Diffusion Tree Restructuring for Indirect Reference Counting

Dr Peter Dickman
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
Scotland, UK
pd@dcs.gla.ac.uk

ABSTRACT

A new variant algorithm for distributed acyclic garbage detection is presented for use in hybrid garbage collectors. The existing fault-tolerance of Piquer's Indirect Reference Counting (IRC) is qualitatively improved by this new approach. The key insight that underpins this work is the observation that the parent of a node in the IRC diffusion tree need not remain constant. The new variant exploits standard mechanisms for implementing diffusion trees and remote references, using four simple low-cost techniques to dynamically restructure the trees to reduce their depth. This variant reduces third-party dependencies, which make standard IRC vulnerable to process failure, while retaining tolerance of message reordering and without incurring substantial overheads. The paper carefully motivates the algorithm, presents the full technical basis for its development, provides a clear explanation of implementation details and includes an initial discussion of performance issues.

1. INTRODUCTION

In this paper a new variant is introduced which qualitatively improves the fault-tolerance of Indirect Reference Counting (IRC), a distributed acyclic GC algorithm designed for use in hybrid collectors. The paper carefully motivates the algorithm, presents the full technical basis for its development and provides a detailed explanation of its implementation.

Many proposals for distributed garbage collection algorithms assume a hybrid collector [14, 16, 17, 20], in which two algorithms are combined: one, used infrequently, collects all garbage including cycles, but at a considerable cost; the other, operating continuously, collects the bulk of the acyclic garbage using a cheap, safe, but incomplete, mechanism. The new algorithm is intended for use as the base acyclic collector in such a hybrid. Acyclic GC algorithms can also be applied in application domains in which cycle construction does not occur or can be explicitly managed.

Standard naïve reference counting is of limited value as the acyclic garbage collection technique for a distributed object system, suffering as it does from a variety of pathological behaviours in the event of message passing failures in the underlying communications mechanisms (described briefly in §2.1). However, variants of reference counting have been proposed which offer improved fault-tolerance.

In 1991, Piquer [15] developed Indirect Reference Counting (IRC), which uses a diffusion tree to eliminate the need for remote increment messages, and hence avoids a race condition which can lead to incorrect behaviour by distributed reference counters. But Piquer's algorithm, although safe, introduces long-term third-party dependencies which potentially reduce the completeness of the acyclic garbage collection in the presence of process failures.

The new algorithm is a variant of Piquer's IRC. The vital insight that underpins this work is the observation that the parent of a node in the IRC diffusion tree need not remain constant. The new algorithm exploits standard techniques for implementing remote references and diffusion trees, extending them to allow dynamic restructuring of the IRC diffusion trees to reduce their depth. This IRC variant thereby reduces the third-party dependencies which make standard IRC vulnerable to process failure, without affecting the existing tolerance of message reordering.

Moreau [13] has also applied diffusion tree reorganisation to distributed garbage collection, as described in section 6, but in an implicitly more eager technique with less fault-tolerance built on a different underlying communication model.

Section 2 presents background material on: the problems posed by distributed reference counting; the IRC algorithm; the low-level implementation of remote references; and the construction and maintenance of diffusion trees. Techniques for restructuring diffusion trees are then introduced in section 3, with a detailed presentation of the implementation issues raised. The new algorithm exploits these advances and section 4 shows how improvements in fault-tolerance are achieved in the presence of fail-stop nodes. An analysis of the performance implications follows in section 5, emphasising the algorithm's effect on the shape of the diffusion trees. The paper concludes with a summary and suggestions for future work.

2. BACKGROUND

This section provides several pieces of background detail, to allow the implementation-level requirements of the new algorithm to be fully understood. Readers with a prior knowledge of distributed systems may wish to skip §2.1, on the problems of reference counting, and §2.3 on remote references. The primary survey of garbage collection algorithms is by Jones' [8], although the surveys by Louboutin [11] and Plainfossé & Shapiro [18] may also be of interest.

2.1 Reference Counting in Distributed Systems

In standard naïve distributed reference counting, remote references indicate an object in another process, at which a reference count is maintained. As copies of references are constructed and discarded in the distributed system, `inc` and `dec` messages are sent to the target object to increment and decrement the reference count. If two such messages are delivered out of order, it is possible for a reference count which should change through the sequence $1 \rightarrow 2 \rightarrow 1$ to instead be changed $1 \rightarrow 0 \rightarrow 1$. This second alternative is disastrous, since the referenced object will be reclaimed when the count becomes zero. The error will be noted when the delayed `inc` message arrives, but by then may be unrecoverable.

These race conditions can be avoided if all messages are passed using a synchronous procedure call model, with careful sequencing of the actions. The crucial step is to not return from sending a reference-carrying message until the corresponding `inc` message has been processed. However, this approach may be prohibitively costly, since it does not allow `inc` and `dec` messages to be delayed until other traffic is passing between the relevant nodes, and hence may increase the network traffic by a substantial factor. There is, in consequence, significant interest in message-based reference counting, as a means of reducing network load, even when "normal" communication is RPC based.

Other message based problems arise if `inc` messages are duplicated (the object will never become garbage) or lost (the object may be incorrectly collected), and dual problems arise with `dec` message duplication (incorrect collection of non-garbage) or loss (incorrect retention of garbage). Since message duplication is easier to detect and avoid than message loss, many reference counting variants have been proposed which avoid the need for remote increment messages and assume no message duplication [1, 2, 4, 5, 7, 15, 23]; these algorithms are safe if messages are lost or re-ordered, but may not be complete.

In addition, all counter based mechanisms with fixed-size counters have the potential for counter overflow. It is safe, but inaccurate, for the counters to overflow to an undecrementable infinite value, as this leads to incorrect retention of garbage. It is, however, unsafe for the counters to wrap-around, as this may lead to incorrect collection of non-garbage.

Finally, all distributed garbage collection algorithms, not just reference counters, may have problems in the event of process(or) failure. If a failed node held remote references, the targets of those references cannot be declared to be garbage unless it is certain that the failed node will never subsequently recover. Furthermore, if a reference chain passes through a failed process, communication between the holder of the reference and its target may be impeded.

2.2 Indirect Reference Counting

Indirect Reference Counting (IRC) was proposed by Piquer as a solution to the message reordering problem for distributed acyclic garbage collection using reference counting [15]. In IRC, the reference count is, in effect, distributed throughout the processes which contain (or have contained) references to the object. A tree structure links these partial counts, with all leaf nodes currently holding references to the object and all internal nodes holding portions of the reference count. Although decrement messages pass along branches of the tree, increments are purely local; hence there are no `inc/dec` message race conditions caused by reordering in the underlying communications layers.

Piquer implements remote references using two references, one (called the primary in this paper) indicating the desired remote object, the other (called the source pointer here) indicating the source from which this reference was obtained. When a remote reference is copied, the primary reference is identical to that in the original copy, however the source pointer indicates the node at which this copy is being generated, and a reference count local to that node is incremented. When a remote reference is discarded a `dec` message is sent to the indicated source node, rather than the primary node. The source pointers for remote references to a given object form a diffusion tree (defined in §2.4) rooted at the object.

