indirection entry item local to t (i.e. located on the same space). In the worst case, this may lead to situations where an object t located on space A can only reach a public object v through a long chain of indirections looping back to space A several times.

An improvement proposed by Corporaal *et al.* [1990] uses a table instead of indirection entry items. When a partial weight drops to 2^0, an entry is created in the table. Further copies of the references continue to point directly to the object, but refer also to the corresponding entry in the table. Discarding such a reference decreases the partial weight associated with the table entry.

Weighted Reference Counting is better adapted to distributed systems than the naïve scheme since it is possible to duplicate a reference without sending a message to the owner. This improves the overhead and avoids race conditions between increment and decrement messages (see Section 3). However, weighted reference counting is not resilient to message loss or duplication.

The loss of a message containing a weight violates invariant (a). In such a case, the total weight associated with the public object becomes greater then the remaining partial weights. For instance, in Figure 6-(i), if the message containing the reference $@v$ is lost, then the sum of partial weights will be lower than the total weight. Therefore, object v won't be collected when all remote references have been discarded. Hence, message loss breaks the liveness property of the algorithm.

Message duplication is not tolerated either by Weighted Reference Counting. A duplicate message violates invariant (a) since the total weight becomes lower than the partial weights. For instance, in Figure 6-(iii), if the decrement message -16 is duplicated, then the corresponding total weight will drop to zero prematurely and v will be reclaimed, even though there still exists references to it.

3.4 Optimised Weighted Reference Counting

Dickman's [1992] Optimised Weighted References Counting (OWRC) improves on Weighted Reference Counting in two aspects: resilience to message loss and indirection entry items. Its weakened invariant is compatible with message loss. The new invariant is an inequality between $total\_weight_v$ and the sum of $partial\_weight_v$ for any public object v:

$$total\_weight_v \geq \sum partial\_weight_v \qquad \text{(b)}$$

A lost or miss-ordered message does not violate this weakened invariant. Like in Weighted Reference Counting, out-of-order message delivery poses no problem. In contrast, a duplicated decrement or duplication message remains problematic because it would make the sum of partial weights greater or equal than the total weight.

Optimised Weighted References Counting avoids of indirection entry items when partial weights cannot be split, by using a special *null weight* value. In this case, the total weight is always greater than the sum of partial weights, thus preventing the object from being reclaimed at all. Liveness of the garbage collection is not ensured for those *weak* objects.

For this reason, the authors assume that Optimised Weighted References Counting is used in conjunction with some cyclic distributed (tracing) collector, in order to reclaim garbage distributed cycles and weak objects. In addition, this cyclic global collector could be also used to (re)compute a strict invariant (a) for objects which only conform to the inequality (b).

3.5 Indirect Reference Counting

The main problem with Weighted Reference Counting is the limited number of duplications. In the worst case, an unnecessarily long chain of indirect references may be created. One solution is to extend the number of bits allocated to the weights, but this has a cost and does not completely avoid the creation of indirect references.

Piquer [1991] suggests an original solution to this problem. The key idea of this algorithm is to encapsulate in an exit item two locators rather than one. A *strong* locator refers to an entry item in the sender space. An additional *weak* locator shortcuts the strong one, and refers ahead, to a better location of the target object. In the absence of migration, the weak locator is always *accurate*: It refers to the space where the target object is located. The strong locator is used only for distributed garbage collection, *i.e.,* it prevents the target object from being reclaimed. The weak locator is used to invoke the target object in a single hop.

Duplication of a reference is performed locally without informing the owner space; as explained for Weight Reference Counting, this avoids race conditions. Upon first duplication of a remote reference a new entry item is created with a counter initialised to one. Further duplication increments the counter associated with the exit item. This exit item is connected to the corresponding entry item which refers to the target object. The weak location found in that exit item is sent along with the message containing the reference. Upon receiving this message, the space allocates an exit item and initialised it with the strong and the weak locator found in the message. Figure 8 illustrates duplication of the reference $@v$ between space A and C.

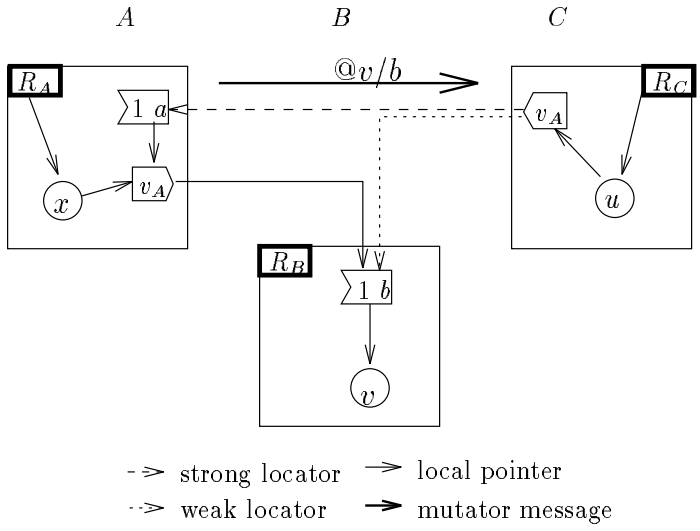


Figure 8:
Reference duplication in Indirect Reference Counting.

The duplication protocol might create an indirect reference that loops back to a space where the object is located (see Section 5.2). This undesirable effect is avoided by allocating a *unique identifier* (UID) for each object. This UID is sent along with the reference, allowing the target space to figure out if it already holds a reference to that particular object. This is not a problem in small scale networks but, keeping UIDs unique on a large scale is a challenge.

Like Weighted Reference Counting, Piquer's technique retains a lot of floating garbage due to the chains of indirect references. Goldberg [1989] partially improves this aspect by keeping a single entry item containing a counter per client space rather than one for all client spaces. This allows to reclaim faster part of the strong chain but at the expense of a significant memory overhead.

Just like the other proposals based on reference counting, Piquer's algorithm is not resilient to message failures: liveness is not preserved with message loss and safety is not preserved with duplicated messages.

4 Reference Listing

Reference listing differs from reference counting in the way entry items are managed. Instead of single entry item for all clients, containing a single counter, a space allocates a list of separate entry items, one each client space that owns a reference to a same object. Each entry item contains the identity of its *predecessor* space. (The number of entry items that point to some object is equal to the count kept in the single entry item in reference-counting techniques). Each exit item referring to an object has a corresponding *successor* entry item for that particular object. Increment and decrement messages are replaced, respectively, by *insert* and *delete* control messages. A delete message informs an entry item that it is no more referenced.

Reference listing improves resilience to message and space failures over reference-counting techniques, at the expense of a some memory overhead.

The major advantage of reference listing over reference counting is that messages are idempotent hence resilient to message failures (duplication and loss). For instance, a same delete message may be sent several times without consequences on the invariants preserved by the particular technique. If a previous delete message has already been received the following one is simply ignored; if the previous delete has been lost then the following one is processed.

However, message delivery latency may lead to reclaim unsafely a public object. If a delete message is delivered late, the result may lead to reclaim an entry item which has just been created. Conversely, a mutator message containing a reference may be delivered too late to a space. If this reference has been already reclaimed on the target space then the message should be ignored. These two race conditions may be avoided by using timestamps as done in Shapiro [1990].

Resilience to space failures relies on the ability for each (owner) space to compute the set of its clients by looking through the entry item lists so it can prompt one of these to send a live (or delete) message. Additionally, the owner may explicitly query about a particular reference that is suspects to belong to a garbage distributed cycle. Furthermore, if one of these clients is down then the owner space can take the proper decision between two alternatives:

1. to keep the objects referred to by the crashed space until it recovers.
2. to reclaim at once objects that the crashed space refers to.

The former policy assumes that entry items and exit items lists will be recovered, e.g. because they are backed up on stable storage. The latter policy assumes that a crashed space will not recover.

4.1 Stub-Scion Pair Chains

SSP Chains [Shapiro *et al.*, 1992, Plainfossé, 1994] combine distributed collection with a reference mechanism to locate remote objects. It is designed for a classi-

cal distributed system composed of spaces, *i.e.,* with no shared memory, partial failures, and unreliable and costly messages.

SSP Chains are an efficient and fault-tolerant variant of forwarders [Fowler, 1986], meaning that a reference is implemented by a chain of point-to-point links (rather than by a global identifier). A remote reference is represented as a chain of exit item-entry item 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. A migration extends the chain at one end whereas reference passing extends the chain at the other end.

A exit item encapsulates two locators: a strong and a weak one. A exit item's strong locator indicates the next entry item in the chain. The strong locator serves only distributed garbage collector purpose. Stub's weak locator shortcuts ahead the strong locator; indicating some better path to the target, if one can be known, without exchanging extra messages. Weak locators are used to invoke remote objects allowing to access an object in a single hop[1].

