Distributed Garbage Collection in a Client-Server, Transactional, Persistent Object System

by

Umesh Maheshwari

B.Tech., Indian Institute of Technology, Delhi, 1990

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

February 1993

Author . . . .
Department of Electrical Engineering and Computer Science
February 8, 1993

Certified by . . . .
Barbara H. Liskov
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by . . . .
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Distributed Garbage Collection
in a Client-Server, Transactional, Persistent Object System

by
Umesh Maheshwari

Submitted to the Department of Electrical Engineering and Computer Science
on February 8, 1993, in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

We present a design for distributed garbage collection in a new object-oriented database system called Thor. Garbage collection in Thor is different from that in conventional distributed systems because Thor has a client-server architecture, in which clients fetch copies of objects from multiple servers and run transactions. Our design accounts for the caching and prefetching of objects done by the clients. It also accounts for the distributed commit protocol, which involves the transfer of modified objects from the client cache back to the servers.

The scalability of Thor precludes the use of global mechanisms. Therefore, our design is based on each server keeping a conservative record of incoming references from clients and other servers; this allows fast and fault-tolerant collection of most garbage. The performance requirement faced by the design is that it minimize the delay added to fetch and commit operations invoked by clients. We have devised techniques that eliminate major overheads when a client fetches a block of objects from a server: no extra messages need be sent, the server need not record the references contained in the objects in the block, and no stable-storage write is required. However, when a client commits a transaction, extra messages and stable-storage writes may be required for garbage collection purposes. We propose a scheme that masks the delay by performing garbage collection work in parallel with normal commit-time work. The rest of the distributed garbage collection protocol is devised so that it works in the background; the protocol uses only unreliable messages and tolerates node crashes and network partitions.

Since our design is based on keeping a record of remote references, it does not collect distributed cyclic garbage. The thesis includes a discussion of various techniques that can be used to augment such a design to collect all garbage. The thesis also contains a novel analysis of a wide range of distributed garbage collection algorithms.

Thesis Supervisor: Barbara H. Liskov
Title: NEC Professor of Software Science and Engineering

Distributed Garbage Collection
in a Client-Server, Transactional, Persistent Object System
by
Umesh Maheshwari

Submitted to the Department of Electrical Engineering and Computer Science
on February 8, 1993, in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

We present a design for distributed garbage collection in a new object-oriented database system called Thor. Garbage collection in Thor is different from that in conventional distributed systems because Thor has a client-server architecture, in which clients fetch copies of objects from multiple servers and run transactions. Our design accounts for the caching and prefetching of objects done by the clients. It also accounts for the distributed commit protocol, which involves the transfer of modified objects from the client cache back to the servers.

The scalability of Thor precludes the use of global mechanisms. Therefore, our design is based on each server keeping a conservative record of incoming references from clients and other servers; this allows fast and fault-tolerant collection of most garbage. The performance requirement faced by the design is that it minimize the delay added to fetch and commit operations invoked by clients. We have devised techniques that eliminate major overheads when a client fetches a block of objects from a server: no extra messages need be sent, the server need not record the references contained in the objects in the block, and no stable-storage write is required. However, when a client commits a transaction, extra messages and stable-storage writes may be required for garbage collection purposes. We propose a scheme that masks the delay by performing garbage collection work in parallel with normal commit-time work. The rest of the distributed garbage collection protocol is devised so that it works in the background; the protocol uses only unreliable messages and tolerates node crashes and network partitions.

Since our design is based on keeping a record of remote references, it does not collect distributed cyclic garbage. The thesis includes a discussion of various techniques that can be used to augment such a design to collect all garbage. The thesis also contains a novel analysis of a wide range of distributed garbage collection algorithms.

Thesis Supervisor: Barbara H. Liskov
Title: NEC Professor of Software Science and Engineering

Acknowledgements

My research advisor, Barbara Liskov, laid down the framework for the design presented in this thesis. I feel fortunate that I could use her expertise in editing to correct my style of expression. In the last few days before the deadline to submit the thesis, she went out of her way to help me finish in time. She has been very kind throughout.

Mark Day, my officemate, put in a lot of effort to proofread all chapters despite a busy schedule of his own. His criticism has been priceless in improving the presentation of this work (he made mincedmeat out of some of the drafts I gave him). He and the other officemate, Andrew Myers, have been an incessant source of humor and lively discussions. Jim O' Toole proofread part of the thesis.

My elder brother, Dinesh, provided telephonic counsel and support from across the country.

Contents

List of Figures

Chapter 1

Introduction

We present a design for distributed garbage collection in a new object-oriented database system called *Thor*. Garbage collection in Thor is different from that in conventional distributed systems because Thor has a client-server architecture, in which clients fetch copies of objects from multiple servers and run transactions.

So that the problem may be specified precisely, Section 1.1 establishes the terminology and the basic concepts that will be used in this thesis. Section 1.2 extends the setting to a distributed system and states what is expected of distributed garbage collection. Section 1.3 describes what is different about a client-server, transactional, database system that poses new challenges for distributed garbage collection. Section 1.4 summarizes what the design proposed in the thesis has accomplished. An outline of the remaining chapters is provided at the end.

1.1 Background and Basic Terminology

Systems that make heavy use of allocating objects on the heap need to recycle the space. The hazards of explicit deallocation of unusable objects are well known: premature deallocation of a useful object leads to the "dangling pointers" problem, and forgetting to deallocate a useless object leads to "memory leaks." Garbage collection, popularly known as *GC*, automates the deallocation to free the user from tracking the usage of space. The part of the running process that allocates new objects and modifies them is often called the *mutator*, while the part that automatically deallocates the garbage is called the *collector*.

This automation can be achieved in different ways, depending on what is deemed to be "useful." Sometimes, it may be possible to have the compiler figure it out by analyzing the program. But the most general and widely used criterion for usefulness is based on run-time accessibility by the user program.

Objects contain references to each other, thus forming an *object-graph*. At any time, the user program is directly using certain objects, such as those referenced from the stack, which must not be deallocated. These objects are known as the *roots*. The objects reachable from the roots through one or more references are also accessible by the user program. All such objects are said to be *live*; the rest are *garbage*, which should be reclaimed.

Being a garbage object is a stable property; that is, once an object becomes garbage, it cannot be live again. The safety requirement for the GC is that it may not collect a live object. The liveness requirement is that any garbage should ultimately be collected. This requirement is especially important for long-lived spaces like databases, since otherwise memory leaks might accumulate over time and exhaust the space.

1.2 GC in Distributed Systems

A distributed system is a group of computers (*nodes*) that communicate by sending messages. It is characterized by a relatively high cost of message passing and occurrence of partial failures. In a distributed system the object graph spans multiple nodes. References crossing node boundaries are called *remote*, while those within one node are *local*. Spatial locality in the object graph is expected to result in many fewer remote references than local ones.

The mutator on a node sends messages to other mutators in addition to doing local computation. These *mutator messages* transfer data, which may include references to objects. The node sending the message will be referred to as the *sender*, the node receiving it, the *receiver*. The object to which a reference in the message points may be on yet another node, called the *owner* (Figure 1-1). For the rest of this chapter, assume that the mutator message is carrying only one reference. On receipt of the message, the receiver's mutator may store the reference in a local object, thus creating a new inter-node reference.[1] In some systems, mutator messages may also transfer objects from one node to another; this is called *migration*.

Like any other distributed program, the GC in a distributed system should take care of the following:

1. Use local computation to avoid message passing. Messages reduce throughput by consuming processor and network bandwidth; moreover, *synchronous* message-sends delay the sender by at least a round trip on the network, and raise problems about what must be done if the intended receiver is inaccessible. A noteworthy exception to the rule is that a round trip on the network may be preferred to accessing the local disk, which is usually even slower.

2. Allow graceful degradation of service in the face of partial failures, often called *fault-tolerance* in this context. Partial failures include crashes of individual nodes, and failures

[1] In the literature, authors sometimes distinguish between *creation* of the first copy of the remote reference by the owner, and its subsequent *duplication* by other senders [Piq91]. This thesis, however, does not distinguish between the two.

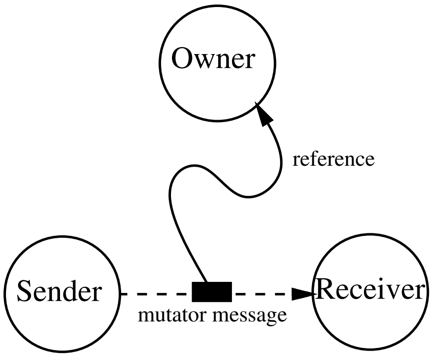


Figure 1-1: A mutator message containing a remote reference

in message delivery: messages can be delayed, lost, duplicated, and delivered out of order, or there might be a network partition, in which a group of nodes becomes virtually disconnected from the rest.

3. Employ mechanisms that are scalable, that is, whose performance does not worsen unduly as more nodes are added to the distributed system. This requires that the chosen mechanisms have minimum dependence on limited resources that do not grow as the system gets bigger.

The need to avoid message passing implies that a scheme designed for single-site GC cannot be translated for distributed systems by simply inserting synchronous communication wherever required. Similarly, schemes requiring some kind of global synchronization are not favored because they do not scale well.

One implication of fault tolerance is that even if some nodes crash, the remaining nodes should collect as much garbage as is possible without interaction with the crashed ones. Similarly, if there is a network partition, each subset of nodes should be able to collect some garbage. Failures in message delivery can be dealt with by using a generic reliable message protocol, but this is a costly solution that often requires one or more round trips per reliable message. The goal then is to design the GC such that messages are idempotent (so that duplicate messages are harmless), and nonessential (so that the loss of a message does not violate correctness, and is expected to be taken care of by later messages). Out of order messages can be serialized by using some variant of timestamping.

Broadly speaking, techniques for distributed GC fall under the same two paradigms as single-site techniques (more about this is presented in Section 2.1):

- *tracing*, also known as marking
- *reference tracking*, a generalized form of reference counting

Tracing performs a traversal of the entire object graph beginning from the roots, involving all nodes in the process. Objects that are not reached by the end of the traversal are garbage. Each run of tracing starts afresh in that it does not use information from earlier runs. It is global in nature since it must proceed to completion on all nodes before any node can collect any garbage. Thus, it might take a long time to before it collects any garbage, and it is not tolerant of node crashes.

On the other hand, if nodes track remote references as they get created by the mutator, each node can perform a local collection that collects most of the garbage, independent of other nodes. Nodes track remote references by storing all incoming references (or a conservative estimate thereof) in an *inlist*, and all outgoing references in an *outlist*. The local collection makes a conservative assumption that references in the inlist are reachable, and counts them as roots in addition to the "primary" roots we have been talking about. Unlike global tracing, reference tracking collects most of the garbage soon after it is created. However, such schemes have a well-known drawback: they fail to collect distributed circular garbage.

Both of these paradigms are analyzed in detail in Chapter 2, but the one adopted in this thesis is a form of reference tracking, so the rest of this chapter is set in a framework that assumes this paradigm.

Figure 1-2 provides an example of a distributed object graph. Objects A and D are the primary roots. Objects reachable from the primary roots are live, namely, A, B, C, and D. The remaining, which are garbage, can be sorted into different categories:

1. *local garbage*: an object that it reachable neither from the local primary roots nor from the inlist. In the figure, objects E and F are local garbage.
2. *distributed garbage*: an object that is not reachable from the local primary roots but is reachable from the inlist, and yet is garbage because all remote objects that reference it (if any) are garbage themselves. In the figure, objects G, H, I, J, and K are distributed garbage.
3. *distributed circular garbage*: distributed garbage containing a cycle of references that spans more than one node. For convenience, any garbage referenced from this kind of garbage is also included, because most of the time, what applies to circular garbage also applies to all garbage reachable from it. In the figure, objects I, J, and K are distributed circular garbage.

Having defined these terms, the division of labour between local and distributed GC can be clearly stated:

Local GC works within individual nodes, and regards the local primary roots and the inlist as its roots. It is responsible for detecting and deallocating local garbage, including entries in the outlist. Further, depending on the exact scheme employed, the local GCs may be required to store extra information and do extra work to assist the distributed GC.

Distributed GC is more like a protocol for exchanging information between local GCs. It is responsible for detecting distributed garbage and turning it into local garbage, which can be collected by local GC. More precisely, it uses the information in the outlists to remove the inlist entries of objects that are not referenced remotely. The flip side of its job is to protect

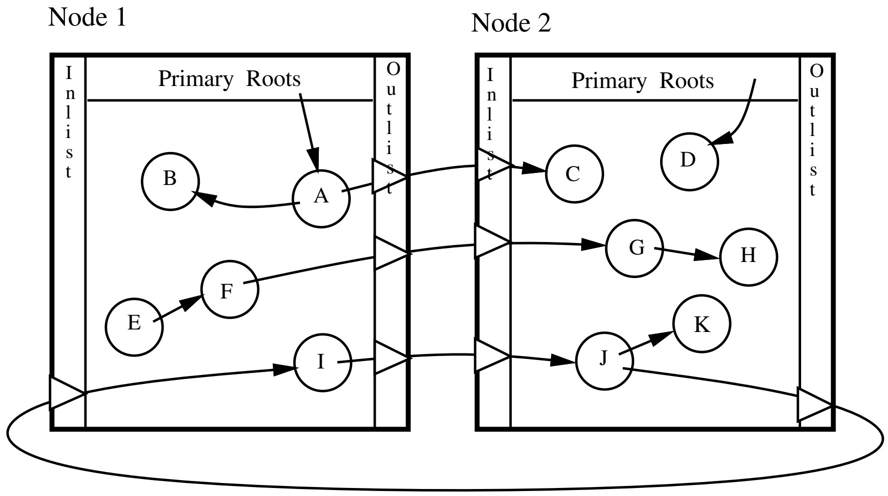


Figure 1-2: Garbage in a distributed object graph

the objects reachable from remote primary roots against local collection. This includes the timely creation of inlist and outlist entries in synchronization with mutator activities.