Figure 1 shows the difference between normal reference counting and IRC. In Fig.1(i) we see three objects holding remote references to a fourth (o1 at A). Two of the three have acquired their references directly from A, while the third acquired its reference indirectly. Standard naïve reference counting is illustrated in Fig.1(ii), while IRC is shown in Fig.1(iii). Note that in IRC all remote references have an associated copy count and may have a source pointer which differs from the primary pointer.

2.3 Implementing Remote References

Piquer assumes a classical implementation of remote references, with a local proxy [19] in a process encapsulating each actual remote reference, and standard language-level references to the local proxies allowing local and remote references to appear the same (modulo semantic differences due to the possibility of remote exceptions, differing parameter passing mechanisms, etc). This technique for reference implementation is illustrated in figure 2. The precise lower-level implementation details of the link between the local proxy and remote entry are not significant for the purposes of this paper, so are not discussed here. Figure 2(iii) shows that a single proxy may be used to represent multiple local references to the remote object, and that a single remote entry may be used by multiple proxies[1].

[1]This is the significant difference between distributed reference counting and distributed reference listing [3, 20, 21]. In reference listing, each remote entry is associated with a particular proxy, so there may be multiple entries per object. In reference counting the entry is normally shared by all proxies.

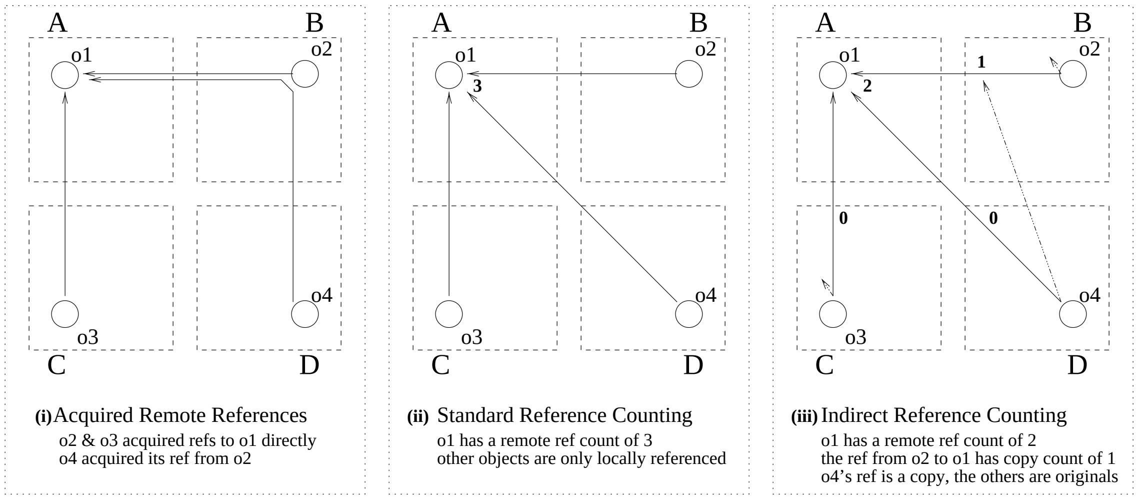
A
B
o1
o2
o3
o4
C
D
3
1
2
0
0
(i) Acquired Remote References
o2 & o3 acquired refs to o1 directly
o4 acquired its ref from o2
(ii) Standard Reference Counting
o1 has a remote ref count of 3
other objects are only locally referenced
(iii) Indirect Reference Counting
o1 has a remote ref count of 2
the ref from o2 to o1 has copy count of 1
o4's ref is a copy, the others are originals

Figure 1: Reference Counts and IRC

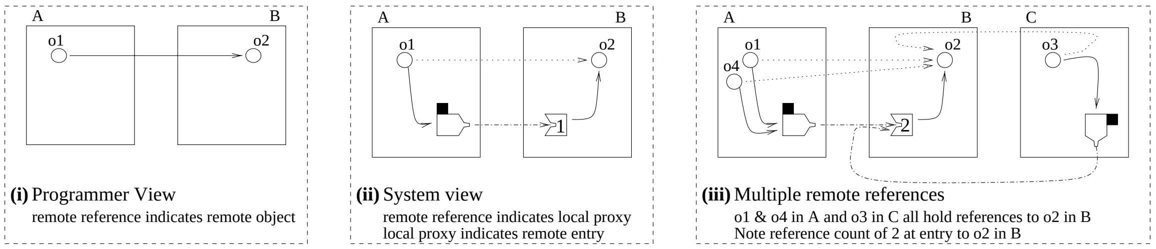
A
B
C
o1
o2
o3
o4
1
2
(i) Programmer View
remote reference indicates remote object
(ii) System view
remote reference indicates local proxy
local proxy indicates remote entry
(iii) Multiple remote references
o1 & o4 in A and o3 in C all hold references to o2 in B
Note reference count of 2 at entry to o2 in B

Figure 2: Standard Remote Reference Implementation

Piquer proposes that the proxies are extended to incorporate the source pointers and this is illustrated in figure 3. The implementation-level notation used in this paper distinguishes between proxies with identical primary and source pointers (e.g. the proxy in process A in Fig.3(ii)) and those with different pointers (e.g the proxy in process C of the same diagram). Furthermore, the proxies must be modified to contain a count of the number of extant copies which originated from that proxy.

Standard techniques for distributed reference implementation can be used to ensure that there is a single entry for a given object, and that only one proxy for a particular object exists in a given process. Consistently with this, the original description of reference unmarshalling for IRC assumed that an existing non-garbage proxy will be retained with its source pointer intact, and that further proxies will not be constructed if new references to the same object are received. Instead the existing proxy is used, and a `dec` message is sent to the source of the new reference copy.

A key feature of IRC, illustrated in Fig.3(iii), is the potential introduction of third party dependencies. The source pointer mechanism requires that proxies with a non-zero copy count must be retained, even if they are not locally referenced. The implication is that, should process A fail, it will not be possible to garbage collect o2 if o3 discards its reference. However, invocations of o2 from o3 do not travel via A, so would not be affected by problems at A.

2.4 Diffusion Tree Management

Diffusion trees are used in a variety of distributed algorithms [12, 22] and are constructed on-the-fly to capture, for example, the routes of dissemination of information as knowledge diffuses through a multi-process application. The nodes of a diffusion tree usually form a superset of the nodes in the system which exhibit a particular property, such as being involved in a computation (for termination detection [6]), or containing a reference to a particular object (for garbage collection [15]). The diffusion tree commonly expands and contracts as the system progresses: adding nodes as they become involved in the computation or are passed a reference; removing nodes when they complete processing or discard the reference. In most uses, the reduction of the tree to the original root indicates that a (stable) property of interest now holds, e.g., the computation has terminated or the object is now garbage.