Sending the reference to some object x from a space A to some other space B creates a reference composed of a *entry item* into A and a *exit item* from B. The entry item associated to "exported" object x is added to the root set of A; hence protecting object x from being prematurely reclaimed. The distributed garbage protocol actually uses a conservative, fault-tolerant variant of reference counting.

When sending a reference, the application-level protocol for marshalling arguments into messages, creates an entry item; when receiving it creates an exit item. When an exit item becomes locally unreachable, the local collector reclaims it. Periodically, spaces exchange idempotent *live* messages that list the set of exit items that are still reachable; the receiver deletes the corresponding entry items that are not in the live list.

Message failures are tolerated by a conservative ordering of actions and by idempotent messages; race conditions are avoided by timestamping all messages and data structures, and ignoring messages that are inconsistent with the data structures. Crashes are tolerated by making space termination appear atomic with respect to reference exports.

When sending a reference, the application-level protocol for marshalling arguments into messages, creates an entry item; when receiving it creates an exit item. When an exit item becomes locally unreachable, the local collector reclaims it. Periodically, spaces exchange idempotent *live* messages that list the set of exit items that are still reachable; the receiver deletes the corresponding entry items that are not in the live list.

[1] In absence of migration weak locators are *exact* in that they always refers to the current location of the object.

4.2 Garbage Collector for Network Objects

Birrell *et al.* [1993] describe a reference listing technique for reclaim *Network Objects* [Birrell *et al.*, 1994]. Network Objects are fined-grained, non-mobile objects, which can be referenced from a remote space. References to Network Objects are created as a side-effect of marshalling references in remote invocations.

Reference duplication to a Network Object shortcuts at once any potential indirect reference. For instance, say client space A duplicates a reference, sending it to C. C sends at once to the owner B an *insert* message, informing B that C has just acquired a reference C. Upon reception of this insert message B, allocates a new entry item for and sends back an acknowledgement. When the acknowledgement message is delivered, C installs locally an exit item referencing to the new entry item.

The potential race condition between concurrent duplication and destruction of a same reference is avoided by preventing the remote reference from being reclaimed on the sender space. This is done by temporally pointing to the corresponding exit item from the local root of the sender. This root pointer is discarded after the *insert* operation has been completed.

Figure 9 sketches the steps of a duplication between (old client) space A and (new client) space C of a reference to t owned by space B. The following scenario starts just after space A has acquired a reference to object t.

(i) Space A in order to duplicate $@t$, makes exit item t_A reachable from its local root R_A, and sends a duplicate message to space C.

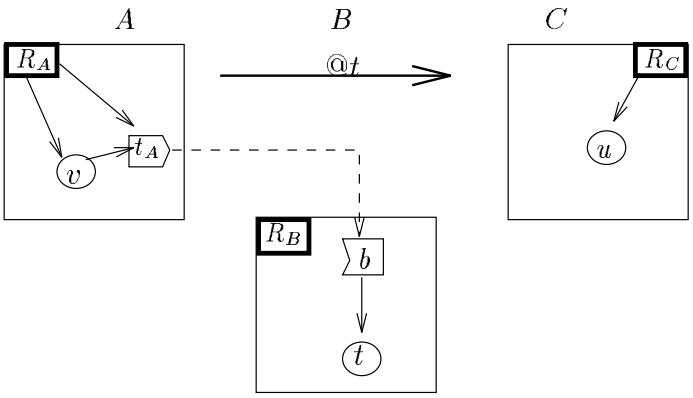
(ii) Upon receiving the message containing $@t$, space C first allocates a initialised exit item and sends an *insert* message to space C.

(iii) Upon reception of the *insert* message, B creates an entry item c for space C and acknowledges creation of entry item b'.

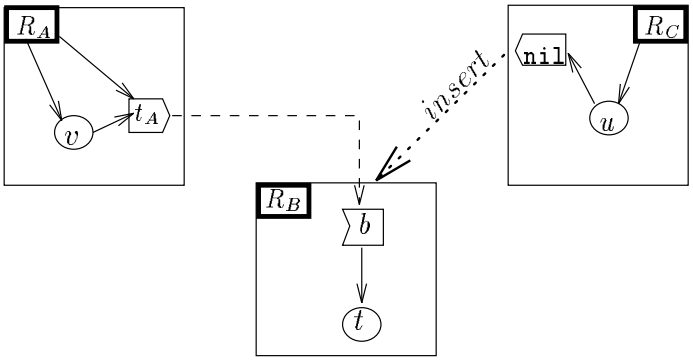
(iv) Upon reception of the *ack* message, space C initialised the exit item with locator of the entry item b' found in the message. Now that *insert* exchange is completed, space C acknowledges to space A reception of the reference $@t$. Subsequently, space A removes root pointer from root R_A to exit item t_A.

This early shortcut policy has several advantages compared to a lazier policy that would defer for instance shortcut to the previous invocation. First, no third party dependency occur. This has the nice effect that a failed space never prevents a running space from accessing an object located on a third one.

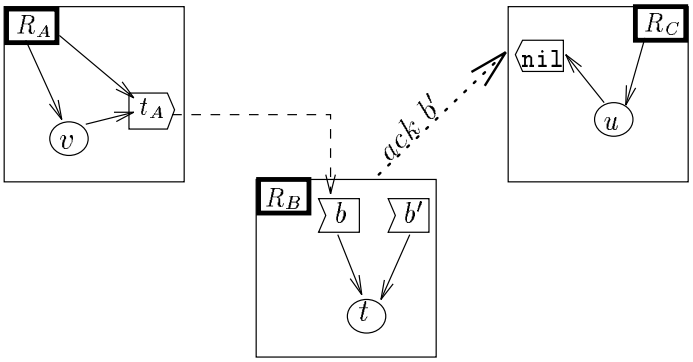
In this scheme, an entry item can only be referred from one space. This simplifies somewhat the fault-tolerance protocol with respect to space failure. When the system detects that a space has failed, it can freely reclaim entry items referred by that failed space. Space detection is handled simply by prompting clients



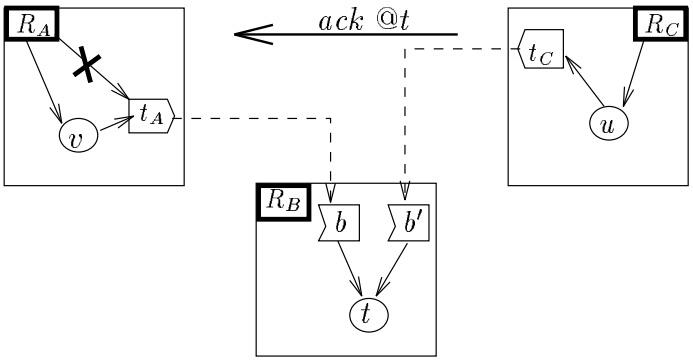
(i) A sends to C a reference to t



(ii) C sends insert message to B upon receiving @t



(iii) B creates scion b' and acknowledges it to C



(iv) C acknowldges to A reception of @t

Figure 9:
Duplication of a reference to a network object.

frequently. The client is assumed to acknowledge the ping message promptly. If not, the client is considered down and the entry items it refers to are reclaimed. This termination detection is ambiguous since, for instance, a temporary network partition cannot be distinguished from a space termination. An owner space may consider that one of its clients is down because its ping messages have simply been lost.

But this scheme has also a number of shortcomings. First, the shortcut policy should be up to applications and not wired in the system. If fault-tolerance is a crucial issue for an application, it can freely perform an insert call in order to eliminate potential indirections through third-party spaces. The second reason for choosing a lazy shortcut policy concerns the additional cost put on reference passing due to insert and acknowledgement messages. This policy doubles the number of message sent for each reference passing.

5 Hybrid Cyclic Techniques

Reference counting and reference listing collectors cannot reclaim garbage cycles spanning spaces. Therefore, such acyclic techniques only work if that cycles are rare enough to be neglected. For instance, this approach is acceptable if servers are short-lived, if sufficient memory is available to support the storage leaks and any additional paging cost due to memory fragmentation is bearable. In contrast, support of long lived-servers could suffer from cumulative storage leakage leading to memory shortage. Several improvements to reference counting techniques have been proposed.

Section 5.1 describes complementary tracing technique which basically combines reference counting based techniques with distributed tracing.

Trial deletion attempts to figure out which objects belong to a garbage cycle by relying on heuristics (see Section 5.3). Finally, object migration may be used to consolidate a distributed garbage cycle on a single space (explained in Section 5.2).