Usually, when the mutator sends messages, it cooperates by executing part of the distributed protocol. It has to make arrangements in anticipation of the creation of a new inter-node reference at the receiver. This may require sending extra *GC messages* to the owner node, although many schemes have been developed to avoid this (Section 2.4). GC messages are also required to inform nodes of the removal of inter-node references, but this can often be done by the collector lazily, in the background.

1.3 GC in a Client-Server, Transactional Database System

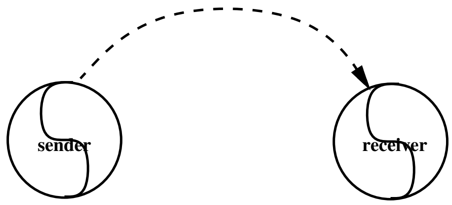
Thor is different from most other distributed systems for which GC schemes have been designed: it has a multiple-client, multiple-server architecture, and it uses distributed transactions, which complicates the way mutator messages transfer data. This section presents only a generic sketch of a client-server, transactional system; a more specific description of Thor itself is available in Chapter 3.

As discussed in the last section, in a conventional system, the transfer of data between the sender and the receiver nodes occurs through mutator messages flowing directly between them. This allows the sender to execute a necessary part of the distributed GC protocol, namely, handling the possible creation of a new inter-node reference at the receiver.

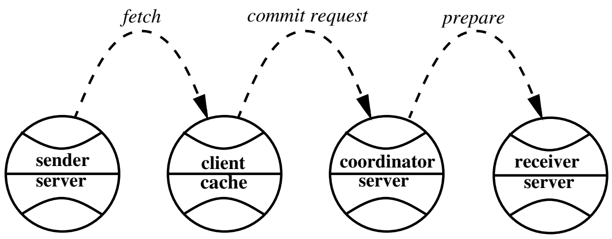
In a client-server system, the client fetches objects from various servers, and keeps them in a local cache. It works upon these objects by copying data between objects, removing data from objects, creating new objects, etc. In other words, objects are gathered from their servers and the transaction works upon them at the client. This is in contrast to the model in which the transaction moves from server to server via remote procedure calls while the objects stay put.

When the time comes to commit the transaction, the client sends copies of the modified and the new objects to a server designated as the *coordinator* of the transaction. It is the coordinator's responsibility to execute the 2-phase commit protocol [Gra78]. As a part of this protocol, it distributes the copies of modified and new objects to the servers where they belong. These servers, the *participants* of the 2-phase commit, incorporate the object copies after the commit succeeds.

A server that sends data to the client does not know which servers, if any, are going to receive some of it. The data hops from the sender to the client, then to the coordinator, and then to the receiver (Figure 1-3). It is possible that at commit time, the server that sent the data is not even a participant in the 2-phase commit protocol. Note that in reality any two or all three of the servers involved may be the same.



(A) A mutator message in a conventional distributed system



(B) Mutator messages in a client-server transactional system.

Figure 1-3: Thor is a different ball game.

The client is only a temporary link in the chain, which will go away when its work is done, while the servers are persistent repositories of objects. Thus, it is not possible to treat the transfer of data from the sender to the client in the same way as the transfer between two servers in conventional systems. Therefore, our distributed GC protocol has two separate components:

1. GC between client and server

2. GC between servers

There is another issue that is peculiar to persistent systems like databases. The effects of committed transactions survive crashes. Some of the information that supports distributed GC must survive crashes too, while the rest can be recomputed on recovery. Updates of the GC information may therefore incur stable-storage writes in addition to those required for durability of transactions [LS79]. The challenge is to reduce the amount of GC information that must be kept stably, without incurring a long wait on recovery to re-establish the remaining information.

1.4 The Thesis

The contribution of this thesis lies in the design of a distributed GC protocol that accounts for the caching of objects at clients as well as the commit-time transfer of modified copies of objects back to the servers.

The design is suited to scalable systems because it avoids the use of global mechanisms such as system-wide tracing. Instead, it is based on a form of reference tracking where each server keeps a conservative record of incoming references from clients and other servers. As noted earlier, reference tracking allows fast and fault-tolerant collection of most garbage; however, it does not collect distributed circular garbage.

We have devised techniques that eliminate major GC overheads when a client fetches a block of objects from a server: no extra messages need be sent, the server need not record the references contained in the objects in the block, and no stable-storage write is required. However, when a client commits a transaction, extra messages and stable-storage writes may be required for garbage collection purposes. We propose a scheme that masks the delay by performing garbage collection work in parallel with normal commit-time work.

The rest of the distributed garbage collection protocol is devised so that it works in the background; the protocol uses only unreliable messages and tolerates node crashes and network partitions.

The scope of this thesis is limited to the *distributed* part of GC; it does not prescribe any scheme for local collection in either the servers or the clients apart from specifying how it must cooperate with distributed collection. Further, the proposed design does not include a scheme for the collection of distributed circular garbage. It is possible, though admittedly tricky, to augment the design by either tracing [LQP92], or forced migration of certain objects [SGP90] to collect circular garbage. These schemes are discussed in Section 2.5.

1.5 Thesis Outline

Chapter 2 provides a study of various distributed GC algorithms. The schemes proposed in this thesis have borrowed many ideas from the analysis in this chapter. Chapter 3 describes Thor in greater detail, and introduces some terminology that is employed in later chapters. Chapter 4 provides an overview of the design proposed in this thesis, and then focuses on the GC protocol between clients and servers. Chapter 5 focuses on the GC protocol among servers. The last chapter summarizes the contributions of the thesis.

Chapter 2

An Analysis of Distributed GC Techniques

A study of the full spectrum of distributed GC techniques is helpful for a good understanding of what is applicable to Thor. Indeed, the design proposed in this thesis has borrowed many concepts from existing techniques. This chapter is not a comprehensive survey of the papers published to date on the topic of distributed GC: instead of summarizing who did what, it explores the how and why.

Broadly speaking, techniques for distributed GC fall under the same two paradigms as single-site techniques: tracing and reference counting. Section 2.1 provides a brief review of the pros and cons of these two paradigms in a non-distributed setting. Some of these observations are also applicable to distributed systems, only in a different flavor. Section 2.2 discusses distributed algorithms based on tracing. It argues that a pure form of tracing is unacceptable for distributed systems. Therefore, the rest of the chapter is related to different aspects of reference counting.

The first step in using distributed reference counting is to divide a large space into areas that can be collected somewhat independently. This is discussed in Section 2.3 without actually addressing issues pertinent to distributed systems. Most distributed GC algorithms use variants of reference counting; we refer to them by the general term: *reference tracking*. Section 2.4 describes a range of reference tracking techniques, and compares them with respect to message passing and fault tolerance. Reference tracking, by itself, does not collect distributed circular garbage (Section 1.2). Section 2.5 describes a variety of schemes that augment reference tracking to collect such garbage.

Figure 2-1 lists the techniques discussed in the paper. The figure includes some references for the purpose of easy identification — in view of the fact that some of the names listed have been invented by us.

In order to analyze these techniques we have recast the terminology used by their authors into our own. The resulting unification allows us to reveal how various algorithms stand in relation to each other, and to bring out their fundamental contributions. In fact, the decomposition of some of the

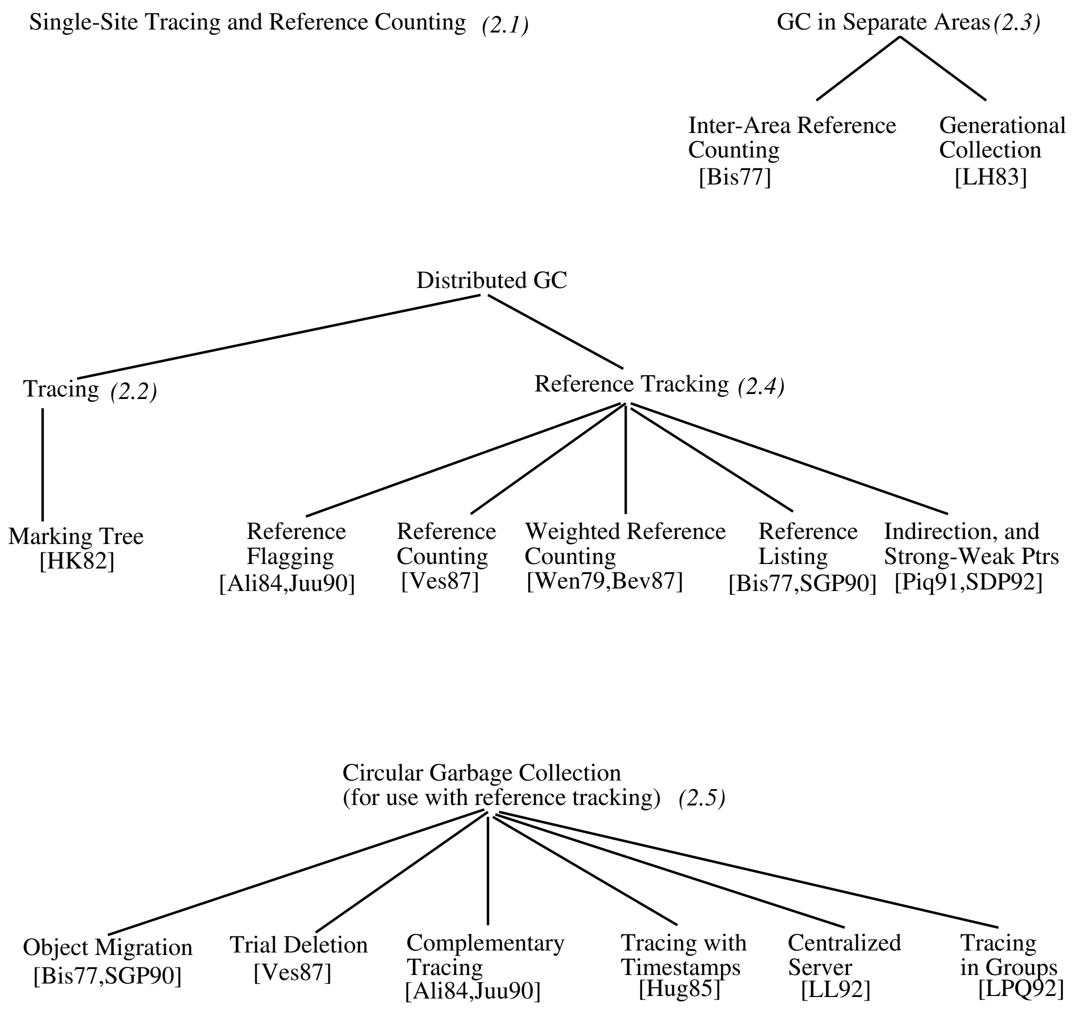


Figure 2-1: Techniques discussed in this chapter

rather involved algorithms into their key ingredients shows how they can be combined with other algorithms to obtain a better mix.

Towards the end of the discussion of each technique, we point out some of its drawbacks. In doing this, different techniques have been judged using somewhat different models; so an observation made for one may not be applicable to another. In particular, the criticism provided for a technique does not indicate much about its suitability for Thor.

2.1 Single-Site Tracing and Reference Counting

Algorithms based on tracing find live objects by directly implementing the definition of what is live: they traverse the object graph beginning from the roots. Each object reached during the traversal is recorded to be live and scanned for references to other objects. Objects that are not reached by the end of the traversal are garbage.

Two popular schemes of this kind are *mark-and-sweep* and *copy-collection* [Che70]. The first scheme records reachable objects by marking them, and collects all objects that remain unmarked into a free-list. It needs a *GC queue* (or a stack) to run a breadth-first (or depth first) marking algorithm. The GC queue holds references to objects from which marking is yet to be propagated further; it is initialized with the roots. The second scheme records reachable objects by copying them to a different space; when all reachable objects are copied, the old space can be fully reclaimed. Rather than creating a free-list, it *compacts* the live objects into one end of the new space, so that a contiguous space is available for allocation. Also, it does not need a separate queue because the queue is inherent in the placement of objects in the new space.

Reference counting uses a more conservative approach to reclaim garbage than tracing: all that is referenced from surviving objects survives itself. This fails to reclaim objects that reference one another but are really not reachable from the roots; we shall refer to them as *circular garbage*.