The source pointers in IRC form a diffusion tree rooted at the referenced object. Figure 4 illustrates the tree developing between the IRC proxies, as references are passed between processes. The IRC diffusion tree is the focus of this paper, and at times that tree alone is drawn. This is illustrated in figure 5, in which the circles represent processes, with one or more objects in each process holding a remote reference to an object in A. Only the source pointers are shown, all primary pointers would indicate a shared remote entry at A.

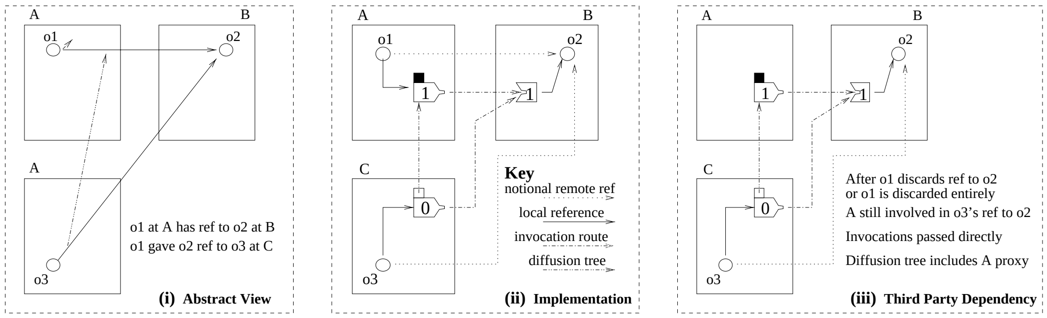
A
B
o1
o2
A
o3
o1 at A has ref to o2 at B
o1 gave o2 ref to o3 at C
(i) Abstract View
A
B
o1
o2
1
1
C
0
o3
Key
notional remote ref
local reference
invocation route
diffusion tree
(ii) Implementation
A
B
o2
1
1
C
0
o3
After o1 discards ref to o2
or o1 is discarded entirely
A still involved in o3's ref to o2
Invocations passed directly
Diffusion tree includes A proxy
(iii) Third Party Dependency

Figure 3: Remote Reference Implementation for IRC

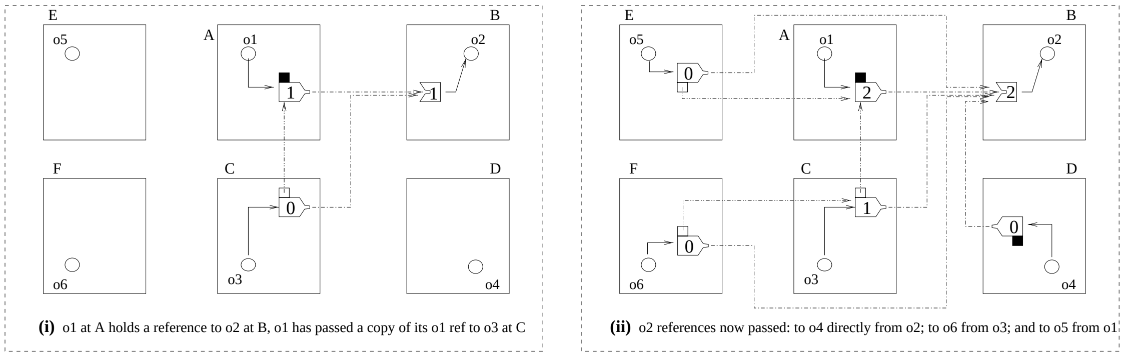
E
o5
A
o1
1
B
o2
1
F
o6
C
0
o3
D
o4
(i) o1 at A holds a reference to o2 at B, o1 has passed a copy of its o1 ref to o3 at C
E
o5
0
A
o1
2
B
o2
2
F
0
o6
C
1
o3
D
0
o4
(ii) o2 references now passed: to o4 directly from o2; to o6 from o3; and to o5 from o1

Figure 4: Diffusion Tree Construction : Implementation View

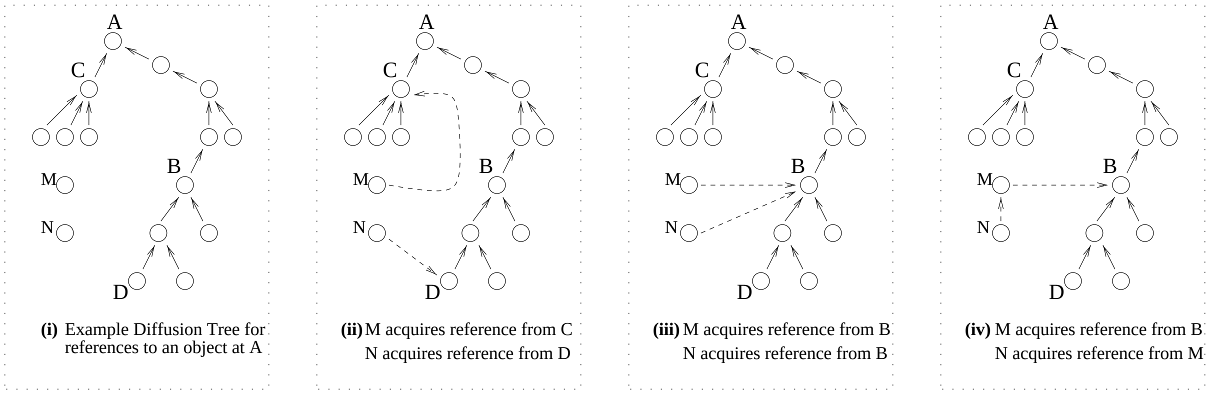
A
C
B
M
N
D
(i) Example Diffusion Tree for references to an object at A
A
C
B
M
N
D
(ii) M acquires reference from C
N acquires reference from D
A
C
B
M
N
D
(iii) M acquires reference from B
N acquires reference from B
A
C
B
M
N
D
(iv) M acquires reference from B
N acquires reference from M

Figure 5: Diffusion Tree Construction : Abstract View of Tree

In general, a diffusion tree is constructed implicitly as messages are passed between processes. A process which is not currently part of the tree may receive a message which requires it to join the tree (because some condition has become true, such as holding a reference to the diffusion tree root). If this happens it joins the tree with the source of the received message as its parent. When the process no longer needs to be part of the tree (because it no longer satisfies the tree membership criterion and has no children), it informs its parent.

3. DIFFUSION TREE RESTRUCTURING

There is a complication in the diffusion tree construction which is exploited by the new algorithm. When a process which is part of the tree sends a message which could induce tree membership, it may not know whether the intended recipient is already part of the tree. To ensure correct and timely diffusion tree extension, the sender of a message assumes that the recipient will need to join the tree, and therefore the sender further assumes it will become the recipient's parent in the tree (hence incrementing the local copy count). If a process which is already part of the tree receives a message which implies tree membership, it now has, notionally, two parents and must rectify this situation. The standard technique is to disregard the "new" parent, sending a `dec` message to the source of the new message which will, in effect, rescind the presumed child status of the recipient.