5.1 Complementary Tracing

The key idea of complementary tracing is to combine an acyclic garbage collector with a cyclic one. Usually, the cyclic distributed garbage collector is triggered at a low rate and most of garbage is assumed to be reclaimed by the acyclic one. However, to be efficient such combination relies on the assumption that global tracing frequency is low compared to the acyclic collector.

Dickman [1991] combines his Optimised Weighted Reference Counting with an (unspecified) cyclic global garbage collector. This cyclic collector is responsible for both reclaiming distributed cycles and objects Optimised Weighted Reference Counting can't collect (explained in Section 3.4). Triggering the cyclic garbage

collector is heuristic based on measurements gathered during computation. This criterion must be carefully chosen in order to optimise the cyclic collection frequency. Since the cyclic collector is not specified, it is not clear whether it runs concurrently with the mutators.

The technique proposed by Juul and Jul [1992] is concurrent and less disruptive than Dickman's proposal. A global cyclic collector based on distributed tracing colours remotely referenced objects and traces through the complete distributed graph of objects. Local garbage collectors may run in parallel with the global one by assuming that all entry items are local roots. Unlike most cyclic garbage collectors, this technique merges garbage detection with garbage reclamation. Indeed, the cyclic garbage collector does not simply discard remote references but it is also able to reclaim garbage as the local garbage collector. However, both garbage collectors are not entirely independent because of the potential overlap in their activity.

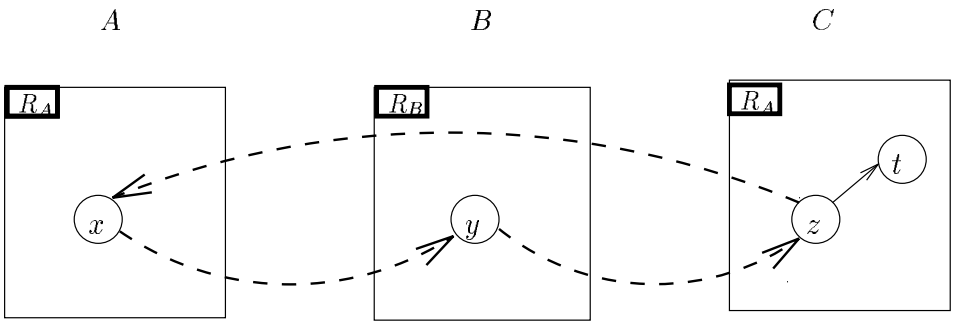
5.2 Object Migration

The key idea of the object migration technique (first proposed by Bishop [1977], is to move all objects of a garbage cycle into a single space, provided that a local tracing collector reclaims intra-space cycles. Figure 10 illustrates the *consolidation* of a garbage cycle composed. Initially (i), objects x, y and z are distributed respectively on spaces A, B and C. Step (ii) migrates object z and t to space B, short-cutting the reference from y to z to a local pointer. Step (iii) consolidates the cycle by migrating subgraph rooted by object y to the space A. The cycle consolidated in A will be reclaimed at next collection on space A provided that the local GC is cyclic.

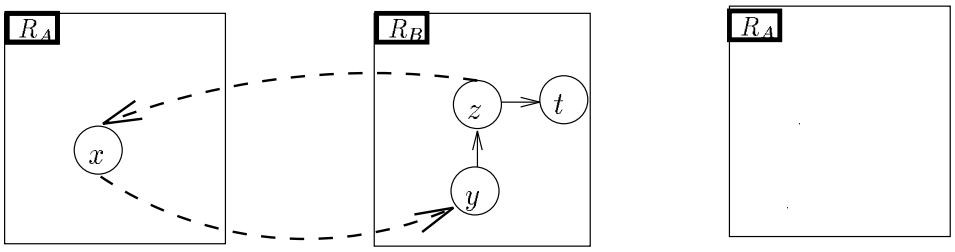
At each step, the subgraph rooted by the suspect object is migrated. For instance, Step (iii) involves migration of both objects z and t although migrating the former object would be enough. However, object t could be part of the distributed cycle if t as indicated by the dashed arrow in Figure 10-(i). Migrating the whole subgraph inspecting each object on it in order to figure out if it belongs to the cycle. However, traversing each object on the subgraph would prevent to migrate objects reachable from the local root.

An object suspected to be part of a distributed garbage cycle is moved to a client space which refers to it. This assumes that an owner space knows its clients, as in reference listing (explained in Section 4). The local collector can distinguish locally-rooted public objects from *non-local* public objects that only remotely referenced. Clearly, only the latter are potentially part of a garbage cycle. Furthermore heuristics may help to distinguish, among non-local public objects, which are more likely to belong to a garbage cycle. For instance, one that has not been invoked for a long while it is a good candidate for a migration. Note however that heuristics can fail, and non-garbage objects might be migrated sometimes. Consequently, the heuristics should taken into account several criteria, such as the number of *delete* messages received to a particular client since the reference has

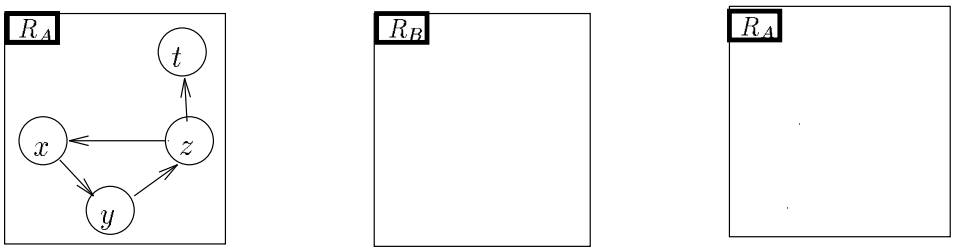
been created and the last invocation received is that object. A multiple-criterion heuristics has the effect of decreasing the number of useless object migrations.



(i) A garbage cycle distributed among spaces A B and C



(ii)The garbage cycle after z and t have migrated to B



(iii) The garbage cycle is consolidated on space A after migration of y, z and t

Figure 10:
Bishop's object migration technique.

A major problem with Bishop's migration technique is that it does not accommodate indirections well. Figure 11 illustrates this problem by showing a distributed garbage cycle composed of two objects x and y on spaces A and B, *before* and after migration of y to space A. It is obvious on Figure 11 that the situation after migration is worse than before. Indeed, the migration of y extends the references between x and y with useless indirections, preventing the consolidation of the cycle. Unexpectedly, the migration of object y does not help the

detection of the garbage cycle since it still goes through y's previous space.

However a shortcut of these useless indirections would consolidate the cycle on space A, but it requires two extra messages for each reference to figure out, for instance, that exit item t_A actually refers to a local object (i.e. located on same space A). Note that shortcut must be triggered by the migration mechanism itself since all objects on the cycle are garbage.

Another solution consists in managing entry items and exit items accordingly to an UID. The object UID is sent along with references in mutator messages upon duplication or creation of a reference. This UID found in the message is used at the receiver space to check if an entry item has been previously registered for that reference. If an entry item is found for that particular UID then there already exists a reference for that public object. The local pointer contained in the entry item is then returned, thus avoiding the creation of a loop. In Figure 11, this mechanism figures out that exit item t_A actually refers to local object x. Consequently, the reference from y to x is shortcut at once into a local pointer. However, if UIDs are useful to avoid loop creation, they have severe shortcomings with large scale system. Particularly, it is difficult to generate unique identifiers in a very large network for long-lived objects.

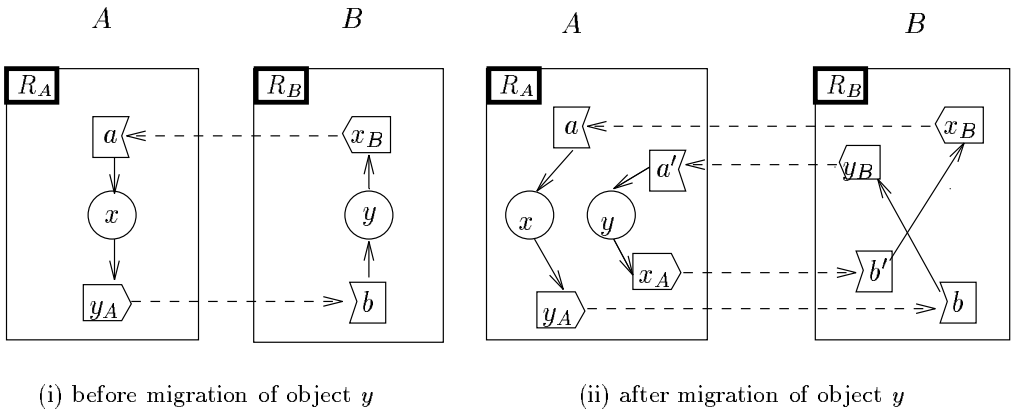


Figure 11:
Bishop's migration techniques conflicts with indirections.