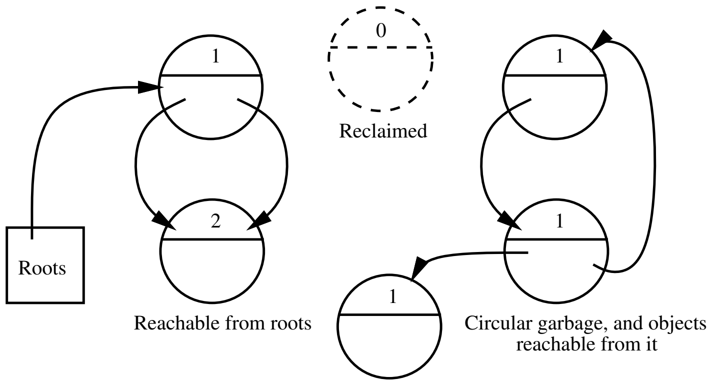


Figure 2-2: Reference Counting

As suggested by the name, a simple implementation is to keep a count for every object of references from other objects (Figure 2-2). The count is updated when a new reference to an object is created or when one is deleted. An object can be reclaimed when the count drops to zero. When an object is reclaimed, all of its references are considered deleted, which causes the counts of the referenced objects to be decremented.

2.1.1 The Pros and Cons

Reference counting runs finely interleaved with the user program, and identifies garbage as soon as it is created. On the other hand, tracing is *invoked* as a corrective measure when the free memory nears exhaustion, and it may lead to a long GC interruption unless an incremental or concurrent version is employed [DLMSS78].

A related point is that tracing must make a global search starting from the roots before any garbage can be collected. But in reference counting, liveness effectively becomes a *local property* of the object, namely, whether there are references to the object. (This is especially desirable in distributed systems, where global mechanisms are considered unscalable.)

Maintaining the reference counts is a substantial bookkeeping overhead. Updates must be made whenever a reference is created, deleted or replaced. In a system where references are frequently modified, a lot of work is wasted. In contrast, tracing does its work in one swoop, so its cost is amortized over the history of modifications made to the object graph since the last collection. However, if the object graph is rarely modified, and the rate of garbage production is low, tracing wastes its effort in traversing the entire graph which has changed very little since the last traversal. This is particularly costly for copy collection: even the part of the object graph that never changes keeps getting copied from one space to another. Schemes to overcome this problem divide the memory into separate spaces and use a hybrid of tracing and reference counting, which we discuss in Section 2.3.

The bane of reference counting is its inability to reclaim circular garbage. While some systems ignore the waste, and others use a system-specific remedy, such as declaring circular data structures in advance, the general solution falls back on a complementary tracing collection.

2.2 Tracing-based Distributed GC

Usually, each run of distributed tracing works in two phases: a global mark phase followed by a global sweep phase. In the mark phase, each node starts to mark object from its local roots. As in the non-distributed setting, when an object is marked, the references contained in it become candidates for marking. If such a reference is remote, a *marking message* must be sent to the remote node. When a node receives a marking message, it marks the indicated object. When marking is over at all nodes, and no mark messages are in transit, the nodes enter the sweep phase, wherein all unmarked objects are collected.

The tricky part of this protocol is to detect when the global marking phase has terminated. In single-site systems, the completion of the marking traversal is indicated by the GC queue going empty (Section 2.1). In distributed systems, each node has its own GC queue. The problem is that the emptiness of a local GC queue is not a stable property. That is, even after a GC queue is empty, an incoming marking message may initiate marking again. We describe one solution below.

2.2.1 Marking-Tree

This scheme is based on the distributed termination detection of "diffusive computation" [DS80]. For the time being, assume that there is only one root object. Also, for simplicity, the concept of marking messages is generalized to include local references: when an object is marked, it sends marking messages along every contained reference, remote or local [HK82].

When an object first gets a marking message, it remembers the sender object in its own *parent* field. Then it sends out a marking message for each of its contained references, and maintains a count of these messages in its *count* field. Every time it receives an *ack* back from a referenced object, it decrements the count. When the count falls to zero, that is, when the object has received acks for every referenced object, it sends back an ack to its own parent. If a marked object gets another marking message, it sends back an *ack* right away (Figure 2-3). The ack message from the root indicates the completion of marking. Note that the parent fields make a tree-like structure, called a *marking-tree* in [HK82], which replaces the GC queue. Multiple roots on different nodes can be handled by waiting until all roots have sent out an ack message. One node is selected as the *leader*, which checks for this condition and notifies all other nodes [Ali84].

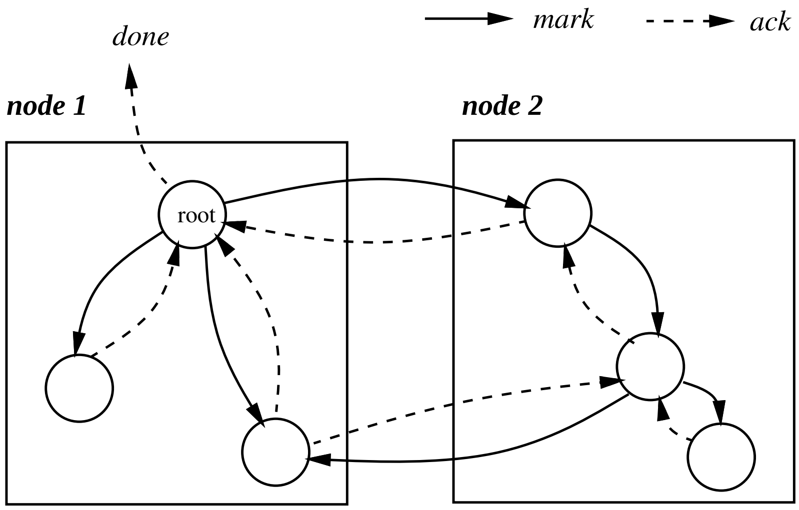


Figure 2-3: Marking-Tree for termination detection

2.2.2 Criticism

Some of the marking messages sent between two nodes can be *batched* together into one packet. For example, in Figure 2-4, $m1$ and $m2$ can be batched. But if there is a chain of references spanning the nodes k times, at least k marking messages must be sent back and forth between the two. Further, if a scheme like the marking-tree is used, an ack needs to be sent for every marking messages.

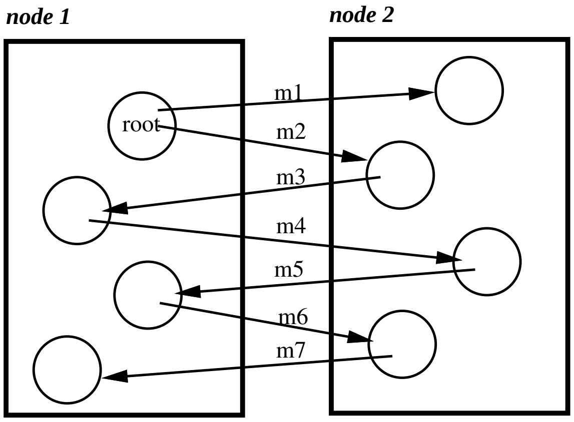


Figure 2-4: A chain of remote references results in a series of marking messages.

Tracing requires all nodes to cooperate. If even a single node is down or inaccessible, other nodes cannot be sure whether an object not reachable from their roots is reachable from the roots on the node that is down. Tracing is therefore not tolerant of crashes or partitions (but see Section 2.5.6). Also, the global synchronization required for phase changes makes it poor in scaling. A node must wait until all nodes have completed marking before it can collect *any* garbage. This might be unduly long if the system has many nodes, or even if some nodes are slow or non-cooperative.

2.3 GC in Separate Areas

Dividing the entire object space into *areas* that can be collected independently is the preliminary step in the direction of distributed reference counting. This section will ignore issues pertinent to distributed systems, such as message passing and fault-tolerance; they will be introduced in the next section. Indeed, the use of separate areas is also useful in singe-site systems with large address spaces [Bis77]:

1. The mutator need not wait for the collection of the entire address space. Having the collector run incrementally[1] or concurrently with the mutator does not solve the whole problem: it

[1] *i.e.*, incremental with respect to *time*, unlike collecting separate areas, which is incremental with respect to *space*

disperses a long GC pause into many shorter ones, but it does not reduce the total work to be done before garbage can be collected. Therefore, if a large space is to be collected, the mutator may run out of memory before the on-going run of collection is over.

2. More importantly, separate collection gives control over how frequently to traverse different parts of the object graph. As noted earlier in comparing reference counting with tracing (Section 2.1.1), some parts of the graph may remain unchanged for a long time, so that traversing them repeatedly is a waste, while other parts may change rapidly, providing a rich source of garbage.

3. In a system with a large heap in virtual memory, tracing the entire heap will result in many page faults. Dividing the heap into areas localizes the working set of the collector.

Note that full-scale reference counting represents the extreme in local collection as it allows individual objects to be collected separately; but that is usually not desirable because of the limitations of reference counting indicated in Section 2.1.1. We therefore describe a general scheme — a hybrid of tracing and reference counting — that is used to create arbitrary sized areas that can be collected separately. Next, we describe *generational collection*, which uses the lifetimes of objects as a metric to group objects into different areas and control the frequency of collection in them.

2.3.1 Inter-Area Reference Counting

While tracing is used to collect garbage within each area, some kind of reference counting is employed to insulate the GC in one from that in another. Each area keeps a record, called the *inlist*, of references from outside the area to objects within the area. For the time being, assume that the inlist maintains the *count* of inter-area references for each object in the area (no entry is kept for a zero count). The count must be updated as inter-area references are created or deleted. In a non-distributed system, updating the inlist entry in conjunction with creation or deletion of inter-area references is straightforward. An entry can be removed when its count drops to zero.

In some implementations, the inter-area references point to the inlist entry, which stores the count and points to the actual object. But this is *not* essential: a remote reference can directly point to the object, provided there is a way to access the object's inlist entry, if any. In this chapter, we will not attach any significance to whether a reference points to the inlist entry or to the object directly.

When an area is traced, the objects with an entry in the inlist are counted as roots in addition to the original root objects present in the area. We call the original root objects the *primary roots* to distinguish them from the inlist entries. Objects not reached by the trace are reclaimed. In tracing from both the primary roots and the inlist, the local GC makes a conservative assumption that all objects referenced from outside the area are reachable. This is what allows independent collection in different areas. But inter-area circular garbage is never collected (Figure 2-5).

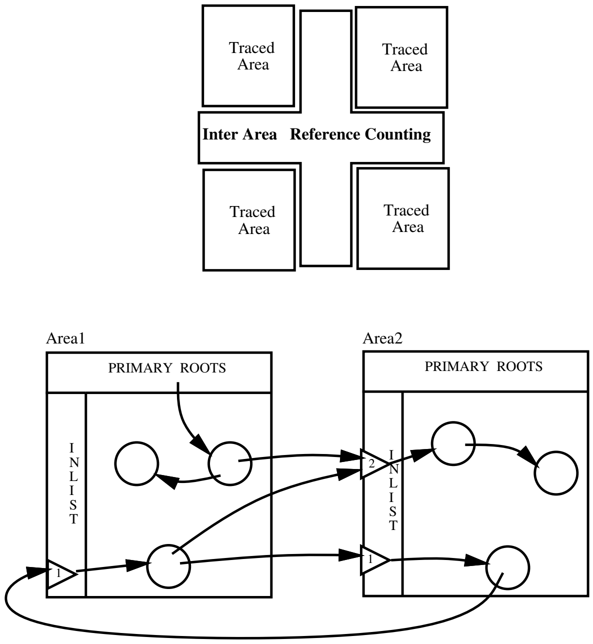


Figure 2-5: Inter-area Reference Counting

For the scheme to be efficient, the memory space must be partitioned in a way that individual areas exhibit spatial locality of references. This will result in fewer inter-area references, and less inter-area circular garbage.

2.3.2 Generational Collection

Generational collection optimizes the GC in separate areas by tuning it to the lifetimes of objects [LH83]. In doing so it exploits two facts:

1. Newly created objects have a higher chance of becoming garbage than those that have already survived many collections. The liveness pattern of heap objects is often similar to that of stack objects; that is, the objects created recently are likely to be done with sooner than others.

2. There are more references from new objects to older objects than the other way round. Old objects may refer to newer ones only if they have been mutated. Mutations are infrequent in many systems.