3.1 The Key Idea

The crucial observation exploited in this paper is that it is not necessary for a node to retain the original parent in the diffusion tree, rejecting all suggestions of new parents. Diffusion tree algorithms function correctly provided the tree structure is maintained, but there is no requirement that the parent of a given node remains constant. It is, therefore, entirely possible for the recipient of additional willing parents to choose between them. A better analogy is to think of a simple employer-employee relationship, rather than parent-child, with occasional changes of employer when new opportunities arise but no periods of unemployment.

Many strategies present themselves for determining which parent to retain, when a new parent offers itself to a tree node. The simplest is the standard "loyal" retention of the original parent and rejection of all new suitors. The new algorithm instead uses two techniques to decrease the depth of the diffusion tree: a depth reducing strategy for parent selection is described in §3.1.2, and an RPC-time mechanism which also reduces the tree depth is presented in §3.1.1.

By way of contrast, an incorrect alternative strategy is illustrated in figure 6; this always accepts the new offer and rejects the existing parent. The "flighty" algorithm behaves incorrectly if the new parent is a descendent of the message recipient, since it will disconnect the tree structure and introduce a loop, as shown in Fig.6(iii). The "flightly" algorithm only behaves correctly when the new parent is in a different branch of the tree, or is an ancestor, as shown in Fig.6(ii) and Fig.6(iv) respectively.

A number of criteria can be used to determine preferences between possible strategies for parent selection. The crucial criterion is correctness, which immediately eliminates the "flighty" algorithm from further consideration. Another is ease of implementation, and there is no doubt that the traditional "loyal" approach is most straightforward to implement. Other factors, such as the precise impact on message traffic, are discussed further in section 5. The motivation for the technique presented in this paper is a desire to reduce third-party dependencies. To fully eliminate third party dependencies requires that all diffusion trees have depth 1, i.e., that they consist solely of the root and leaves. The two techniques utilised in this paper work towards this goal, but may not always achieve it.

3.1.1 Sub-tree re-rooting during RPC

Consider an invocation from an object at process B in Fig. 6(i), to the object in A which is the tree root. The low-level RPC implementation at the proxy can detect that the primary and source pointers differ and that this proxy is not, therefore, directly adjacent to the root in the diffusion tree. The invocation can carry that datum as a single bit, informing the low-level RPC implementation at the remote entry in the target process, A. Before returning from the remote call, the reference count at the remote entry can be incremented, and when the call returns to the proxy, the source pointer can be changed to directly indicate the root process, i.e. it can match the primary pointer. This requires, of course, that a `dec` message be sent to the original target of the source pointer, the ex-parent of this process in the diffusion tree.

This technique means that during normal activity the diffusion trees can become shallower. Whenever a non-adjacent process is the originator of an invocation, the sub-tree based at that process moves to become directly attached to the root node — hence the name "sub-tree re-rooting". The approach does not, however, affect the tree structure if invocations are rarely or never made, as may be the case at a name server for example.

3.1.2 Depth-reducing parent selection

A depth-reducing parent selection strategy can be deployed in conjunction with, or independently of, sub-tree re-rooting during RPC. On receipt of a message indicating that a second parent exists, the recipient chooses between the candidate parents based on their proximity to the tree root. This requires an estimate of depth in the diffusion tree to be maintained, and that these depths are passed as part of the marshalled form of remote references.

The algorithm assumes that each proxy has an estimate of its depth in the diffusion tree, and that all descendents of it have a greater depth. On that basis, the algorithm chooses to change parent only if the new one has a depth indicator at least two smaller than its own, since it assumes its own parent has a depth one less and that its descendents all have greater depth values. Note that the depth indicator need not be absolutely correct. This is explained further in §3.2.2.

3.2 Implementation Details

The exploited techniques require minor changes to the implementation of the diffusion tree and RPC mechanism. These changes, and further implementation-level mechanisms which improve the behaviour of the algorithm, are described in detail in the following sections. The performance implications of the changes are discussed in section 5.

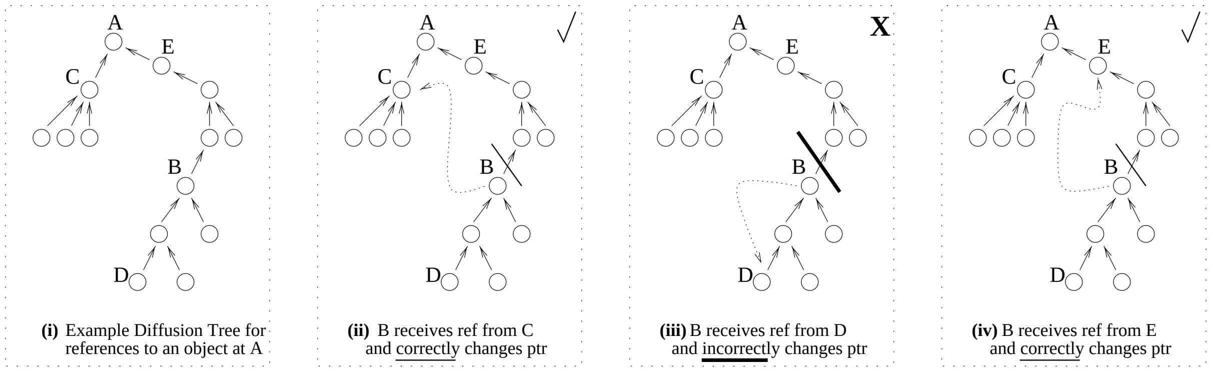
A
E
C
B
D
√
X
(i) Example Diffusion Tree for references to an object at A
(ii) B receives ref from C and correctly changes ptr
(iii) B receives ref from D and incorrectly changes ptr
(iv) B receives ref from E and correctly changes ptr

Figure 6: Diffusion Tree Restructuring

3.2.1 Sub-tree re-rooting during RPC

This technique does not require adjustment of the diffusion tree itself, but does impact on the RPC mechanism and affects traffic patterns. An additional bit is required in RPC request packets to indicate whether the originator of the call requires re-rooting. Furthermore, on receipt of the reply, the RPC originator will generate a `dec` message to its ex-parent, and this may require additional network traffic that could otherwise have occurred far into the future, or perhaps never in the application's lifetime.

Note that the `dec` message must be sent by the originator of the RPC, on receipt of the reply, as only then can it be guaranteed that the original parent is no longer depended upon. Thus sub-tree re-rooting is predicated upon the use of an RPC, rather than message-passing, model for normal computation. This is in sharp contrast with a related mechanism introduced by Moreau, described in §6.

An implementation issue is whether to request re-rooting only in the first RPC call from a proxy with differing source & primary pointers, or in all such calls.

Requesting re-rooting just once requires that a flag is maintained in the proxy showing that such a request has been made. Furthermore, if that re-rooting call fails[2] then further attempts at re-rooting will be delayed until the failure is noticed and the flag cleared.