Bishop's technique has other drawbacks. For instance, a heterogeneous network poses a serious problem since it is cumbersome to migrate objects between different architectures. Moreover, since migration is based on heuristic, non-garbage objects might be migrated. Moving non garbage-objects could conflict somewhat with other system components such as load balancing or clustering.

5.3 Trial Deletion

Trial deletion technique as first proposed by Vestal [1987] is based on heuristics to figure out if a cycle of garbage exists: objects suspected to belong to a garbage

cycle are used as a seed for a cycle-detection protocol. This technique essentially consists of a trial deletion of the object seeded in order to check if descendant's counts drop to zero. For this reason, trial deletion assumes that the main collector is reference counting-based. The main difficulty of this technique consist in coming up with a good heuristics to select seed objects.

5.4 Local Tracing

Lins and Jones [1991] have proposed to combine Weighted Reference Counting with mark-and-sweep in order to collect garbage cycles. The tracing algorithm does not walk the whole distributed graph but from its root, instead traces locally from an object suspected to be part of a garbage cycle. This algorithm looks for a garbage cycle each time a pointer to a shared object, say v, is deleted. Upon such deletion, the subgraph rooted by v is traced in order to decrement the object's counter (previously copied). At the end of tracing, if an object's counter has dropped to zero, it means that this object belonged to a garbage cycle and it can be safely reclaimed.

There are two main problems with this technique. First, the overhead on each pointer deletion is clearly not acceptable. Furthermore, the distributed tracing is poorly concurrent with mutator activity. If the frequency of pointer deletion is high enough, the mutator may be suspended most of the time by the cyclic collector. These shortcomings were later corrected in part [Lins, 1990] by queueing suspected objects and batching tracing on several subgraphs. Second, concurrent tracing of overlapping subgraphs necessitates either some form of locking or extra data fields to manage a counter per tracing. If the graph of objects is distributed then the algorithm requires some form of global synchronisation to prevent a space from triggering a cyclic collection until all previous ones have completed. We suspect either solutions to be probably too costly in time or space respectively.

5.5 Discussion

Bishop's migration, Vestal's trial deletion, and Lin's local cyclic RC techniques share the desirable property that only those spaces that contain parts of the garbage cycle are traversed. In contrast, complementary tracing techniques require all spaces on the network to cooperate in the collection. However, both techniques rely heavily on heuristics to select a candidate seed object. A failure of the heuristics results in wasted computation. This effect is even worse in Bishop's technique since the migrated object can still be reachable from its previous space.

Vestal's technique requires the local collector to be reference-counting, requires additional counter fields, and cannot run concurrently with other collection processing. Bishop's migration technique seems better adapted to distributed systems, but requires to shortcut indirect references, which either costs extra messages or requires unique identifiers.

6 Tracing-based Distributed Garbage Collectors

Reference counting may be made fault-tolerant and efficient, but still suffer from incomplete liveness due to an inability to detect distributed cycles. Hybrid techniques are able to collect garbage cycles but suffer from serious drawbacks. In contrast, fully tracing-based proposals are intrinsically cyclic.

We first explain why tracing cannot be directly adapted to a distributed system (explained in Section 6.1). We describe then three main tracing-based garbage collection techniques. Liskov and Ladin compute on a central space the global graph of remote references (explained in Section 6.3); Lang-Queinnec-Piquer's technique performs a global mark-and-sweep within a dynamic group of spaces (explained in Section 6.4) and Hughes uses a timestamp to collect public objects (explained in Section 6.2).

6.1 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 exit items and entry items.

The main problem with distributed tracing is to synchronise the distributed mark phase with independent sweep phases. During the mark phase local collectors receive and sends marking messages exchanged between clients and owner space. A local GC can be resumed if it receives a marking message for an object it owns. Therefore, spaces are alternatively cooperating to the global marking (running state) and alternatively waiting for a marking message (idle state) as illustrated in Figure 12. The mark phase is complete when all reachable public objects have been marked and there is neither marking or acknowledgement message in transit. Afterwards, each space triggers independently a sweep phase in order to reclaim public and local garbage objects.

Another problem of fault-tolerant distributed tracing is to maintain the consistency of entry items with exit items, in the face of message and space failures, and of race conditions. In fact, if local GCs, mutators, and the inter-collector all operate in parallel with each other and messages are not instantaneous, then strict consistency is not achievable. Therefore, local GCs rely on local, necessarily *inconsistent* information in order to detect if an object is garbage or not.

Here is an example illustrating this consistency problem. Consider the system, illustrated in Figure 13, composed of only two spaces, A and B and a *coordinator* space C. Each space independently triggers a local GC and afterwards informs the coordinator of references it owns within the system. The coordinator builds up a snapshot of the global state of the system based on information provided by the spaces. Here is a scenario, sketched by Figure 13, where the coordinator is fooled due to inconsistent information.

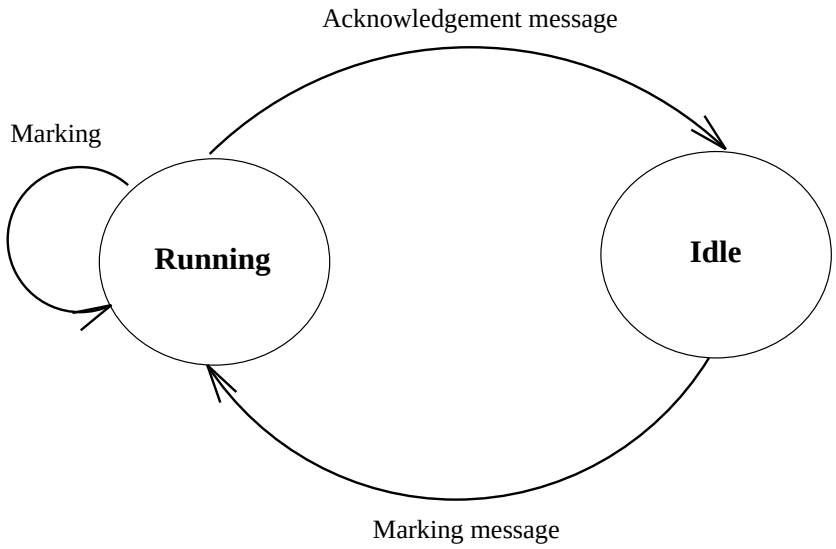


Figure 12:
Distributed tracing.

(i) Space A, holding a local object (i.e. accessible from its local root) x, sends a reference to x to space B, then discards its local pointer to x.

(ii) Space A performs a local GC and concludes that x is not locally reachable; and sends this information to C.

(iii) Space B sends the reference $@x$ back to A, then discards its own reference to x.

(iv) Space B performs a local GC; similarly, it concludes that x is not locally reachable.