Objects are segregated into *generations* based on how long they have survived. We will talk of just two generations – *old* and *new*, but the scheme can be extended to any number of generations. Since the new generation is where most garbage is created, it is collected more frequently. also, since inter-area references are commonly oriented from the new to the old generation, they are

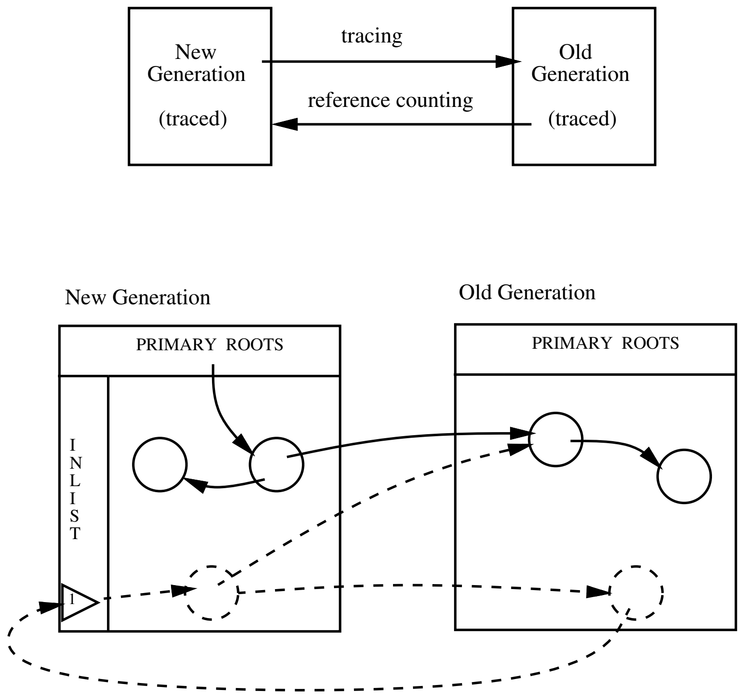


Figure 2-6: Generational collection collects inter-area circular garbage.

traced in that direction and reference-counted in the other (Figure 2-6). [2] References from the old generation to the new are recorded in the inlist for the new generation. If the assumptions hold, there would be few such references.

Allocations are made from the new generation, and when it fills up, it is collected independently by starting from its primary roots and the inlist. Typically, when an object in the new generation has survived some number of collections, it is *tenured* and moved to the old generation. To collect the old generation, the new and the old generations are traced together, starting from their primary roots (but not the inlist of the new generation). Since reference counting is used only in one direction, inter-area circular garbage is collected when tracing is used in the other direction.

2.4 Distributed Reference Tracking and Its Many Forms

Now we turn to reference tracking in distributed systems. Each node maintains its local space as a separate area — or more than one area, but we will discount this for simplicity. It tracks incoming remote references in some form or other. In general, an inlist entry contains a reference to the local object and some information about incoming remote references to that object. Different schemes

[2] This idea can be traced back to [Bis77], which uses *links* for reference counting in one direction, and *cables* for tracing in the other.

result in differences in the message passing protocol and fault tolerance.

When the mutator creates a new remote reference, or deletes one, extra messages may be required to update the remote inlist entry. We call the part of the distributed GC protocol that takes care of creation of references the *increase protocol*, and the one that takes care of deletion the *decrease protocol*. The increase protocol ensures the safety property that live objects will not be collected; it is therefore run by the mutator *before* creating a new inter-node reference. On the other hand, the decrease protocol ensures the liveness property that garbage will ultimately be reclaimed; therefore, it can be delayed and executed in the background.

The mutator may create a new remote reference by copying such a reference locally, or by sending it in a message to another node. It is unnecessary to run the increase protocol for local copying of references. Similarly, it is unnecessary to run the decrease protocol when a remote reference is deleted while there are other copies of it at the same node. Therefore, all remote references at a node that point to the same object are grouped together. We model this by having each node contain an *outlist* that stores a single entry for each outgoing remote reference. The remote references contained in objects are replaced by a local reference to the outlist entry that stores that remote reference (Figure 2-7). An useless outlist entry is collected by the local GC in the same way as it collects the objects. The interaction between the local and the distributed GC is illustrated in Figure 2-8.

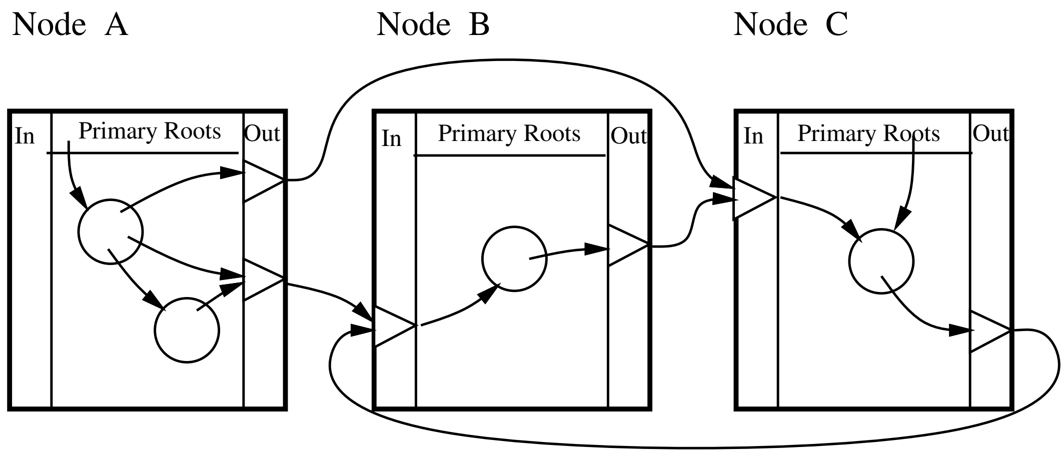


Figure 2-7: Inlists and outlists track inter-node references.

The increase protocol is run when a remote reference received in a message from another node creates a new outlist entry. The decrease protocol is run when the outlist entry is removed. The inlist entry for an object tracks the outlist entries pointing to it from other nodes.

The following subsections discuss various forms of reference tracking, focusing on their increase and decrease protocols.

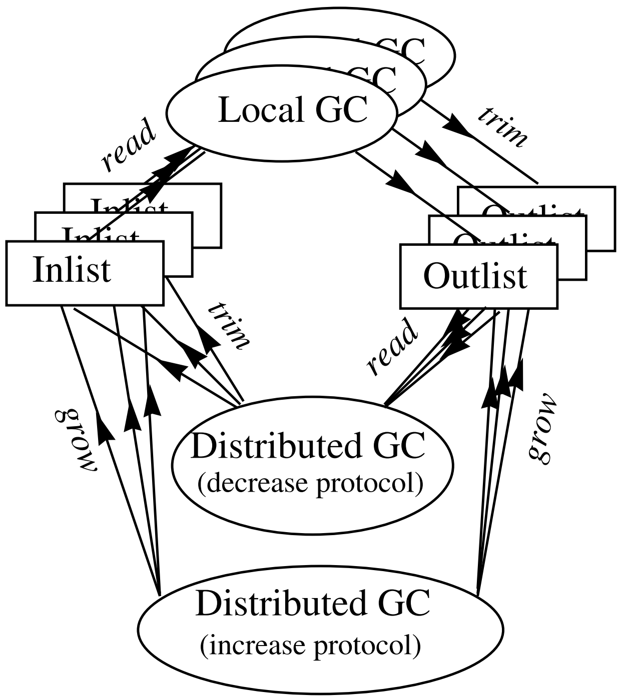


Figure 2-8: The roles of local and distributed GC in reference tracking

2.4.1 Reference Flagging

In this scheme, the only information kept is which local objects are referenced by one or more remote references [Ali84, Juu90]. Such objects are distinguished by the presence of an inlist entry; the entry does not contain any other information. This is similar to using a 1-bit wide reference count.

The inlist entry is created when the owner node first sends out a mutator message containing a reference to the object. Subsequently, if the reference is passed on to other nodes, the inlist entry is not affected at all. Thus, after the first send, the increase protocol does not involve any work.

Since an inlist entry does not track the spread of remote references to other nodes, the deletion of a remote reference does not affect it either: it is impossible to infer if there is no remote reference to the local object any more. Thus, once an inlist entry is created it cannot be removed without, say, help from periodic global tracing. This is ideal for systems that would have used a complementary tracing scheme anyway to collect circular garbage (Section 2.5.3). A run of global tracing collects all garbage, including useless inlist entries. As common to all reference tracking techniques, a node can do an independent local collection to collect local garbage.

The virtue of this scheme is simplicity: no additional work needs to be done in passing a remote reference or deleting one. Also, the inlist entries are compact. The problem is that it relies on global tracing to collect all distributed garbage — not just distributed *circular* garbage (Section 1.2).

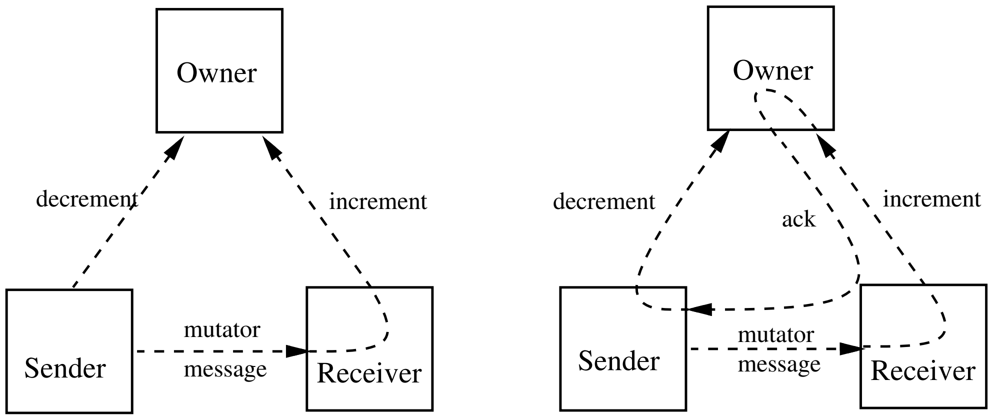
2.4.2 Reference Counting

This scheme has been described for the single-site, multiple-area setting in Section 2.3.1. Each inlist entry stores a count of the number of outlist entries (same as the number of nodes) that point to it. As part of the increase and decrease protocol, *increment* and *decrement messages* need to be sent to the owner node to update the inlist entry.

When a mutator message contains a remote reference, either the sender or the receiver could send the increment message. Ideally, we would like the receiver to send an increment message because one needs to be sent only if the receiver installed the remote reference into a local object and that resulted in a new outlist entry. However, this is not safe because a decrement message sent by the sender (after it sent the mutator message) may reach the owner before the increment message sent by the receiver, as shown in Figure 2-9 A(i). If the sender had the only remote reference to the object, this will reduce the reference count to zero, resulting in the collection of an object that is reachable from the receiver. Therefore, the sender must not send a decrement message for the reference in question until it has received an ack confirming the receipt of the increment message at the owner (Figure 2-9 A(ii)). But note that the sender is *not* blocked from doing other work in the meanwhile. The ack from the owner may come directly or via the receiver. Also, if the receiver is not going to send an increment message, it can send an ack to the sender straightaway.

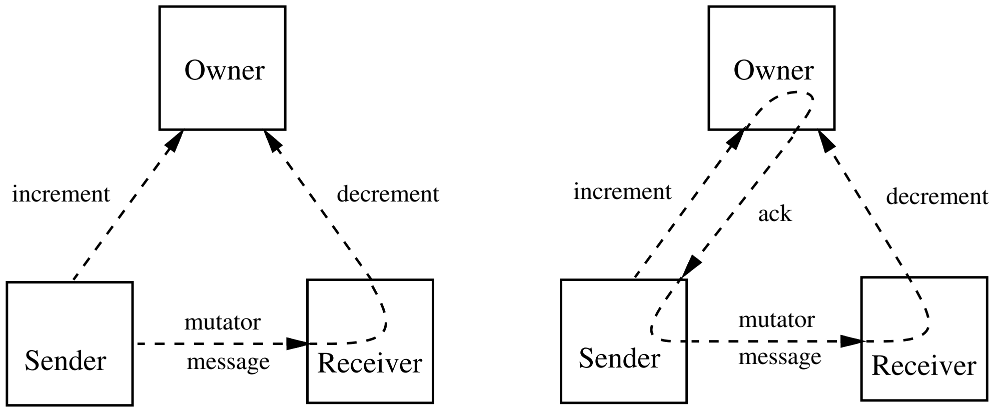
One way for the sender to refrain from sending a decrement message for a reference it sent out to another node — until it has received the ack — is to store all such references in a separate data structure, called the *translist*, whose entries include the receiver node to which the reference was sent. (The concept of a translist is borrowed from [LL86], where it is used in a somewhat different context.) Entries are deleted from the translist when the sender receives an ack from the receiver (or the owner, depending upon the exact protocol). When an outlist entry is deleted, a decrement message is deferred until the corresponding entry in the translist, if any, is deleted too. A different perspective helps to simplify this model: a translist entry can be viewed to be protecting the corresponding outlist entry, so that the outlist entry is not deleted as long as the translist entry exists. Further, since an outlist entry is normally protected by its reachability from the local roots, a translist entry can be viewed as a root with a reference to the corresponding outlist entry. This model helps to consolidate all outgoing remote references into the outlist instead of maintaining them in two separate lists. The two models are conceptually identical, but we shall use the "consolidated outlist" one because it is easier to discuss.

Now consider the case when the sender itself sends the increment message to the owner when it sends the mutator message to the receiver. When the receiver receives the remote reference, it may decide not to install it and send a decrement message to the owner. Again, the problem is that the decrement message might be received before the increment message from the sender, as shown in Figure 2-9 B(i). To avoid this problem, the sender must wait until it has received an ack confirming receipt of the insert message *before* it can even send the mutator message (Figure 2-9 B(ii)). Since this scheme adds a round trip delay before mutator messages can be sent, the earlier scheme is usually preferable.