If, instead, all calls carry the re-rooting request, the re-rooting will occur as soon as any one of the RPCs completes. However, this does mean that multiple increments will occur at the remote entry and it will be necessary to send `dec` messages for each of the additional spurious increments.

The apparent cost of the spurious increments, if multiple re-rooting calls are made, can be reduced if an additional optimisation is used. To substantially reduce this cost a simplified weighted reference counting technique can be incorporated into the proxies, as described in §3.2.4. In the absence of such an implementation it appears, however, that the single-call mechanism is preferable.

[2]If such a failure occurs when the reply is being returned, the reference count at the remote entry will already have been incremented. This has the same (safe) consequences as loss of a `dec` message.

3.2.2 Depth-reducing parent selection

In order to implement depth-reducing parent selection, depth indicators are required in the diffusion tree. If the depth at a particular diffusion tree node is reduced, the descendents of that node have also become closer to the root. However, updating their depth fields would incur additional cost. Although propagation of these updates could be deferred, lazily piggy-backing on other traffic, this has little value since it requires that either:

(i) the parent knows its children: this means reference listing is being used, which introduces considerable additional costs and offers alternatives to IRC as the base mechanism; or

(ii) the child has sought contact with its parent: but this is likely only in the case of a `dec` message, following garbage collection or sub-tree re-rooting during RPC, both of which imply that the depth information is no longer of interest.

It follows that a depth indicator is required which satisfies the criteria that descendents have a higher depth than their ancestors. However, the more accurate the depth indicator the greater the benefit of this parent selection algorithm.

As shown in figure 7, the proposed implementation is to annotate proxies with a depth indicator as well as the source pointer. In proxies which are adjacent to the diffusion tree root, the source pointer matches the primary pointer and the depth indicator is zero[3]. For all other proxies the source pointer indicates the parent in the diffusion tree and the depth indicator is non-zero, as shown in process C of Fig.7(i). Fig.7(ii) illustrates the simplest cases of diffusion tree growth.

The effect of the depth-reducing parent selection algorithm is illustrated in figure 8. The diffusion tree has been annotated with accurate depth indicators in Fig.8(i) and subsequently a message, carrying a reference to the tree root, is passed from an object in process C to an object in process B. This causes a restructuring of the tree, as indicated in Fig.8(ii) and the depth indicator at B will then be changed to 2. Since depth changes are not propagated through the sub-tree, the proxy at process D will incorrectly believe it remains at depth 5 in the tree, and if D received a message from E, containing a reference to the root object, the tree would be further restructured as shown in Fig.8(iii).

[3]Indicated by a black square on the proxy for emphasis.

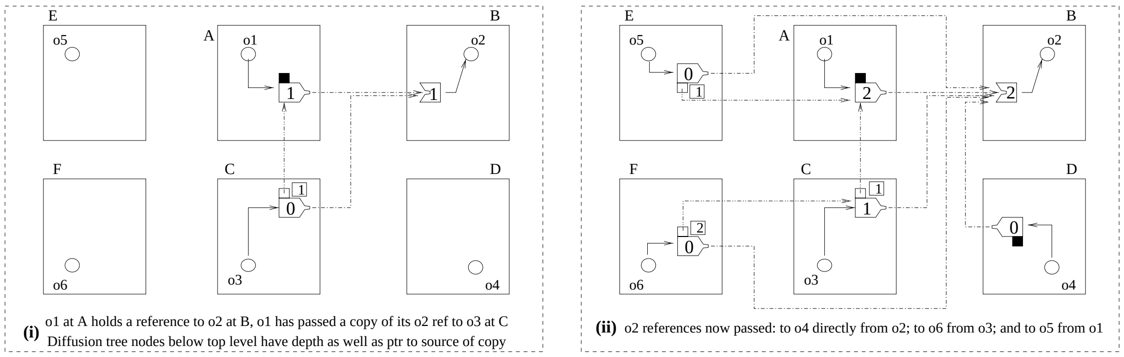
E
A
B
F
C
D
o1
o2
o3
o4
o5
o6
(i) o1 at A holds a reference to o2 at B, o1 has passed a copy of its o2 ref to o3 at C
Diffusion tree nodes below top level have depth as well as ptr to source of copy
(ii) o2 references now passed: to o4 directly from o2; to o6 from o3; and to o5 from o1

Figure 7: Remote Reference Implementation for IRC with Diffusion Tree Restructuring

This final change, although safe, is somewhat unhelpful and demonstrates a potential drawback of this strategy. To reduce these unhelpful changes a third technique, near-leaf parent avoidance, is used. This is described in §3.2.3.

This implementation of depth-reducing parent selection requires that proxies be annotated with a depth indicator, which as a counter may suffer from overflow problems and must therefore include the possibility of a sticky "infinite" value. In addition, the marshalled form of remote references must include the incremented depth value, making them larger by a small number of bytes[4]. Finally, as with sub-tree re-rooting during RPC, it may be necessary for the low-level RPC code at the proxy to redirect the source pointer and therefore generate a `dec` message for the ex-parent.

3.2.3 Near-Leaf Parent Avoidance

The tree restructuring illustrated in Fig.8(iii) shows two minor difficulties. One problem is that process D, in the mistaken belief that it is at depth 5 in the diffusion tree, has selected a parent with depth 3 when its own parent is now at depth 2. Although this could be avoided, by interacting with the current parent, the cost of doing so appears excessive for the benefit achieved.

The second, more worrying but easily avoidable, problem is that the newly selected parent was previously a leaf node, and the ex-parent remains a non-leaf node in the diffusion tree. This means that the number of nodes at which failure would induce problems for the diffusion tree has potentially increased. B is still depended upon, by its one remaining child, and E is now depended upon by D.

Near-Leaf Parent Avoidance requires that when a remote reference is marshalled at a leaf node in the diffusion tree, it is marked with a single bit. This indicates that the source was a leaf node prior to the marshalling of the reference. If the reference recipient finds this bit is set and has a choice about its parent in the diffusion tree, i.e. if it is already in the tree, it will retain its current parent and reject the new one. This approach will lead to a `dec` message being sent to the source of the message and it will again become childless, i.e. a leaf node in the tree.

[4]The increase depends on the desired magnitude of the "infinite" depth value, one byte allows diffusion trees of depth 255 without counter overflow.

The Near-Leaf Parent Avoidance technique may not be optimal, since B's other child might become garbage while E could subsequently pass references to many other processes. However, it does ensure that no one application of the depth-reducing parent selection strategy ever directly increases the number of non-leaf nodes in the diffusion tree.

3.2.4 Simple Weighted Proxies

There are two situations in which `dec` messages can be eliminated by a simple application of a technique akin to weighted reference counting [1, 2, 23]. In weighted reference counting, each remote reference has an associated weight and when a proxy is discarded, the reference count at the associated remote entry is reduced by the weight of the reference, rather than decremented. A similar mechanism can be utilised in the proxy of the optimised IRC mechanism, with each proxy having a weight field. This has advantages in two situations.