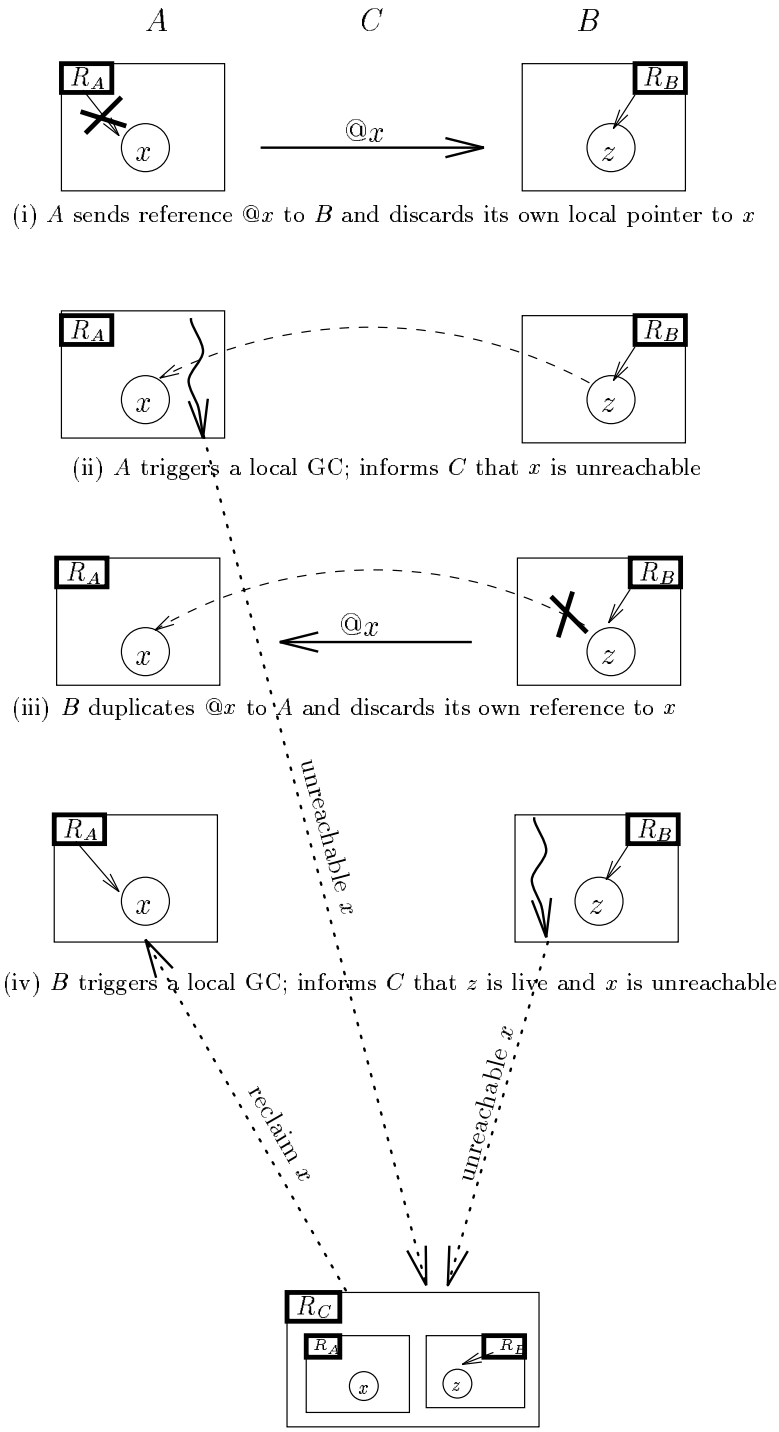
(v) The coordinator C summarises the informations received from spaces A and B: x is not accessible in either A or B. It wrongly concludes that x is garbage, whereas in fact it is locally reachable from A, and directs space A to reclaim x.

A message containing a reference may be in transit at the time of a local GC is triggered. If those messages are not taken into account by the coordinator then some objects may be considered unsafely as garbage. Here is an example, illustrated by Figure 14, where such messages in transit pose problems.

(i) Space B holds a reference to object x on space A; B's reference to x is discarded.

(ii) B performs a local GC; concludes that x is no longer reachable from B; and sends this information to C.

(iii) The coordinator C infers that x is not remotely reachable from B and sends a *delete* message to A.



(i) A sends reference @x to B and discards its own local pointer to x

(ii) A triggers a local GC; informs C that x is unreachable

(iii) B duplicates @x to A and discards its own reference to x

(iv) B triggers a local GC; informs C that z is live and x is unreachable

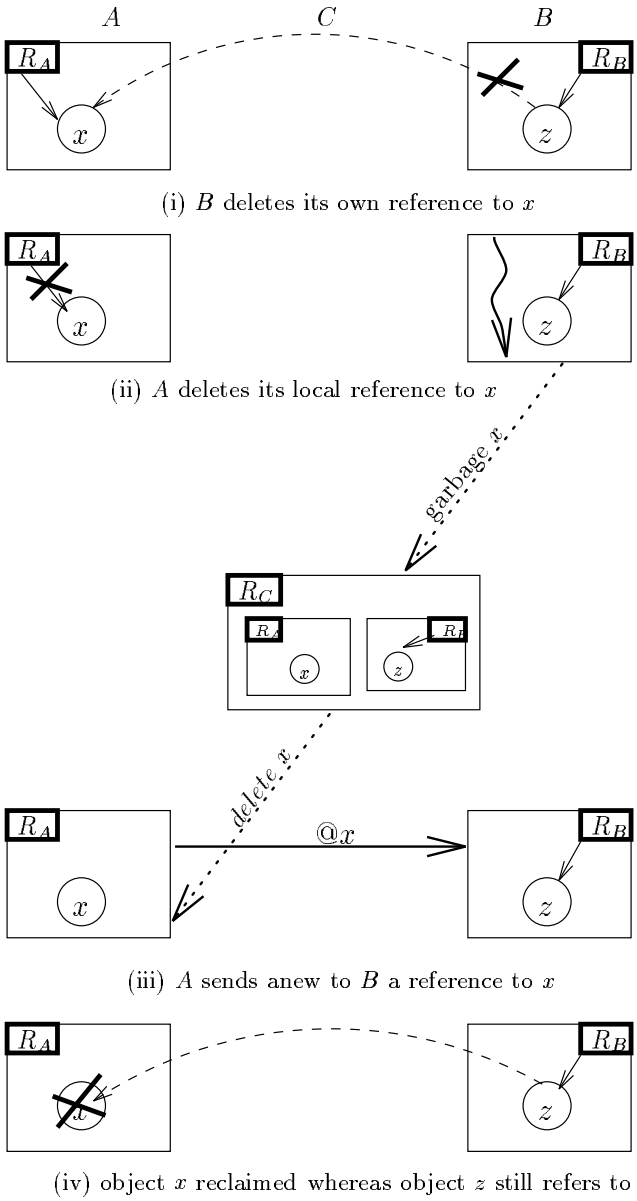
(v) C computes (from an inconsistent view) that x garbage

Figure 13:
Inconsistent GC snapshot.

(iv) While the *delete* message is in transit, A sends this reference @x to B, then makes x locally unreachable (i.e. from A's local root).

(v) The *delete* message arrives to A; hence x is collected.

(vi) B receives the reference to x; hence x is now reachable from B's root, although it has been collected.



(i) B deletes its own reference to x

(ii) A deletes its local reference to x

(iii) A sends anew to B a reference to x

(iv) object x reclaimed whereas object z still refers to it

Figure 14:
Message in transit problem.

A standard solution is to fight inconsistency by using strong protocols such a global barrier to synchronise with a barrier the end of all the local mark

phases [Lang *et al.*, 1992]. An alternative is to allow *safe inconsistencies*, i.e. which do not violate the safety invariants of GC.

6.2 Tracing with Timestamps

Hughes [1985] describes an appealing algorithm based on and a distributed mark-and-sweep algorithm, when mark bits are replaced by timestamps, a global GC propagates an exit item's timestamp to its successor entry item. The key idea of the algorithm is that a garbage object's timestamp remains constant whereas a non-garbage object's timestamp increases monotonically. A timestamp threshold is computed in order to avoid the barrier synchronisation between the mark and the sweep. Scions and an exit items both contain a timestamp initialised with a global clock. Each local GC repeatedly traces objects from the local root and from the entry items. A exit item reachable from the root is marked with the time at which marking started; one reachable from an entry item receives that entry item's timestamp. Scions with a timestamp less than the global threshold are collected. Scions that carry timestamps less than the threshold can be safely reclaimed. Hughes' algorithm collects both cyclic and acyclic distributed garbage since it is based on tracing.

The time when the local GC was triggered is called *GC-time*. The local root is labelled with this GC-time. The local GC repeatedly scans objects reachable from the local root first, then from entry items down to exit items. Scions are scanned in decreasing timestamp order. This avoids multiple scanning of the same object.

At the end of a local GC, up-to-date exit item timestamps are sent to the corresponding owners to increase the entry item timestamps (if lower than the exit item's). Upon increasing an entry item timestamp, the space records that this particular timestamp has not been propagated. Each space maintains a local *redo* timestamp equal to the greatest timestamp totally propagated. When all timestamping messages have been processed, the owner sends back an acknowledgement message to the sender. The sender space collects acknowledgements message before increasing its own local redo to GC-time.

The basic idea of this algorithm is that any entry item whose timestamp is lower than a global *threshold* is garbage. The threshold is equal to the lowest value of all redos. Computation of the threshold is tricky and relies on a global termination protocol. Clocks are assumed to be synchronised and message delivery latency is bounded. However these assumptions are not required for correctness but only for liveness.

Figure 15 illustrates how Hughes' algorithm reclaims a distributed garbage cycle composed of two objects u and v located respectively on spaces A and B. The following scenario chronologically describes steps (i) to (v) decomposed in Figure 15. For the sake of clarity, the timestamp of some entry item b is noted $b \cdot timestamp$ and that of exit item y_A is noted $yA \cdot timestamp$.