(i) Possible race condition (ii) The remedy

(A) The receiver sends the increment message



(i) Possible race condition (ii) The remedy

(B) The sender sends the increment message

Figure 2-9: Race conditions between increment and decrement messages

Without the acks, the race conditions noted in the above two schemes can happen even if the links are reliable and deliver messages in FIFO order. The reason is that in the scenario depicted, the increment and decrement messages are sent on different channels. The solution therefore is to delay the decrement message until an ack for the increment message has been received from the owner.

The decrement message can be sent in the background. Note that both the increment and decrement messages must be sent on reliable channels that do not duplicate messages. Neither increment nor decrement is idempotent. If the same increment message is received twice by the owner, the reference count of the object will remain higher than its real value, and the object will never be collected. If the same decrement message is received twice, the reference count may drop to zero prematurely. Because it requires reliable message-sends (together with the extra ack messages), plain reference counting is not favored for distributed GC.

2.4.3 Weighted Reference Counting

This scheme is devised to avoid increment messages. It associates weights with remote references, which are stored in the outlist entries. The invariant maintained is that the sum of the weights in all outlist entries pointing to an object is the same as the weight in the inlist entry at the owner of that object, discounting decrement messages in transit [Wen79,Bev87].

When an inlist entry is first created, it is assigned the maximum weight, which is also the weight assigned to the outlist entry pointing to it. this is illustrated in Figure 2-10(A). Subsequently, when the holder of a remote reference passes it to another node, it divides the weight in its outlist entry into two parts, retains one part, and sends the other with the message. Usually, the weight is divided equally, but that is not necessary. The receiver creates an outlist entry for the reference with the weight included in the message, or, if it already had an outlist entry, adds the weight to the existing value. The effect is that the sum of weights associated with the reference remains unchanged (Figure 2-10(B)). Thus, the increase protocol involves no extra messages, only some local computation.

When a node removes an outlist entry, it sends a decrement message, including the entry's weight, to the owner. The owner subtracts the included weight from that in the inlist entry (Figure 2-10(C)). Note that after an inlist entry is created with the maximum weight, its weight never increases. If the weight drops to zero, the inlist entry can be collected. As in plain reference counting, the decrement message must be sent in reliable way — without loss or duplication.

The biggest problem with this scheme is that the weight in an outlist entry is halved every time it is copied into a mutator message. Thus, an initial weight of 2^k can be copied only k times before it falls to 1 and cannot be split any further. Of course, this will happen only if the copying of the reference was lop-sided — always occurring from a node that already had a low weight, but it is still a possibility. Several solutions have been suggested. One is that the sender sends an increment message to the owner, in response to which the owner increments the weight in the inlist entry by

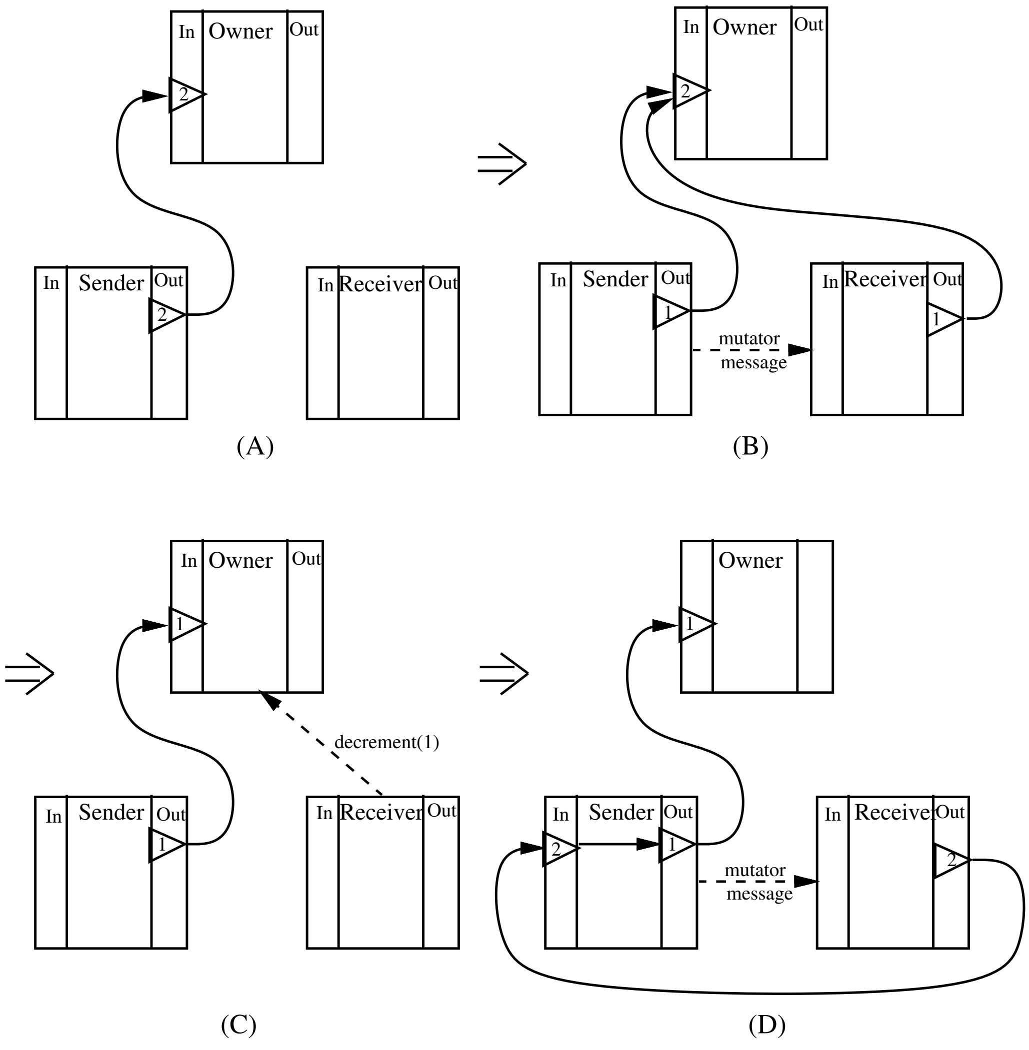


Figure 2-10: Weighted reference counting

some amount. When the sender receives an ack from the owner, it can increase the weight of its outlist entry by the same amount. An alternate solution, which avoids the synchronous increment message, is to use *indirection*: create a new inlist entry at the sender, initialized to the maximum weight, that points to the outlist entry (whose weight has dropped to 1), and send a reference pointing to the new inlist entry to the receiver (Figure 2-10(D)). When the receiver accesses its reference, which points to the sender, the sender redirects the access to the owner.

There is another scheme similar in spirit to weighted reference counting, called *generational reference counting* [Gol89], which is not described here. (Caution: the scheme is not related in any way to the generational collection described in Section 2.3.2.)

2.4.4 Reference Listing

Here, instead of keeping just a count, the inlist entry for an object keeps the *list* of all nodes that contain a remote reference to it [Bis77, SGP90]. Assume that the nodes are identified by distinct *node-ids*. It is common in the literature to break the inlist entry into separate elements, each containing a reference to the local object and a single node-id from where the object is referenced. In the context of reference listing, we will call each such element an inlist entry. The invariant maintained is that if a node has an outlist entry for an object, then that object has a *matching* inlist entry for the node. Instead of the increment and the decrement messages used in reference counting, a node sends *insert* and *delete* messages, which include the node's node-id.

As in the context of reference counting (Section 2.4.2), the receiver sends an asynchronous insert message to the owner, while the sender remembers the reference in transit in a translist until the delivery of the insert message is confirmed. ([SGP90] avoids the sending of synchronous insert messages by using another scheme, which involves indirection; see Section 2.4.5.) Upon receiving an insert message, the owner creates a new inlist entry of the referenced object for the indicated node, if one does not exist already. An entry in the translist protects the corresponding entry in the outlist, so that a delete message is not sent as long as the translist entry exists (Figure 2-11). Upon receiving a delete message, the owner deletes the entry for the indicated node, if one exists.

Reference listing improves fault-tolerance in message delivery. With the use of an additional technique, it does not matter if delete messages are lost. Recall that if a decrement message is lost, the associated object will never be collected. The technique is as follows:

1. Each node periodically sends messages to all nodes it contains references to. The message sent to a node includes all entries in the sender's outlist that refer to objects on that node. We call these messages *trim messages*.

2. When a node receives a trim message, it gathers the inlist entries for the sender node. By the invariant this scheme maintains, the part of the outlist sent by the sender is a subset of the part of the inlist at the owner. The inlist will have additional entries if some delete messages were lost, or were not sent. The owner can remove all such entries.

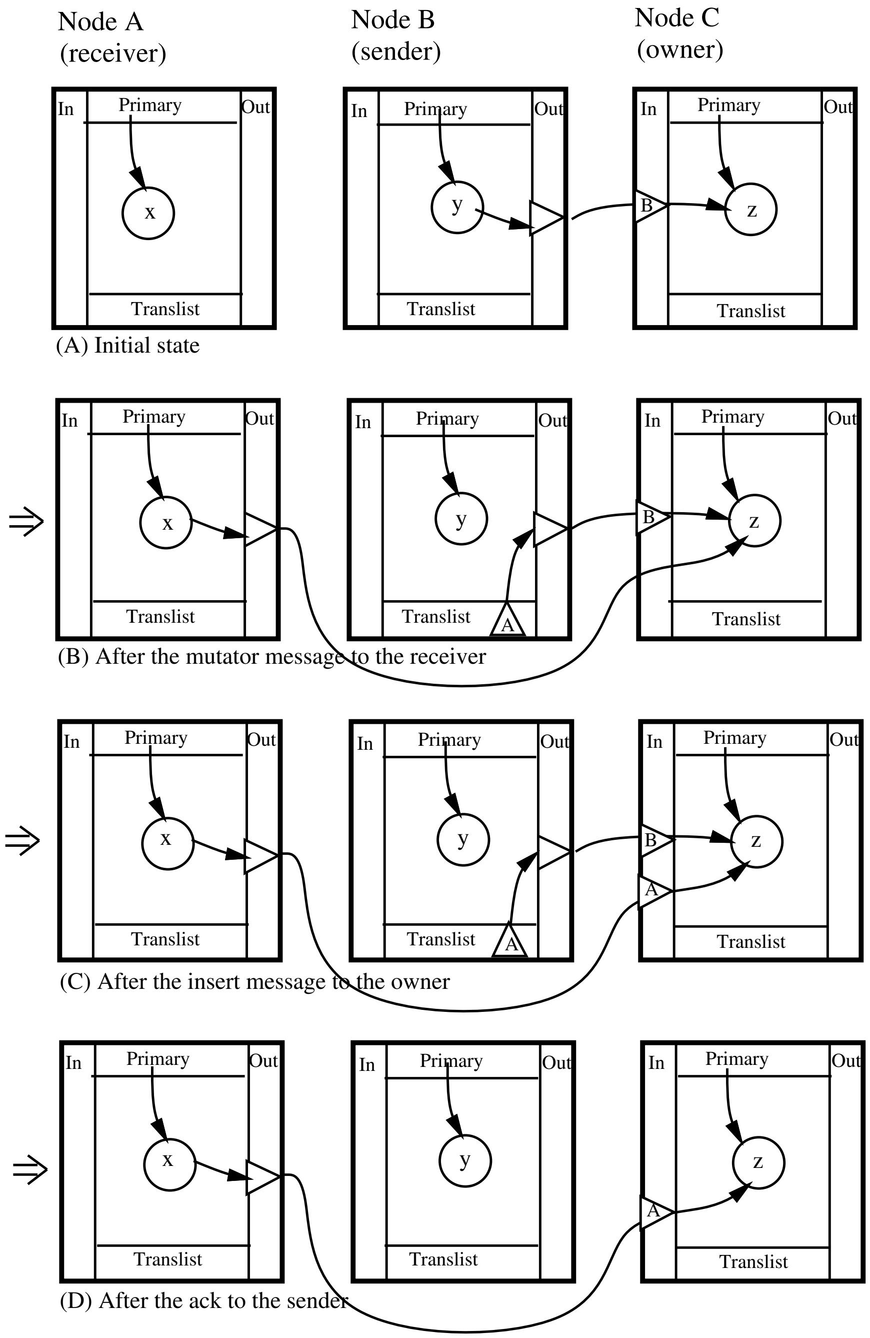


Figure 2-11: A translist entry exists until the delivery of the asynchronous insert message is confirmed.

In fact, the delete messages need not be sent at all provided trim messages are sent periodically. The advantage of sending trim messages instead is that they effectively batch together all earlier delete messages. No harm is done if a trim message is lost — the next one will do the work. One problem with a trim message, however, is that it must be sent in its entirety to be effective; that is, the message must contain *all* outlist entries at the sender pointing to the owner. This problem exists because the owner removes inlist entries if they are *absent* in the outlist.

Further, unlike increment and decrement messages, insert and delete (or trim) messages are idempotent, so accidental duplication does not hurt. However, delayed messages that get re-ordered behind messages sent later could still be problematic. A delayed duplicate insert message can re-insert an entry in the inlist after it had been deleted. But this is benign under the scheme described above, because a subsequent trim message will get rid of the entry.