When a message is received which contains a copy of a known remote reference no new proxy is needed. In the scheme as presented so far, a `dec` message will be sent to either the source of this message or the source as indicated in the proxy. However, if these two sources are the same, the weight of the proxy can instead be increased. If the proxy is subsequently discarded the associated `dec` message will carry the weight, and cause the associated counter to be reduced by two (or more) rather than one.

The same benefit arises if the sub-tree re-rooting mechanism attempts to trigger re-rooting on all remote invocations until the re-rooting is completed. In that approach multiple RPC replies may indicate that re-rooting has occurred, each of which implies an increment of the reference counter at the remote entry. Rather than sending multiple `dec` messages on receipt of those additional re-rooting success indicators, the proxy weight can again be incremented.

Overflow of the weight counter would prevent the algorithm from correctly garbage collecting the target remote object. However a `dec` message can be generated that removes almost all of the accumulated weight when necessary. Weight-reducing `dec` messages allow the counter size to be limited, permitting a trade off between additional delayable messages and proxy memory size. The upper bound on proxy weights determines the required increase in size of both proxies and `dec` messages: a single byte in each would suffice.

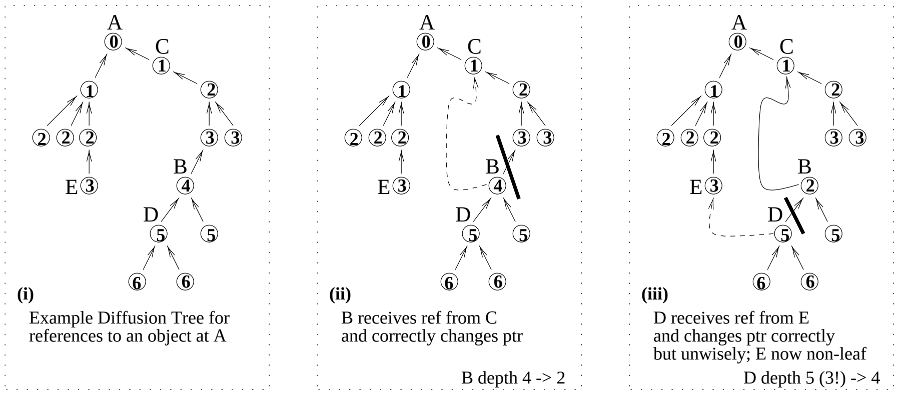
(i)
Example Diffusion Tree for references to an object at A
(ii)
B receives ref from C and correctly changes ptr
B depth 4 -> 2
(iii)
D receives ref from E and changes ptr correctly but unwisely; E now non-leaf
D depth 5 (3!) -> 4

Figure 8: Diffusion Tree Restructuring by Depth Reduction

This use of weights is simpler than in true weighted reference counting, which requires additional marshalling code for copied remote references. However, the inter-linking of these two algorithms has a historical precedent, as Corporaal's variant of WRC [4] in effect uses a diffusion tree to handle weight underflow.

4. FAULT TOLERANCE

The effect of these changes to the diffusion tree structure is to dynamically reduce the depth of the diffusion tree. This has the potential, but is not guaranteed, to reduce the number of nodes in the tree which are retained purely to maintain tree integrity. The proposed mechanisms do not alter the number of nodes which hold a locally accessible reference to the object at which the diffusion tree is rooted.

One benefit arises when a node which was required purely to maintain tree integrity becomes a leaf of the tree, due to the last of its children transferring their source pointer to a node with a smaller label. Leaf nodes which do not locally reference a proxy can discard the proxy, removing themselves from the associated diffusion tree and freeing a small amount of memory. The more important benefit is that reducing the number of nodes which hold proxies purely to maintain the diffusion tree improves fault tolerance.

The new variant maintains the robustness of IRC in the face of message loss and re-ordering. Although some additional `dec` messages may be generated in the new variant, the use of weighted proxies and piggy-backing of `dec` messages helps to limit increased vulnerability due to message loss.

4.1 Avoiding Third-Party Node Failure

Although a failed node does not interfere with the use of the primary references, provided the failed node is not the host of the root object, it does affect garbage collection. A fail-stopped node cannot receive or generate `dec` messages, hence any object whose diffusion tree includes the failed node will never be deemed to be garbage. Even if the node subsequently restarts with its data structures intact, if any `dec` messages have been missed garbage may be incorrectly retained — and recall that re-transmission of `dec` messages is dangerous, as a duplicated message will cause incorrect behaviour, notably the collection of non-garbage.

Both original IRC and the new variant fail safe in the event of node failures, but may not maintain accuracy or timeliness. The scope for improvement of the IRC algorithm lies in the number of nodes which are incorporated into the diffusion tree purely to maintain its structure, due to third party dependencies. Failure of any one of those nodes may defeat the IRC algorithm, yet they are not required for normal computation. If it is assumed that node failures are independent, the probability of unnecessary failure of the IRC diffusion tree is proportional to the number of these additional nodes. Hence, reducing the number of nodes in a given diffusion tree reduces the *probability* that node failure will make it impossible to garbage collect the root object for that tree. So the new variant offers an increased level of fault tolerance. In many cases, sub-tree re-rooting during RPC will eliminate the unnecessary failures, and in other cases the depth-reducing parent selection, supplemented by near-leaf parent avoidance, will reduce the risk of such problems.

5. BASIC PERFORMANCE ANALYSIS

All four techniques introduced in this paper have minor space implications, and small impacts on the processing time for individual RPCs. The mechanisms also modify the traffic patterns, which has a secondary impact on CPU time. Furthermore, in order to determine the value of the new algorithm, its effect on the diffusion tree shape must be considered, since that directly correlates to the improved fault-tolerance.

5.1 Message Count, Space and Time

Messages: At first sight, all parent selection strategies send the same number of `dec` messages. Any difference in network message counts will therefore be due to different piggy-backing behaviour, in carrying the `dec` messages as part of other traffic. In particular, if remote references are frequently carried in RPC calls, an immediate `dec` can be returned as part of the reply. This optimisation will be lost if the new variant chooses to abandon the old, rather than new, parent. In fact, the new parent selection strategy can induce additional messages: changing parent may change the ex-parent into a garbage leaf node leading to the generation of a `dec` message which would otherwise have occurred far in the future, or possibly never in the application's lifetime (if the object never becomes garbage).

Furthermore, sub-tree re-rooting during RPC will lead to an additional `dec` message for each re-rooted proxy. The proxy must send a `dec` message to its ex-parent, and if it subsequently becomes garbage it sends a second `dec` message to the root.

It follows that more messages are almost always required for the new algorithm, with the measurable change being highly application dependent. Simple weighted proxies do allow a small reduction in the number of messages, but this effect could be gained independently of the other mechanisms proposed in this paper, so cannot be legitimately viewed as a compensating benefit.