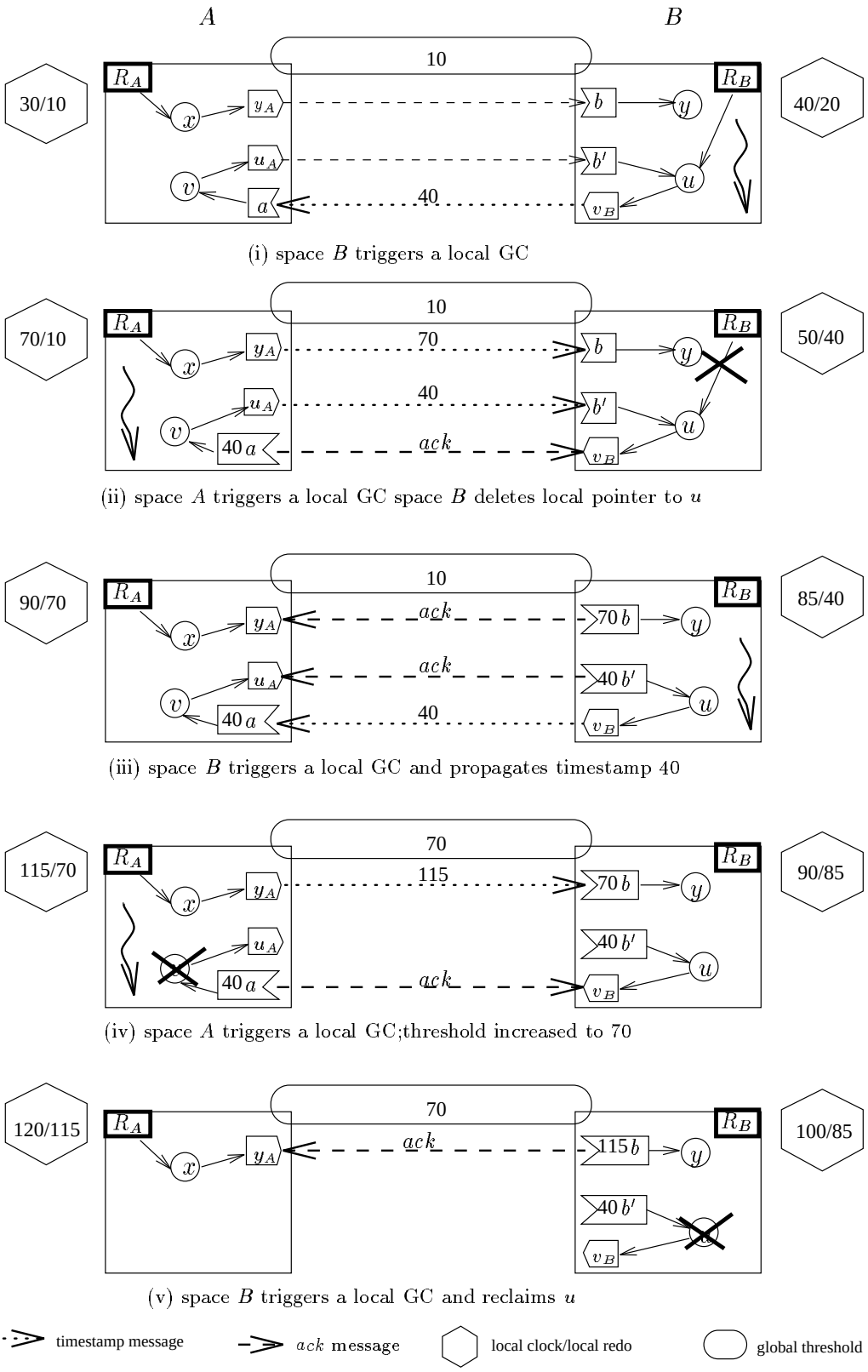


Figure 15:
Example of Hughes' timestamping technique.

(i) At *clock*= 40, space B triggers a local GC and propagates a timestamp 40 to entry item a of space A.

(ii) At *clock*= 50, space B removes the local pointer from R_B to u.
At *clock*= 70, space A processes and acknowledges timestamp message 40 and initiates a local GC. Then, $yA \cdot timestamp = 70$ and $uA \cdot timestamp = 40$. It propagates timestamp 40 to entry item b' of B.

(iii) Space B processes and acknowledges timestamp messages.
Thereafter $b \cdot timestamp = 70$ and $b' \cdot timestamp = 40$.
At *clock*= 85, space B triggers a local GC. The timestamp 40 of entry item a is propagated to exit item y_B and timestamp 40 message is flowed to entry item A.

(iv) At clock = 100, $threshold$ = min 70 and 85.
At *clock*= 115, A triggers a local GC; since $a \cdot timestamp \leq threshold$ then entry item a and object v are reclaimed.

(v) At *clock*= 100, B triggers a local GC; u is reclaimed; $b \cdot timestamp = 115$.

As illustrated by the above example, a local collection starts a new round of global tracing by marking its local root with the current timestamp. Moreover, the collection also performs some work for previous tracing rounds by propagating the timestamp of entry items. For instance, in Figure 15-(iv), the local collection on space A increases the timestamp of exit item y_A to the *clock* value and also reclaims garbage objects such as v.

Hughes' algorithm has some shortcomings. The threshold computation relies on a termination algorithm which is notoriously costly and not scalable. It is likely to slow down computation on each space and in particular local GC. Moreover, the algorithm is not resilient to space failures, since a failed space prevents increasing the threshold, hence blocking garbage collection on all other spaces. Even a slow space unwilling to trigger a local GC, will leave the global threshold stuck to an old value. This is true even if the failed or slow space does not hold any (remote) reference to other spaces.

6.3 Logically Centralised Reference Service

In contrast to previous proposals that attempt to compute on each space the global accessibility of public objects, Liskov and Ladin [1986] compute global accessibility of objects on a highly available centralised service. This service is logically centralised but physically replicated, hence achieving high availability and fault-tolerance. All objects and tables are backed up in stable storage. Clocks are synchronised and message delivery delay is bounded. These assumptions enable the centralised service to build a consistent view of the distributed system.

In the 1986 paper, the garbage detector relies on a local tracing garbage collector, extended with the ability to identify *some* of the paths between entry items

and exit items. Each local collector informs the centralised service about its references to remote objects, about the references it has sent and about the paths. Based on the paths transmitted, the centralised service internally builds a representation of the graph of inter-space references, and detects garbage (including garbage cycles), using a standard tracing algorithm. Afterwards, the centralised service informs spaces about the reachability of their public objects.

Rudalics [1990] proves that this algorithm is incorrect, by exhibiting a counter-example. The algorithm is unsafe when a a cyclic graph is distributed among several spaces and remotely reachable from another one. Figure 16 exhibits such a graph composed of objects z, y and t distributed among spaces B and C and an object x which locally points to object y. Objects x and z are global whereas object y is local to space B. All objects in the graph are live since they are all reachable from root R_A.

The unsafe behaviour occurs because object y is local and locally pointed by (at least) two public objects x and z. The correctness of Liskov and Ladin's algorithm depends on the order of traversal by the local GC. For instance, assume that the local GC traverses object z first and then object x. The traversing of object y will only occur once and the single path detected is the path between object z and t, although there also exists a path between object x and t. Consequently, at the end of the local GC space B informs the centralised service only of the first path, omitting the second one. Since object t is not locally accessible from R_C the centralised service will deduce unsafely that object z and t are garbage.

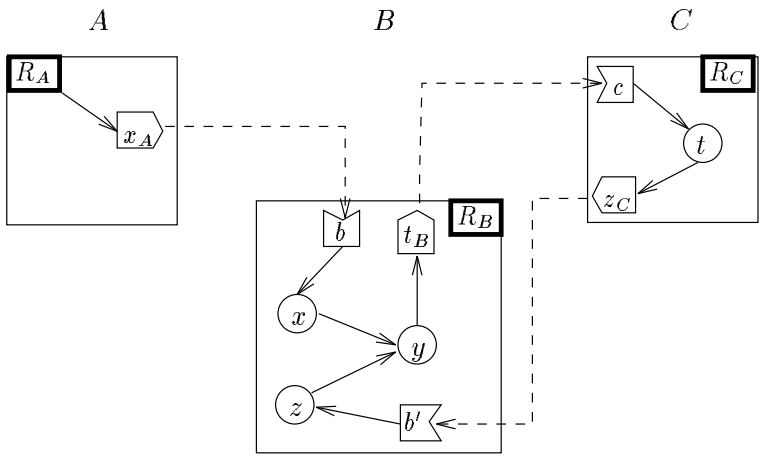


Figure 16:
Rudalics' counter-example to Liskov's central service.