However, a straggler delete (or trim) message is potentially unsafe. Suppose that an insert message was sent after the delete message to re-create the inlist entry. If the delete message reaches the owner *after* the insert message, the inlist entry will be deleted for good. One way to avoid this problem is to use timestamped insert and delete messages. The owner node stores the timestamp from the insert message in the inlist entry. A delete (or trim) message is effective only if it is timestamped higher than the inlist entry.

For this timestamping scheme to work, the timestamped insert and the delete messages must be sent by the same node — the receiver, which was the holder of that reference. It does not work if the inlist entry is timestamped by the owner instead. This happens, for instance, when an insert message from the receiver is suppressed because the sender is the owner itself. As before, the delete message is still timestamped by the node that held the reference, namely, the receiver. Since the clocks of the owner and the holder may not be synchronized, a somewhat more complicated protocol is required (see [SGP90]).

Another way in which reference listing provides extra fault-tolerance is that a node can send *query* messages to the nodes for which it has inlist entries. Such a message prompts the recipient to send a trim message to the owner. Alternatively, the owner may explicitly query about a particular remote reference.

The upshot of all this is that GC messages are sent in the background and do not require reliable delivery. One drawback of using reference listing is that it consumes more space than reference counts; this might be acceptable if remote references are rare.

2.4.5 Indirection, and Strong-Weak Pointers

The primary goal of the indirection technique, which was briefly described in the context of weighted reference counting (Figure 2-10(D)), is to avoid synchronous increment or insert messages. The key observation is that the sender already has an outlist entry that protects the remote inlist entry at the owner. Now, if the remote reference sent to the receiver is made to protect the outlist entry at the sender, the inlist entry at the owner will be protected indirectly. To this end, te sender passes the receiver a reference to its own outlist entry instead of a direct reference to the object.

The problem with using indirection is that if the receiver accesses the reference, it is indirected through the sender. One solution is to *snap* the indirection by communicating to the owner in the background [SGP90]. As soon as the receiver has its own inlist entry at the owner, it can switch its reference to point directly to the object. However, it is likely that the receiver will attempt to access the reference as soon as it obtains it from the sender, and if snapping has not occurred by then, the access will take an indirection.

Another solution eliminates the indirection in accesses completely. The sender sends a pair of references for each original reference: a *weak pointer* and a *strong pointer*, and the receiver stores both of them in its outlist entry. The strong pointer is used for protection against erroneous collection, while the weak pointer is used for direct access to the object.

If the sender is also the owner of the reference, both of the two pointers it sends to the receiver point to its object. If not, the sender must have a two-pointer outlist entry itself, which was created when *it* received the reference. It creates the two pointers for the receiver as follows:

1. It passes its own weak pointer as the weak pointer.

2. It passes a pointer to its own outlist entry as the strong pointer.

The above scheme maintains the invariant that the weak pointers point directly to the actual object, and a chain of strong pointers protect that object against collection. As an optimization, the strong pointers can be snapped in the background to point directly to the object after the owner has created a corresponding inlist entry. Note that the snapping is not essential for the performance of accesses: its only benefit is that it frees up the intermediate inlist and outlist entries, which can then be collected if nothing else references them. If the maintenance of such entries is not a big cost, the snapping is unnecessary, so the increase protocol does not involve any extra message.

The use of indirection and strong-weak pointers is illustrated in Figure 2-12. Both the strong and weak pointers of the outlist entry at the sender point to the actual object z because it was sent the reference by the owner itself (Figure 2-12 (A)). When the sender sends the reference to the receiver, it creates the two pointers for the receiver in accordance with the two rules given above. This protects the sender's outlist entry even when it deletes its own copy of the reference from object y (Figure 2-12 (B)). Later, in the background, a separate inlist entry is made for the receiver, and then its strong pointer can be changed to point directly to the object. This allows the collection of the outlist entry at the sender (Figure 2-12(C)).

[Piq91] uses this technique in conjunction with reference counting, although it is modeled differently. [SDP92] uses the technique in conjunction with reference listing.

One drawback of using strong-weak pointers is that every reference included in mutator messages actually occupies the size of two references. This is awkward if the mutator message is carrying

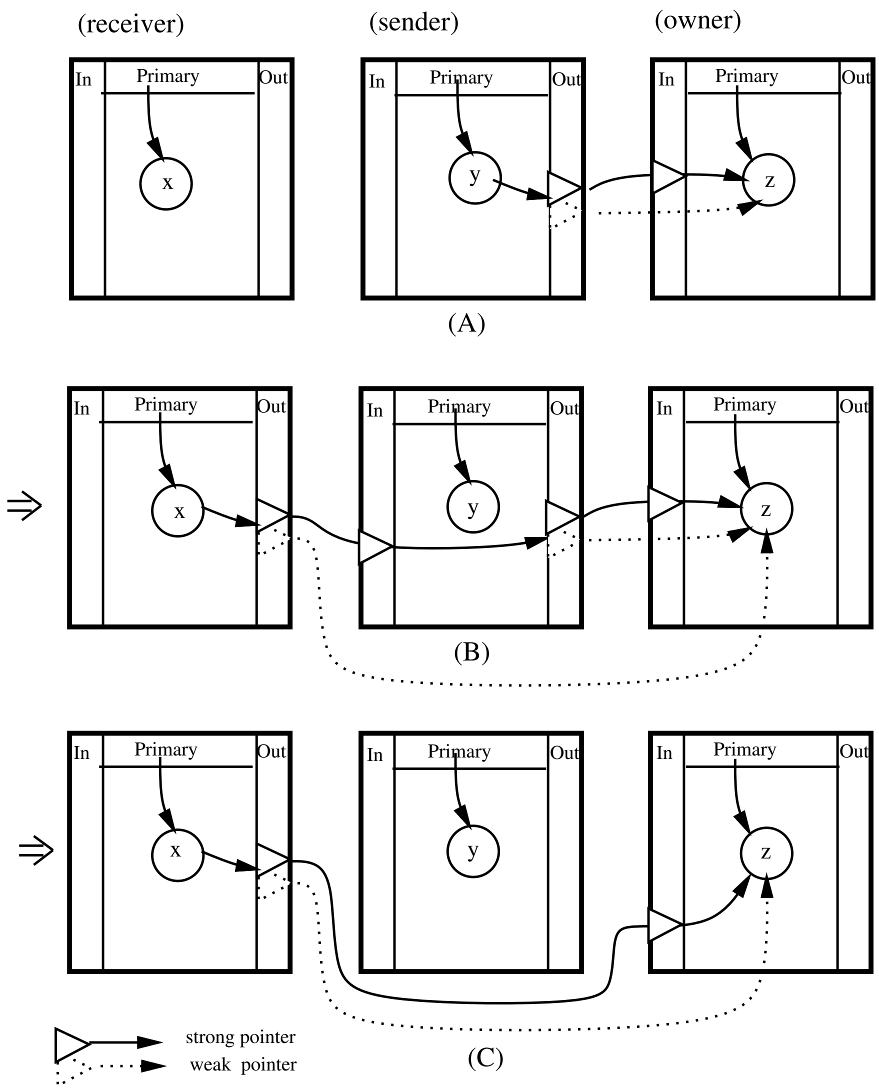


Figure 2-12: Indirection and strong-weak pointers

an object that contains references. It requires that the object be marshalled into a different format wherein the contained references are twice as big.

A comparison between the use of translist and strong-weak pointers is enlightening. Both avoid synchronous insert messages by securing the outlist entry at the sender. With the translist technique, the owner is sent an asynchronous insert message, and the outlist entry at the sender is protected by a translist entry until the delivery of the insert message is confirmed (Section 2.4.2). The asynchronous insert message is necessary to remove the translist entry: without it, the referenced object will not be collected because the outlist entry will persist for ever. With strong-weak pointers, the outlist entry is protected until the receiver holds the reference. This technique obviates the need to send insert messages altogether: the outlist entry is automatically unsecured when the receiver deletes its copy of the reference. Asynchronous insert messages may still be sent for possible reclamation of intermediate entries even *before* the referenced object becomes garbage, but all garbage will be collected even in the absence of such messages. An associated benefit of avoiding insert messages is that there is no danger of out-of-order delivery of insert and delete messages, which otherwise requires some kind of timestamping protocol to ensure correct order. However, it is not clear if this is a big enough advantage to outweigh the drawbacks of strong-weak pointers listed earlier.

2.5 Collection of Circular Garbage

Distributed reference tracking does not collect distributed circular garbage by itself (Section 1.2). Some systems can afford to ignore this kind of memory leak because they are short lived, or because distributed circular structures form only rarely in them. Others have to augment their reference tracking scheme to collect all garbage. Some of these techniques are described in the following subsections. It is assumed that any *local* circular garbage is collected by the local collector.

An important metric to keep in mind while judging a technique is the following property:

> The collection of a distributed cycle should not require the cooperation of any node other than those containing the cycle.

This property is desirable for fault tolerance in large distributed systems. With respect to Figure 2-13, the collection of the cycle that passes through Node B and C should not require cooperation from Node A.

2.5.1 Object Migration

The aim here is to consolidate an unreachable distributed cycle into a single area where it can be collected by the local collector [Bis77, SGP90]. This is implemented as follows: if an object is

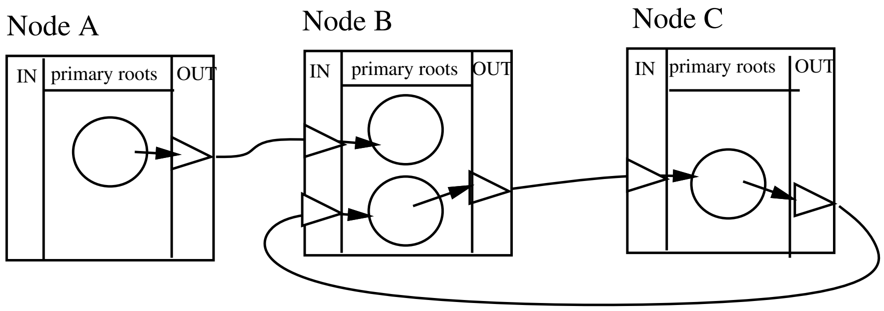


Figure 2-13: Collection of a cycle should not require all nodes to cooperate

unreachable from the local primary roots but is remotely referenced, it is moved to a node that references it. The local GC helps in finding such objects. As a part of local tracing, it first marks all objects reachable from only the primary roots. Any object that has an inlist entry and has not been marked is a candidate for migration to one of the nodes that reference it. This requires the inlist entry to maintain information about the remote nodes referencing it, as is done in reference listing (Section 2.4.4). All unmarked objects that are remotely referenced from the same node, and other unmarked objects reachable from them can be batched and sent together to that node (Figure 2-14).

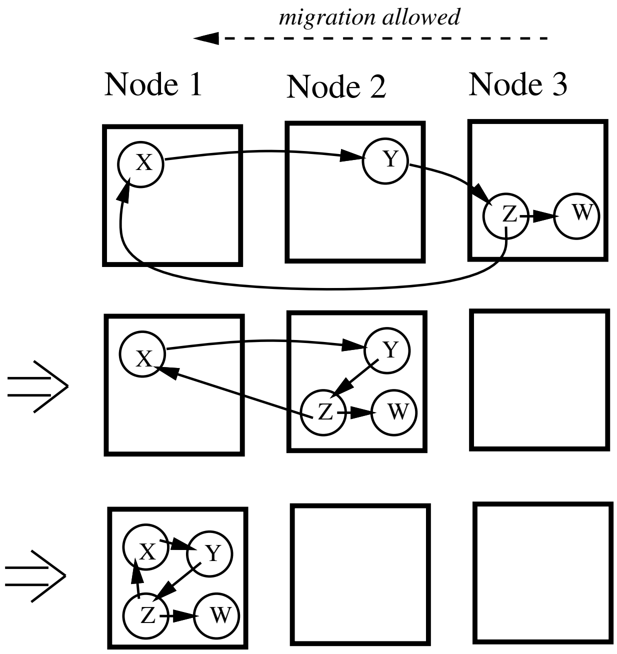


Figure 2-14: Consolidation of distributed circular garbage into one node

Now consider circular garbage comprising two objects on different nodes, each referencing the other. It is possible that the two objects might migrate to each other's node at the same time. To avoid this ping-pong effect, the nodes are totally ordered in some way and objects are allowed to migrate in one direction only.

The virtue of this scheme is that collection of a distributed cycle involves only pairwise communication among the owner nodes; it does not matter if other nodes are inaccessible. In fact, all of the nodes on the distributed cycle need not be up at the same time.