Space: Sub-tree re-rooting adds a single bit to RPC calls, near-leaf parent avoidance adds a single bit to the marshalled form of remote references, depth-reducing parent selection requires that proxies and marshalled remote references contain an additional depth field, and simple weighted proxies add a weight field to both proxies and `dec` messages. All four techniques therefore increase the space required in the system, but the changes are small.

Savings also accrue, due to the partial elimination of source pointer chains through nodes which do not locally reference the object at the root of a diffusion tree. The precise trade-off in space costs depends on the implementation of remote references and the application behaviour, but is unlikely to be especially favourable to the new algorithm as proxies are small, so the savings made are not large. However, the overall space cost is unlikely to be significant unless most objects are both small and remotely referenced, which is not normally the case in distributed applications.

Time: The new variants slightly increase the cost of marshalling and unmarshalling of remote references, however this change is likely to be undetectable compared with the primary costs of RPC: object graph traversal for argument marshalling and the network transmission delays.

A secondary cause of additional time costs is extra context switches. Sending a `dec` to a prospective new parent has some possibility of interacting with the current task at that processor: since that node has just communicated the reference, its current task may be the desired recipient of the `dec` message. In contrast, sending a `dec` to the old parent node is more likely to cause a context switch, since there is no reason to suspect the required process is the running task. Whether this change will be detectable in normal operation is hard to judge but it seems unlikely, and the costs, if detectable, will again be highly application specific and dependent on the extent to which piggy-backing of `dec` messages occurs.

Summary: It follows that the IRC variant proposed in this paper has marginally higher costs than classical IRC. This cost should, however, be offset against the improved fault tolerance achieved by reshaping the diffusion tree.

5.2 Tree Shape

The new algorithm will never actively increase the number of non-leaf nodes in the diffusion tree, and it endeavours to reduce the number of such nodes by sub-tree re-rooting and depth reduction.

Sub-tree re-rooting during RPC is guaranteed to improve the diffusion tree shape, flattening it and hence reducing the number of third party dependencies which can arise. Depth-reducing parent selection will normally, but is not guaranteed to, also improve the shape of the tree; near-leaf parent avoidance ensuring that individual decisions do not worsen the situation.

It follows that both the depth of the diffusion tree and the number of non-leaf nodes are normally reduced by the new algorithm. The effect of this is to reduce the number of nodes included purely to maintain diffusion tree integrity, in many cases reducing this total to zero and hence qualitatively improving the fault-tolerance of the acyclic garbage collector.

6. MOREAU'S ALGORITHM

Luc Moreau has developed an acyclic distributed garbage collection algorithm which also uses diffusion tree reorganisation [13].

In Moreau's approach the diffusion tree is only a transient construct: proxies only hold a reference to the primary object and do not also contain a source pointer. However, the proxies do contain a reference count indicating the number of copies that have not yet been re-rooted. Rather than maintaining the diffusion tree, Moreau introduces `inc-dec` messages, which are sent by recipients of a new copied reference to the target of the reference. The `inc-dec` message causes the remote entry reference count to be incremented, and is then forwarded to the source of the copied reference to cause the copy count at the source to be decremented.

Thus, if an object at A is referenced from B, and a copy of the reference is passed to C, which does not initially hold such a reference, the sequence of events is as follows:

- B increments local copy count
- B sends reference copy to C
- C sends `inc-dec` message to A
- A increments reference count in local entry
- A sends second-stage of `inc-dec` message to B
- B decrements local copy count

In contrast, when a duplicate copied reference is received a `dec` message is sent directly to the source of the reference, as in standard IRC.

There are a number of fundamental differences between Moreau's algorithm and the optimised IRC presented here, in terms of their behaviour and implementation:

The relative laziness of the two algorithms depends more on the underlying implementation than on any implicit or explicit feature of their core design. Moreau's `inc-dec` messages could be sent immediately, or lazily piggy-backed on other traffic with the implicit diffusion tree temporarily represented by the queued messages. Similarly, in the algorithm presented in this paper, sub-tree re-rooting can await normal application invocations, or a daemon could generate dummy calls to eagerly resolve proxies with differing source and primary pointers.

The precise messaging costs of the two approaches depend crucially on the degree of laziness and hence are difficult to quantify in general. Moreau's `inc-dec` message roughly corresponds to the re-rooting request & reply bits and the subsequent `dec` message sent to an ex-parent following successful sub-tree re-rooting during RPC. Moreau's approach therefore introduces or affects two messages, rather than three in similarly eager or lazy implementations of the optimised IRC. However the size of the piggy-backed data is negligible in two of the optimised IRC interactions.

The lack of source pointers reduces the size of both marshalled references and proxies, in Moreau's algorithm, compared with the technique suggested here. In contrast, however, his `inc-dec` messages are somewhat larger than the normal `dec` messages used in both proposals.

Moreau's technique reintroduces a form of increment message, but avoids one of the major race conditions between increment and decrement messages by chaining the sending of the second-stage `dec` message after the processing of the increment triggered by the first-stage `inc-dec` message. A second potential cause for concern, loss of the second-stage of the `inc-dec` message, is also handled safely, since the local copy count at the source of the copied reference will remain incremented. This means the proxy at that site will be retained in perpetuity, so the algorithm fails safe.

The most important difference relates to fault-tolerance and the underlying model of communication:

The use of a form of increment message leaves Moreau's algorithm vulnerable to message reordering and loss. If, for example, a `dec` message from C overtakes the first-stage `inc-dec` message, the target object may be wrongly garbage collected. A similar problem arises if the `inc-dec` message is lost during the first stage of its travels and a `dec` message is subsequently generated. Moreau therefore has to assume that individual message channels are FIFO and that first-stage `inc-dec` messages are not lost. This assumption is embodied in his requirement that *"reliable message-passing and FIFO handling is provided by the transport layer"*. This is a significant restriction, limiting the applicability of Moreau's algorithm. With reliable FIFO channels most (but not all) of the problems of naïve reference counting are eliminated.

Such assumptions are not required by the variant IRC algorithm presented in this paper.

7. FUTURE WORK

This paper has presented, in detail, the key idea and implementation details that enable optimisation of Indirect Reference Counting. Several distinct developments are possible, building on this work.

7.1 Mobile Objects

Some care must be taken when modifying the IRC diffusion trees if objects are mobile, however the technique proposed is safe in this context. Paper length restrictions preclude detailed explanation, but the key issue is the value used for the diffusion tree depth indicator when a proxy is created for an outgoing migrating object. A negative value, based on the number of migration steps taken by the object, gives the desired behaviour, but may eventually stick at the limit of the counter.

7.2 Other Applications

The potential benefits of diffusion tree restructuring can also arise with other diffusion tree based algorithms, such as Dijkstra Scholten termination detection [6]. The costs and benefits will again always be highly application dependent.