Rudalics comes up with two inconvenient solutions to deal with this unsafe behaviour. The first solution consists of computing a *connectivity matrix* for each space. The connectivity matrix records the paths among *all* pairs of entry items and exit items. This solution increases significantly the cost of path computation and the space required to record the paths. Assuming that a space contains m

entry items and n exit items the connectivity matrix will necessitate at least $n * m$ bits. Furthermore, the cost of computing the connectivity matrix is proportional to $n * o$ given that there exists o local objects. Note that any tracing local collector might be used provided that different marking colours are used for each trace.

The second solution is to inform the centralised service of all local pointers between global and local objects. For instance, in Figure 16, B's local collector must record pointers from x to y and from z to y, as well as the reference between y and t. This information is provided to the centralised service for further analysis. Thus, the centralised service is able to deduce that object t and z are globally reachable from R_A. This solution is cheaper than the previous one in terms of local computation but increases significantly the communication between spaces and the centralised service.

In a later paper, Ladin and Liskov [1992] simplify and correct the deficiencies of the above proposal by adopting Hughes' algorithm (see Section 6.2) and loosely synchronised local clocks. Hughes' algorithm eliminates inter-space cycles of garbage, thereby eliminating the need for an accurate computation of the paths and for the central service to maintain an image of the global references. Furthermore, the centralised service determines the garbage threshold date, making a termination protocol unnecessary.

6.4 Tracing Within Groups

Recently Lang Queinnec and Piquer [1992], suggested to combine reference counting and mark-and-sweep in order to perform garbage collection within groups. A group is a dynamic collection of spaces (i.e. a space may be removed or added during garbage collection) that may overlap or include other groups. The dynamic property of groups enable to remove failed spaces in order to not block garbage collection. Group nesting allows to build a hierarchy of groups in order to support cyclic garbage collection within networks as large as the world. A space belonging to a group is a *member* of that group. Conversely, *external* spaces of a group do not belong to that group.

The algorithm proceeds in several steps. The first step is a *group negotiation*. During this step, spaces exchange messages to build up a group. The next step, *initial marking*, distinguishes inter-group from intra-group references. For this purpose, each space sends a decrement message for each exit item it holds within the group. At the end of the initial marking, entry items with a counter equal to zero are internal to the group and coloured *white*; the others are referred from at least one external space, and will be coloured *black*.

The following step, *global marking*, performs a global mark-and-sweep within the group. This step relies on local tracing garbage collections to propagate the black colour from entry items down to exit items. The marking phase first traverses the local root and then the list of list entry items, first the black ones, then the ones white. This order of traversing entry items prevents to whiten black objects

(i.e. exit items). At the end of tracing, all blackened exit items are reachable either from a root within the group, or from some external space. Conversely, white exit items are garbage. Each space sends a colour message containing a list of black exit items it holds to each corresponding space within the group. The marking step completes when all spaces have sent a *colour message* to each peer and when there is no more colour message in transit. Note that colour message may lead to blacken white exit items, leading to send additional colour messages.

At the end of the marking step, white entry items can be freely reclaimed. Each space runs a *sweep step* to reclaim unreachable public objects.

Figure 17 shows two steps of Lang-Queinnec-Piquer's technique: the global marking and the sweep steps. This example, considers one group composed of spaces A and B, but the distributed graph includes space C, not showed in Figure 17. The distributed graph contains two garbage cycles: one composed of objects v and u (spaces A and B), the other composed of objects x, y and z (respectively owned by spaces A, B and C). At the end of the collection, the former cycle is collected since it is entirely included in the group, whereas object x and y, belonging to the second cycle, remain unreclaimed. Step (iii) shows why a termination protocol is required to ensure completion of the global marking. In this step, space B sends to space A a colour message to blacken entry item a. Although space A has already triggered a local GC, this message must be processed by space A before completing the marking phase. This necessitates space A to trigger again a local GC in order to propagate colours from the newly-blackened entry item a to exit item y_A.

The counts associated with entry items must be kept accurate during the computation. As stated earlier (see Section 3), keeping counts accurate requires causal reliable communication channels. Therefore, the algorithm inherits from reference counting the inability to tolerate message failures.

A entry item referred from an external space is coloured black. Therefore, garbage collection is conservative with respect to inter-group references: a subgraph referenced from outside the group is not considered for collection until a larger group is formed, encompassing the entire graph. Consequently, the liveness of the algorithm relies heavily on the group negotiation in order to group together spaces which interacts strongly. Large subgraphs — in particular cyclic data structures — will necessitate to extend groups to all spaces owning part of the data structure.

Scalability of the group garbage collector to very large networks is achieved through a hierarchy of nested groups. Nested groups benefit from larger groups that perform some of their work. However, large group GCs are longer than smaller ones and therefore retain more floating garbage. For that reason, the authors assume that large group GCs are rare compared to small group GCs.

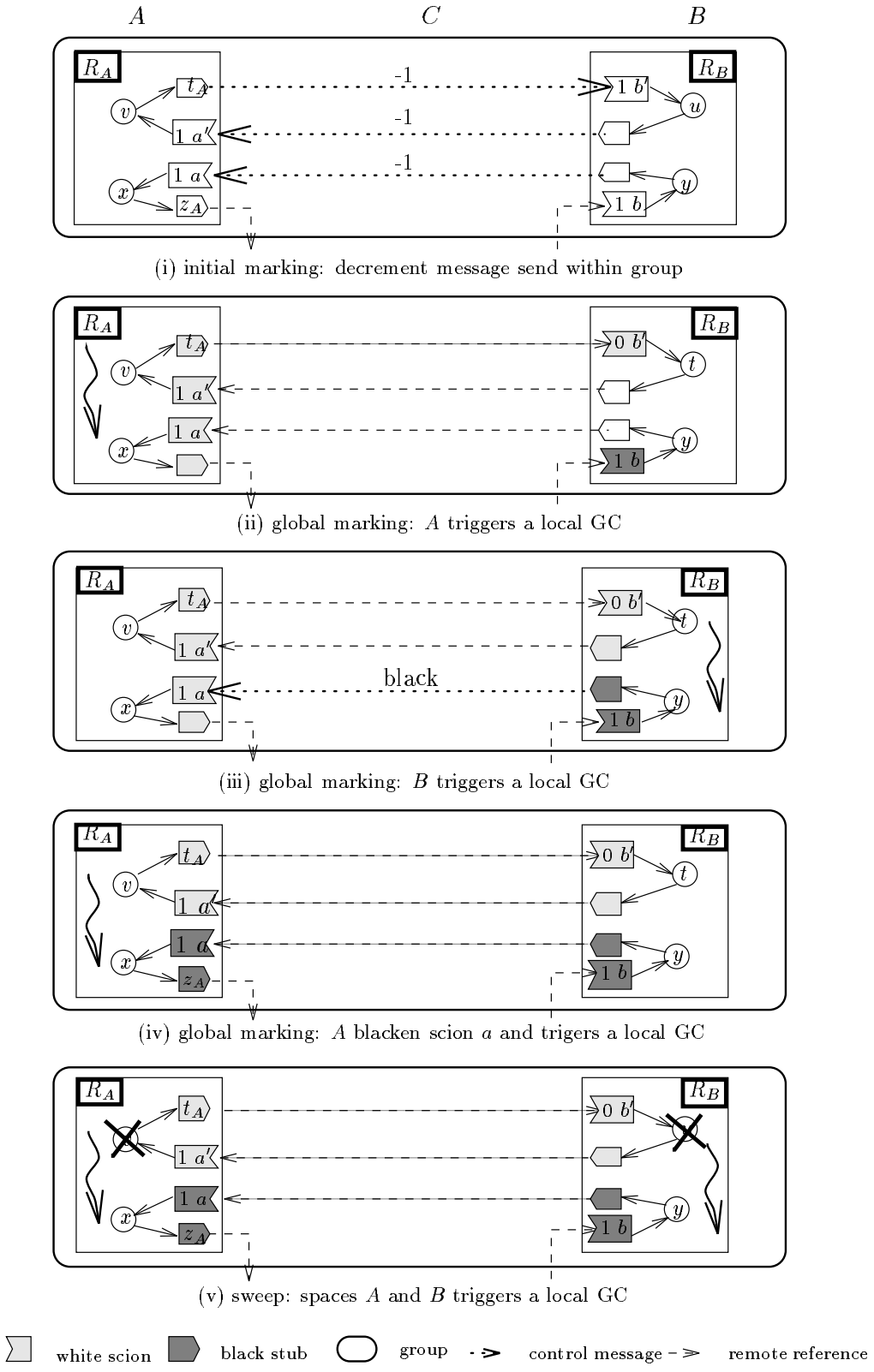


Figure 17:
Tracing within a group composed of spaces A and B.

Table 1: Taxonomy of some popular distributed GC techniques.