This scheme might migrates even those objects that are not garbage. This has a benign re-clustering effect in that objects settle down in nodes they are referenced from. But migration may be undesirable for several reasons:

1. Some heterogenous systems either do not allow migration or make it rather cumbersome.
2. If object locations are visible to users and can be controlled by them, they may not like automatic migration of objects.
3. Object migration is problematic if references are location dependent because the references pointing to the old location must be taken care of. A technique sometimes used is indirection: leave a *surrogate* at the old location that stores a reference to the new location [LDS92]. However, indirection will not help in consolidating the cycle because, in effect, the cycle still goes through the object's old node (Figure 2-15). Before migration can be effective in claiming circular garbage, the indirection must be snapped so that the references point to the actual object directly, and the surrogate must be collected so that there is no remote reference to the objects in the cycle.
4. Migration can create a load imbalance between nodes. This is because objects move only in the direction governed by the total ordering of nodes; the nodes at the target end cannot offload their objects to other nodes because those objects will migrate back to them.

Fixes have been suggested to the problems listed above. One of the schemes suggested in [Ves87] is to move the "logical structure" of an object, while its real physical copy remains fixed. Note that the logical structure needs to include contained references if the collection of cyclic garbage is to occur. The real copy is collected only if the logical copy proves to be garbage. The drawback here is the overhead and complexity of maintaining two copies. Another solution is to use "logical areas" [Bar88]. Instead of moving objects between nodes, the area boundaries are changed to include objects on other nodes. This means that the local collector of an area may have to reach other nodes to complete local GC; for example, it may involve sending some marking messages to remote objects (Section 2.2).

2.5.2 Trial Deletion

This scheme is based on the observation that circular garbage survives reference tracking because it is self-supportive. If an object is temporarily deleted for the sake of experiment and the reference

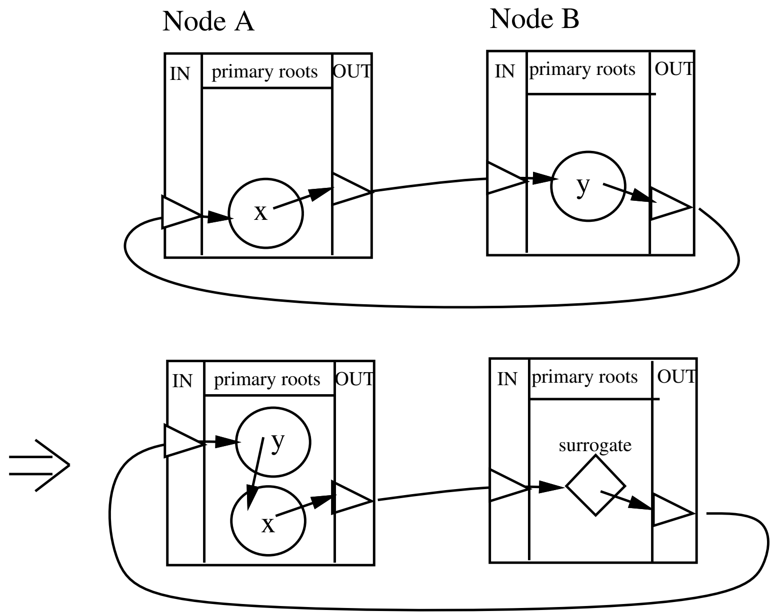


Figure 2-15: Treating migration using surrogates does not consolidate cycles by itself.

counts in the object graph adjusted accordingly, then if the reference count of the original object does drop to zero, it must be part of circular garbage [Ves87].

This scheme works well only if the local collection is also based on reference counting, because then it is simple to propagate the effects of an experimental deletion. It is possible to achieve the same effect with local collection based on tracing, but only at a high cost: propagating the trial deletion of an inlist entry to check which outlist entries get affected requires that the entire local space be traced afresh while ignoring the chosen inlist entry.

To help conduct this experiment without messing up the true reference counts, each object has two count fields: actual count and trial count. An object heuristically suspected to be on an unreachable distributed cycle is selected as the seed, and is temporarily deleted. Its trial count is initialized to its actual count (Figure 2-16(A)). The trial counts of all objects referenced from the temporarily deleted object are decremented. If the trial count of an object drops to zero, the object is temporarily deleted itself, and the process is repeated (Figure 2-16(B)). Note that as the updates spread to more objects, their trial counts are initialized from the actual count. If, ultimately, the trial count of the seed object drops to zero, the object, and all other objects whose trial count dropped to zero, can be collected (Figure 2-16(C)).

Since different nodes can initiate trial deletions on different seed objects simultaneously, objects actually needs to maintain a *list* of trial counts. Different trial counts may be distinguished by distinct stamps, say, the identity of the seed object.

As noted earlier, this scheme is suited only for reference-counting based local collection. Another

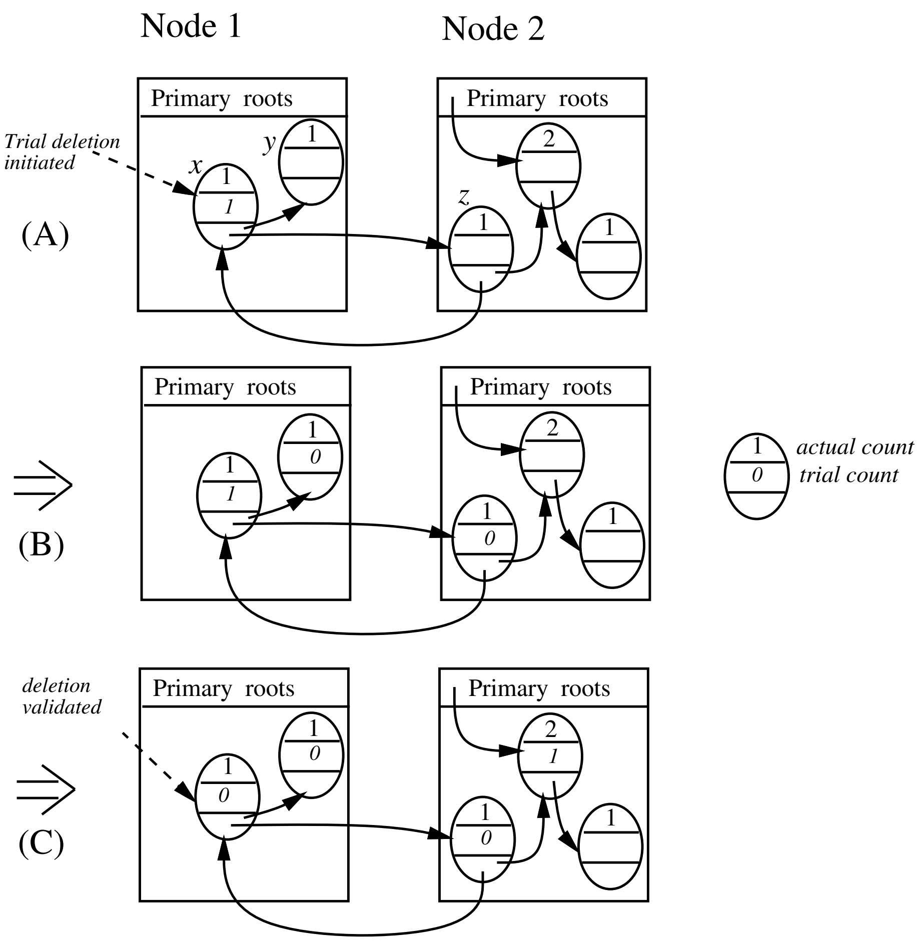


Figure 2-16: Trial deletion to detect circular garbage

big problem with this scheme is how to choose the seed object. One solution might be to select an object that is not reachable through primary roots, and has not been accessed for a long time. However, even if the chosen object *is* garbage, it could be either *on* a cycle, or it could be just reachable from a cycle, such as object y in Figure 2-16. Trial deletion is not fruitful unless the object is directly on the cycle. This further worsens the odds of selecting the correct seed. A bad selection results in wasted computation. Another drawback is that each object needs to have an extra field beside the one used to hold the actual reference count.

2.5.3 Complementary Tracing

The idea here is to invoke a global tracing periodically to collect circular garbage [Ali84, Juu90].

The drawbacks of this scheme are the same as those of global tracing itself (Section 2.2.2), except with reduced severity because tracing as a complementary scheme is run infrequently, and its responsibility is limited to collecting circular garbage. In particular, all the nodes must be up together for tracing to complete. This violates the desirable property that the collection of a cycle not depend on nodes other than where it passes through.

The next few subsections describe the use of more sophisticated techniques that improve upon the fault tolerance of plain tracing.

2.5.4 Tracing with Timestamps

In conventional global tracing, nodes have to synchronize their mark and sweep phases: no garbage can be swept until marking is complete at all nodes. [Hug85] uses propagation of timestamps instead of marks to do away with the synchronization. The usual local collection, which traces the local space starting from the primary roots and the inlist, also plays the role of propagating the timestamps on behalf of global tracing. As before, nodes can perform local collection independently. Each local collection contributes toward global tracing, pushing it a bit further ahead every time. As a result, global tracing turns into a smooth, albeit slow, ongoing process executed incrementally through local collections. However, it still remains the case that a node that is inaccessible or unwilling to do local collection will ultimately hold up the entire global tracing.

The algorithm as described in [Hug85] uses reference flagging augmented with timestamping. That is, each inlist entry contains a timestamp, but no other information such as the reference count. Consequently, the scheme relies on global tracing to collect all distributed garbage, not just circular garbage (Section 2.4.1). This is particularly bad because the tracing technique employed here can take a lot of time to collect garbage. In fact, we hold that it is possible to use the same technique in conjunction with reference counting or reference listing instead, which will speed up the collection of non-circular distributed garbage.

The main idea of the technique is that the timestamp of a reachable object keeps advancing, while the timestamp of a garbage object eventually saturates. After a while, a certain global minimum rises above the timestamp of the garbage object, and then the object can be collected. We now describe the algorithm, which is somewhat involved, in greater detail.

A local collection propagates the timestamps associated with the roots to the reachable entries in the outlist. Each node has a clock that it uses to record the time when the local collection was initiated; call it the *GC-time*. The primary roots are all timestamped with the GC-time, while the inlist entries retain the timestamps last put into them. When an inlist entry is first created at the owner, it is marked with the current timestamp. The local collection is expected to mark an outlist entry with the largest timestamp of any root (primary or inlist) from which it is reachable. To achieve this, roots are selected for tracing in the decreasing order of their timestamps. Any unmarked outlist entry reached can be marked with the timestamp of the current root. But objects in general, which intervene between the roots and the outlist entries, need only a bit for marking, as is usual in tracing. A marked object need not be traversed again: selecting the roots in the decreasing order of their timestamps avoids the need to trace the object graph multiple times.

At the end of the local collection, the outlist stands ready with fresh timestamps. The outlist is divided according to the owner nodes, and the parts are dispatched to those nodes. This appears to be similar to the batched trim messages of Section 2.4.4, but their functionality is actually similar to that of marking messages (Section 2.2). When a node receives a marking message, it updates the timestamps of the corresponding inlist entries to the maximum of their existing value and that suggested by the outlist. When a node increases the timestamp of an inlist entry, it records the fact that it has not propagated the increased timestamp. To this end, each node maintains a timestamp, called *redo* in [Hug85], such that all timestamps less than or equal to that are guaranteed to have been propagated. Therefore, in increasing an entry's timestamp, the redo is set to the entry's old timestamp, if that is *lower* than its current value. When a node has processed the marking message, it sends back an ack to the sender. When the sender has received acks from all nodes it sent the new outlist to, it can bump its own redo to the GC-time (provided it did not receive marking messages from other nodes itself).

It can be shown that an inlist entry timestamped below the global minimum of the redo's of all nodes is garbage. However, it is tricky to find the global minimum of the redo's at any time because redo values keep bobbing up and down. (Note that while timestamps of inlist and outlist entries can only increase, the redo value of a node often decreases on the arrival of a marking message.) [Hug85] modifies a distributed termination detection algorithm [Ran83] to compute the global minimum, but other distributed snapshot algorithms could be used as well. [Ran83] presumes the use of global synchronized clocks and instantaneous messages. We believe that apart from the use of this algorithm, the technique in [Hug85] will work just as well *without* these two requirements. If the clocks on different nodes are out of synchronization, that will only affect the performance by further delaying the collection of garbage.

One way to view the above algorithm is that each local collection kicks off a new run of tracing by marking its primary roots with the current timestamp. Further, by propagating the timestamps in the inlist entries, it simultaneously propagates the tracing runs initiated at other nodes.

There are several problems with this scheme. Although it does not require the nodes to synchronize the marking or be all up together, it does require all nodes to cooperate. If a node that crashed does not recover, or is simply unwilling to do its part, the entire global tracing will eventually come to a halt. In terms of the algorithm, the global minimum redo will get stuck at that node's redo (which remains fixed), so newly created distributed garbage will never get collected. This is true even if the inaccessible node is not related to the distributed garbage in question. Another drawback is that the execution of the distributed algorithm to compute global minimum redo is costly and slows down the collection of garbage.

2.5.5 Centralized Server

This scheme makes use of a logically centralized service that tracks all inter-node references [LL92]. The implementation of the service may actually be distributed, and may use replication for high availability and reliability, but it appears as if it were run by one server [LLSG90].

Nodes communicate with the service to provide it with information of their outlists and translists (Section 2.4.2), typically, soon after they have performed a local collection. They also query it for the accessibility of their inlist entries. More precisely, a node queries whether an inlist entry unreachable from its primary roots might be reachable from primary roots at other nodes. If the service responds with a no, the node can delete the corresponding inlist entry.