7.3 Performance Evaluation & Proof

Minor algorithmic changes, such as proposed here, benefit from very careful evaluation in a controlled context using fine-grained measurements and a range of distributed applications. Since the costs are due to changing the pattern of context switches, introducing single bits into RPC calls, and other such minor modifications it is difficult to accurately characterise them amongst the noise of distributed computing. If a Distributed GC Algorithm Testbed were constructed [9], evaluating this variant IRC algorithm might provide an interesting sensitivity experiment. In the absence of such an environment for DGC evaluation it remains to implement the algorithm in a variety of systems, and to formally prove its properties, e.g. that integrity of the diffusion tree is maintained.

7.4 Further Integration and Hybridisation

A fusing of Weighted Reference Counting and Indirect Reference Counting would appear to be very promising, given the clear benefits indicated by the use of simple weighted proxies, and Corporaal's use of diffusion trees to manage weight underflow.

Constructing a hybrid collector, combining this acyclic distributed collector with a high quality local GC algorithm and a full distributed cycle-collector would provide further evidence of the applicability of the techniques.

Finally, it would be interesting to apply this variant in a reference listing system, hybridising with a cycle collecting distributed garbage collector, such as the timestamp propagator of Le Fessant, Piumarta and Shapiro [10]. Such a hybrid would illustrate the applicability of these techniques to reference listing, and generalising to such an environment (as also proposed by Moreau for his algorithm) could offer further insights into the implementation of these mechanisms.

7.5 Sub-tree re-rooting for parameters

In this paper sub-tree re-rooting is only applied when an invocation is made through a proxy. An anonymous reviewer noted that the optimisation can also be applied whenever a marshalled reference is sent as an RPC argument to its primary target node. This does indeed potentially allow additional re-rooting and reduction in the diffusion tree depth.

Such references already benefit from special handling in the lower levels of the RPC, since they can be unmarshalled as local references in the recipient of the call. Applying an equivalent of the re-rooting optimisation simply requires that a further bit be set in the message for each such reference, and a vector of bits corresponding to these "same target" reference arguments be carried by the reply. Note, however, that during the unmarshalling of the reply it will be necessary for the RPC layer to know which "same target" reference arguments were passed in the call. Further work is required to determine whether the increased complexity and costs in the RPC layer are justified by the benefits gained.

8. SUMMARY

In this paper a new variant of Indirect Reference Counting (IRC) has been proposed, which uses a dynamic restructuring of the reference diffusion trees to reduce their depth, with the goal of reducing the tree sizes. Decreasing the diffusion tree depth may reduce the probability of IRC failing safe due to node failures, and hence reduces the frequency with which alternative, more expensive, algorithms are required to garbage collect objects. Four mechanisms are used to achieve the low-cost tree reshaping and the same techniques can be applied to other diffusion tree based algorithms with the same potential benefit of reduced tree depth and increased fault tolerance. The algorithm behaves safely if messages are lost or reordered, or processes fail. It correctly collects all acyclic garbage provided the only complication is message reordering, and reduces the likelihood of the GC failing safe (compared with standard IRC) if third-party nodes fail.

9. ACKNOWLEDGEMENTS

Thanks are due to Huw Evans, Rolf Neugebauer and the anonymous reviewers for their helpful comments on drafts of this paper, and to members of the Long-Lived Systems Group at Glasgow for feedback at an early stage in the work on the IRC variant. Special thanks also to Jenn Logan for her encouragement throughout the development of the ideas and the writing of this paper.

10. REFERENCES

[1] David I. Bevan: *Distributed Garbage Collection using Reference Counting* in the proceedings of PARLE'87, available as LNCS volume 259, Springer Verlag; 1987

[2] David I. Bevan: *An Efficient Reference Counting solution to the Distributed Garbage Collection problem* Parallel Computing, volume 9 #2; 1989

[3] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki & Edward Wobber: *Distributed Garbage Collection for Network Objects* Digital SRC Technical Report 116; 1993.

[4] H. Corporaal, T. Veldman & A.J. van de Goor: *An Efficient, Reference Weight-based Garbage Collection Method for Distributed Systems* in the proceedings of PARBASE-90, IEEE; 1990

[5] Peter Dickman: *Distributed Object Management in a Non-Small Graph of Autonomous Networks with few failures* Ph.D. dissertation, Cambridge; 1992

[6] E.W. Dijkstra & C.S. Scholten: *Termination Detection for Diffusing Computations* Information Processing Letters volume 11, #1; 1980

[7] Ben Goldberg: *Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme* in the proceedings of PLDI'89, available as ACM SIGPLAN Notices volume 24 #7; 1989

[8] Richard Jones, with Rafael Lins: *Garbage Collection* Wiley 1996, ISBN 0-471-94148-4

[9] Richard Jones: *Proposal for a Distributed Garbage Collector evaluation testbed* Private Communication, November 1999.

[10] Fabrice Le Fessant, Ian Piumarta & Marc Shapiro: *An implementation for complete asynchronous distributed garbage collection* in the proceedings of PLDI'98, available as ACM SIGPLAN Notices, volume 33 #6; 1998

[11] Sylvain Louboutin: *A Reactive Approach to Comprehensive Global Garbage Detection* Ph.D. dissertation, Trinity College Dublin; 1997

[12] Nancy A. Lynch: *Distributed Algorithms* Morgan Kaufmann 1996, ISBN 1-55860-348-4

[13] Luc Moreau: *A Distributed Garbage Collector with Diffusion Tree Reorganisation and Mobile Objects* in the proceedings of ICFP'98, available as ACM SIGPLAN Notices, volume 34 #1; 1999

[14] Luc Moreau: *Hierarchical Distributed Reference Counting* in the proceedings of ISMM'98, available as ACM SIGPLAN Notices, volume 34 #3; 1999

[15] José M. Piquer: *Indirect Reference Counting: A distributed garbage collection algorithm* in the proceedings of PARLE'91, available as LNCS volume 505, Springer Verlag; 1991

[16] José M. Piquer: *Indirect Mark and Sweep: A Distributed GC* in the proceedings of IWMM'95, available as LNCS volume 986, Springer Verlag; 1995

[17] José M. Piquer: *Indirect Distributed Garbage Collection: Handling Object Migration* ACM Transactions on Programming Languages & Systems, volume 18 #5; 1996

[18] David Plainfossé & Marc Shapiro: *A Survey of Distributed Garbage Collection Techniques* in the proceedings of IWMM'95, available as LNCS volume 986, Springer Verlag; 1995

[19] Marc Shapiro: *Structure and encapsulation in distributed systems: the Proxy Principle* in the proceedings of ICDCS'86, IEEE; 1986

[20] Marc Shapiro, Peter Dickman & David Plainfossé: *Distributed References and Acyclic Garbage Collection* in the proceedings of PODC'92, ACM; 1992

[21] Sun MicroSystems: *Java Remote Method Invocation Specification*; 1996

[22] Gerard Tel: *Introduction to Distributed Algorithms* Cambridge 1994, ISBN 0-521-47069-2

[23] Paul Watson & Ian Watson: *An Efficient Garbage Collection Scheme for Parallel Computer Architectures* in the proceedings of PARLE'87, available as LNCS volume 259, Springer Verlag; 1987