Distributed GC Techniques	*main distributed GC characteristics*							
	cyclic	*floating garbage*	*large scale*	*imple-mented*	*space failure*	*message failure*		
						duplic.	*loss*	*late*
Reference Counting								
[Bevan, 1987]								•
[Dickman, 1991]	○	•				•		•
[Piquer, 1991]		•	•	•				•
[Goldberg, 1989]								•
Reference Listing								
[Shapiro *et al.*, 1990]		•	•	•	•	•	•	•
[Birrell *et al.*, 1993]		•	•	•	○	•	•	•
Tracing								
[Ali, 1984]	•							
[Hughes, 1985]	•	•						
[Ladin and Liskov, 1992]	•				•			
[Lang *et al.*, 1992]	○	•	○		•			
[Juul and Jul, 1992]	•			•				

○ : the characteristic is *not* intrinsically achieved by the distributed GC
• : the characteristic is intrinsically achieved by the distributed GC.

7 Conclusion

Reference counting associates with each public object a count of the number of references to it. This direct adaptation of uniprocessor reference counting is not well adapted to distribution because maintaining the count accurately is costly in terms of messages and necessitates that messages be delivered reliably in their causal order.

Weighted Reference Counting (explained in Section 3.2) associates a partial weight to each reference and a total weight to each public object. The invariant ensures that a total weight is always equal to the sum of extant partial weights. Upon duplication or creation of a reference, the partial weight contained in the original exit item is shared with the newly-created reference, thus avoiding race conditions.

However, Weighted Reference Counting technique is not resilient to message failures. Optimised Weighted Reference Counting tolerates message loss by relaxing the invariant, at the expense of creating floating garbage. Reclamation of this floating garbage is ensured by relying on global tracing to gather objects that only conform to the relaxed invariant.

When a total or partial weight drops to a minimal value the weight cannot be shared anymore, preventing further duplication of that reference. One solution consists in creating an indirect reference. Access through an indirect reference is cumbersome for (at least) two reasons: it increases significantly the number

of messages needed to access remote objects and a failed indirect space prevents clients to access an object.

Indirect Reference Counting (see Section 3.5) solves this problem by keeping two kinds of locator in each reference. The strong one is indirect and serves only garbage collection purposes, whereas the weak one is direct (in the absence of migration) and carries remote invocations. However, Indirect Reference Counting lacks of a shortcut protocol in order to speed up reclamation latency.

Most reference counting-based techniques are fragile in the sense that they attempt to keep the counters accurate. In contrast, Reference Listing (explained in Section 4) gives up counters by creating a separate entry item for each client space. This renders control messages idempotent, hence resilient to loss or duplication. Since each owner space knows the set of its clients, it may detect a failed client or a client not communicating.

Reference counting and Reference Listing share the same inability to reclaim garbage cycles. Several hybrid techniques (see Section 5) combine reference counting with tracing technique to collect cycles of garbage. An obvious combination is to run (infrequently) a distributed tracing encompassing all the spaces in the network; hence all spaces must cooperate, even those that do not contain any part of the distributed graph. In contrast, Bishop's migration technique and Vestal's trial deletion only need the cooperation of spaces that owns a part of the distributed cycle. However, Bishop's migration technique does not accommodate indirect references well, and several attempts of trial deletion cannot run concurrently.

Tracing is inherently cyclic and potentially more fault-tolerant, because each execution of the collector forgets the consequences of past faults. However most tracing-based garbage collection techniques (see Section 6) make strong assumptions on the reliability of the network. The main reason is that they try to track accurately at all times the minimal set of reachable objects. For instance, Liskov and Ladin's central service (explained in Section 6.3) requires a bounded message delivery delay in order to build a complete and consistent view of the distributed graph.

Tracing-based techniques require all spaces to cooperate in the distributed collection. Therefore, those techniques cannot make progress if even a single space is crashed, even if that space does not own any part of the distributed graph. Lang-Queinnec-Piquer's technique (explained in Section 6.4) is somewhat less sensitive to space failures, provided that the failed space is not part of the group. However, the technique inherits the shortcomings of reference counting and would benefit from reference listing in order to be scalable.

Acknowledgements

We are specially grateful to Michel Ruffin and Laurent Amsaleg for their helpful comments on early drafts of this paper. We would like also to thank Sacha Krakowiak who greatly encouraged us to publish this work.

References

[Ali, 1984] K. A. Mohammed Ali. *Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems.* PhD thesis, Royal Institute of Technology, Dept. of Computer Systems, Stockholm, Sweden, 1984.

[Bakker *et al.*, 1987] W. Jacobus Bakker, L. Nijman, and Philip C. Treleaven, editors. *Parallel Architectures and Languages Europe*, number 258, 259 in Lecture Notes in Computer Science, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[Bevan, 1987] David I. Bevan. Distributed Garbage Collection Using Reference Counting. In Bakker et al. [1987], pages 117–187.

[Birrell *et al.*, 1993] A. Birrell, D. Evers, G. Nelson, S. Owicki, and T. Wobber. Distributed garbage collection for network objects. Technical Report 116, Digital Equipment Corporation Systems Research Center, December 1993.

[Birrell *et al.*, 1994] A. Birrell, G. Nelson, S. Owicki, and T. Wobber. Network objects. Technical Report 115, Digital Equipment Corporation Systems Research Center, February 1994.

[Bishop, 1977] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR-178.

[Cardelli *et al.*, 1988] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical report, Digital Systems Research Center and Olivetti Research Center, Palo Alto, CA, 1988.

[Corporaal *et al.*, 1990] H. Corporaal, T. Veldman, and A. J. van de Goor. An Efficient, Reference Weightbased Garbage Collection Method for Distributed Systems. In *Proceedings of the PARBASE-90 Conference*, pages 463 – 465. IEEE, 1990.

[Dickman, 1991] Peter Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks With Few Failures.* PhD thesis, University of Cambridge, United Kingdom, September 1991.

[Dickman, 1992] Peter Dickman. Optimising Weighted Reference Counts for Scalable Fault-Tolerant Distributed Object-Support Systems. Submitted for HICSS 26, June 1992.

[Ferreira and Shapiro, 1994] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241. ACM, November 1994.

[Fowler, 1986] 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.

[Goldberg, 1989] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–320, Portland, Oregon, June 1989. ACM Press. Published as *SIGPLAN Notices 24*(7), July 1989.

[Hughes, 1985] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *ACM Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy, France, Software Practice and Experience 1985. Springer-Verlag.

[Juul and Jul, 1992] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.

[Kernighan and Ritchie, 1978] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, Englewood-Cliffs, N.J., 1978.

[Ladin and Liskov, 1992] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Principles of Distributed Computing*, pages 708–715, 1992.

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

[Lang *et al.*, 1992] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 39–50, Albuquerque, New Mexico (USA), January 1992. ACM Press.

[Lins and Jones, 1991] Rafael D. Lins and Richard E. Jones. Cyclic weighted reference counting. Technical Report 95, University of Kent, Canterbury, United Kingdom, December 1991.

[Lins, 1990] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. Technical Report 75, University of Kent, Canterbury, United Kingdom, June 1990. To appear in *Information Processing Letters*.

[Liskov and Ladin, 1986] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault- tolerant distributed garbage collection. In *Fifth*

ACM Symposium on the Principles of Distributed Computing, pages 29–39, 1986.

[Piquer, 1991] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In Eddy Odijik, M. Rem, and Jean-Claude Sayr, editors, *Parallel Architectures and Languages Europe*, number 365, 366 in Lecture Notes in Computer Science, pages 150–165, Eindhoven, the Netherlands, June 1991. Springer-Verlag.

[Plainfossé, 1994] David Plainfossé. *Distributed Garbage Collection and Reference Management in the Soul Object Support System.* PhD thesis, Université Paris-6, Pierre-et-Marie-Curie, Paris (France), June 1994. Available from INRIA as TU-281, ISBN-2-7261-0849-0.

[Rudalics, 1990] Martin Rudalics. Correctness of distributed garbage collection algorithms. Technical Report 90-40.0, Johannes Kepler Universitat, Linz Austria, 1990.

[Shapiro *et al.*, 1990] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, inria, rocquencourt, November 1990.

[Shapiro *et al.*, 1992] 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.

[Stroustrup, 1991] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[Vestal, 1987] S. C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, University of Washington, Seattle, Washington, January 1987.

[Watson and Watson, 1987] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architecture. In Bakker et al. [1987], pages 432 – 443.

[Weis, 1990] P. Weis. The CAML Reference manual, Version 2.6.1. Technical Report 121, INRIA-Rocquencourt, 1990.