Since nodes do their local collections asynchronously and inform the service accordingly, the service never has a snapshot of the entire object graph at any time. Instead, it must make do with a fuzzy view and make conservative decisions while still guaranteeing that the nodes will ultimately be able to collect all garbage. To this end, it uses a timestamping protocol involving loosely synchronized clocks at the nodes and a maximum lifetime for messages in transit. The violation of these bounds does not result in incorrectness, but may force certain messages to be dropped and degrade performance.

The service can detect non-circular distributed garbage relatively easily. An inlist entry is not reachable from other nodes if it is not in the outlist or translist of any node. The service requires more information to detect circular garbage. For example, each node could provide connectivity information describing exactly which outlist entries are reachable from each of the inlist entries. The service can use this information to build a dense graph of all inter-node references present in the system, and perform a local tracing to detect garbage. This scheme is not viable, however, because to provide the connectivity information, the nodes may have to traverse their local spaces multiple times. [LL92] therefore employs the technique used in [Hug85] instead.

The nodes do not have to communicate with each other for the purpose of GC. The communication with the service can be performed in the background. Further, having a special server to detect distribute garbage offloads work from other nodes. The drawback, however, is that the server, albeit replicated, can become a bottleneck in a large system. Also, the nodes have to transfer a fair amount of information to the server in order to have it detect all garbage.

2.5.6 Tracing in Groups

This scheme aims to collect unreachable distributed cycles without requiring the cooperation of nodes other than those containing the cycle. The main idea is to reuse the concept of separate areas at a higher level: any set of nodes can decide to form a *group*, and perform a group-wide tracing that collects all cycles within the group [LQP92]. But unlike areas, groups of nodes can be formed and dismantled dynamically, although it may be desirable to form some groups statically and never dismantle them.

As in the case of separate areas, the conservative assumption that needs to be made to collect a group independently of others is that all in-coming references from outside the group are reachable. Therefore, the roots for a group-wide tracing comprise the primary roots of the member nodes and the references from outside the group. We shall refer to them as the *group's roots* for brevity. However, it is tricky for a group to find out the references from outside. Note that an area is statically fixed, so it can keep track of the references from outside in the inlist, but when a group is formed it must figure this out afresh. In [LPQ92], the group derives the information from the reference counts in the inlists at each node. It uses [Chr84] to compute precisely how many times an inlist entry is referenced from outside the group; if this is non-zero, the entry is included in the group's roots. We claim that the same could be achieved more easily if reference *listing* were used (Section 2.4.4). Simply stated, all inlist entries except those that stand for other member nodes of the same group are included in the group's roots. Once the group roots are found, the group members mark the included inlist entries to distinguish them from the rest (Figure 2-17).

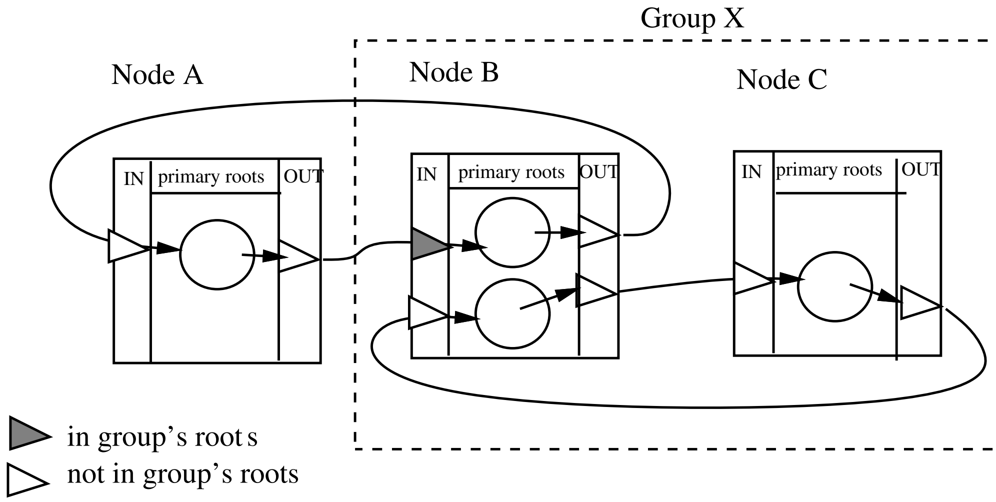


Figure 2-17: A group-wide tracing collects cycles lying within the group.

Just as [Hug85] used local tracing to propagate timestamps, [LPQ92] uses them to carry out the group-wide tracing. But unlike [Hug85], the mark and the sweep phases are separated. In the mark phase, the local collection is expected to propagate marks from the group roots to the outlist entries reachable from them. This is achieved by having the local tracing start from the group roots first, marking all outlist entries reached, and then continuing the remaining trace from other inlist entries.

The outlist entries that get marked are sent to their owner nodes, but only if the owner is a group member. On receipt of such a message, the owner node marks the corresponding inlist entries. If it does mark an entry that was not marked already, it records the fact that it has yet to propagate some marks by setting a *redo* flag. These additional marks will be propagated the next time it does a local collection itself. The owner then sends back an ack to the sender. When the sender has received acks from all nodes it sent outlists to, it can reset its own redo flag (unless it received outlists from other nodes in the meanwhile). Further, when a new inlist is created, the owner sets its redo flag. The marking terminates when all redo flags are reset. A distributed termination detection algorithm is employed to detect this condition. Once marking is over, all unmarked inlist entries can be collected.

This scheme is fault tolerant. If some nodes are down, or if there is a network partition, the set of nodes accessible to each other can still form a group and collect circular garbage lying within the group. Secondly, a small distributed cycle can be collected quickly by a small group instead of having to wait for a global tracing. Multiple group collections can be active at the same time. They may even overlap, though this puts more burden on local collections. If remote references exhibit spatial locality, *e.g.*, if there are many more remote references within the same local area networks than across them, groups can even be organized hierarchically.

Among the drawbacks of this scheme is that dynamic configuration of nodes into groups that succeed in collecting circular garbage is a non-trivial task. Further, it still involves a group-wide termination detection algorithm. This is in contrast with schemes like migration (Section 2.5.1), which makes incremental progress by communication between pairs of sites.

2.5.7 Summary

Among the techniques discussed in this section, object migration and trial deletion have the property that the collection of a distributed cycle involves only the nodes the cycle resides on. However, forced object migration may result in load-imbalance, while trial deletion is not suited to tracing-based local GC. Complementary tracing is effective provided all nodes cooperate. This may be acceptable in relatively small distributed systems. Using a highly available centralized server to coordinate the tracing improves fault tolerance by requiring only asynchronous message-passing between the server and the nodes. Tracing in groups limits the involvement in each run of tracing to the member nodes of the group, and collects distributed cycles lying within the group.

Chapter 3

An Overview of Thor

Thor is an object-oriented database system being developed by the Programming Methodology Group at the Laboratory for Computer Science at MIT. Thor can be used in a heterogenous distributed system and it allows programs written in different languages to share objects [Lis92, LDS92]. This chapter describes the system architecture of Thor, but the description is restricted to what is relevant to the design proposed in this thesis. The setting and the terminology introduced herein will be assumed in later chapters.

The nodes that run client programs are different from server nodes that store the database. Thor runs on both client and server nodes: the component that manages persistent objects at a server is called an *object repository* (*OR*), while the component that interacts with the client program is called a *front end* (*FE*). Thor's universe of objects is spread across multiple ORs, and multiple clients can access it concurrently through their own front ends (Figure 3-1). Thor is designed to be scalable: a database may span a large number of ORs separated by a wide-area network.

Objects in Thor are encapsulated and can be accessed by clients only through their operations. Thor provides transactions that allow clients to group operations so that objects are consistent in spite of concurrency and failures. The effects of committed transactions persist across crashes.

3.1 Object Repositories

An OR manages the objects stored at a Thor server. Each OR has a distinct *OR-id*. Objects at an OR may contain references to other objects at the same OR or objects at other ORs. A reference to a local object is in the form of an *oref*[1], which is a location-dependent name. (An oref is devised such that the object can be efficiently located while still allowing some flexibility in its exact placement.) A reference to a remote object, called an *xref*[2], is composed of the OR-id of the

[1]pronounced as "O-ref"
[2]pronounced as "X-ref"

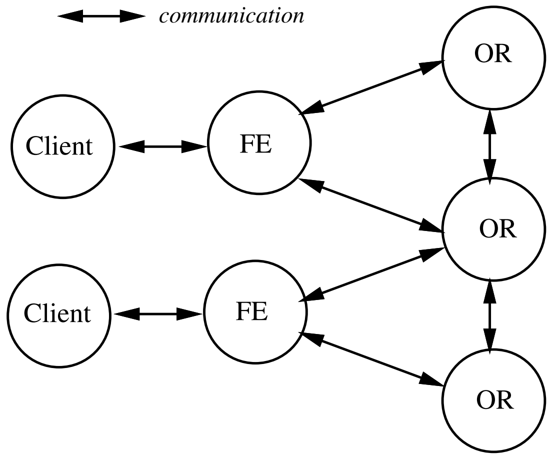


Figure 3-1: A configuration of clients, front ends, and object repositories in Thor.

remote OR and the oref of the object within it. Most references in the objects are expected to be local.

The slots for storing references in an object are only as wide as an oref, therefore an object cannot contain an xref directly. Instead, it contains an oref to a special local object, called a *surrogate*, whose sole purpose is to contain the xref of the remote object (Figure 3-2(A)). We shall often refer to non-surrogate objects as *actual* objects.

Each OR has a *root directory* object. The root directory contains references to other objects, presumably other directories.

The ORs use some kind of stable storage mechanism to store their objects. Our plan is to use primary copy replication [OL88]. Nonetheless, in the rest of this thesis we shall simply use the terminology commonly used for disks.

3.2 Front Ends

A front end is created for each client program. It fetches copies of the required objects from their ORs into its local cache, and runs operations on them. The objects at the ORs will be referred to as the *stable versions* to distinguish them from their *cached versions* at the front ends. The front end terminates together with the client program, or possibly earlier.

When the front end contacts an OR for the first time, it opens a *session* with it; during the session the front end and the OR maintain information about each other. The front end closes all sessions before it terminates.

To fetch an object from an OR, the front end asks for it by its oref. It can also ask for the root directory object at the OR without knowing its oref; in fact, that is how the front end begins to acquire other orefs.

When an OR receives a fetch request, it sends over the requested object plus some other objects that it suspects might be required by the front end in the near future. This mechanism is commonly known as *prefetching*, and we call the group of objects sent over a *block*. When the front end receives the block, it caches all objects therein. Later, the OR may *stream* more blocks in the background so that the front end has even more prefetched objects available.

As described earlier, objects at the OR contain references in the form of orefs. It is the front end's job to convert them into direct memory pointers to cached objects for better performance; the conversion is called *swizzling* [Mos90]. To do swizzling, the front end maintains a *swizzle table*, which maps xrefs to memory pointers for the cached objects. If an oref refers to an object that the front end has already cached, it replaces the oref with a memory pointer to the object. If the referenced object is *not* in the cache, the front end creates a *surrogate* containing the xref of that object, and the reference is replaced by a memory pointer to the surrogate (Figure 3-2(B)). To distinguish between the surrogates at ORs and those at front ends, we call them *or*-surrogates and *fe*-surrogates, but the prefix is dropped whenever the context is clear.[3]

The treatment of or-surrogates must be made clear here. Consider an or-surrogate stored at xref x_1 in OR_1, which contains the xref x_2 of an object in OR_2. When the or-surrogate is fetched by a front end, the front end makes two entries for it in the swizzle table: one at x_1, and another at x_2. Both point to the same object at the front end — the copy of the actual object, or an fe-surrogate if the object at x_2 has not been fetched. Thus an object may be entered in the xref table under a number of different xrefs.

If the front end runs into an fe-surrogate during the course of some operation, it *actualizes* the object by fetching it from the OR. The xref contained in the surrogate provides the OR-id and the oref. Then the surrogate is *filled* with the memory address of the actual object, so that all existing pointers to the surrogate now point indirectly to the actual object (Figure 3-2(C)). A background process at the front end keeps snapping such indirection by replacing pointers to filled surrogates with direct pointers to the actual objects. Subsequently, the local GC at the front end collects the surrogate.

The operations executed by the front end may modify the objects in the cache as well as create new ones. Objects at the front end for which a stable copy exists at some OR are said to be *persistent*; others are said to be *temporary*. When a transaction commits, copies of the persistent objects that were modified during the transaction are sent to the ORs they were fetched from. The temporary objects that are reachable from these objects are also sent along to the ORs, whereupon they become persistent.[4] They are said to be *newly persistent*. This is how the ORs get populated.

Before the front end sends a copy of an object to an OR, it replaces the memory pointers contained in the object with orefs. This conversion is known as *unswizzling*. To do unswizzling, the front

[3]"fe-surrogate" is pronounced as "FE-surrogate," and "or-surrogate" as "OR-surrogate."

[4]The mechanism is explained further in Section 3.4.

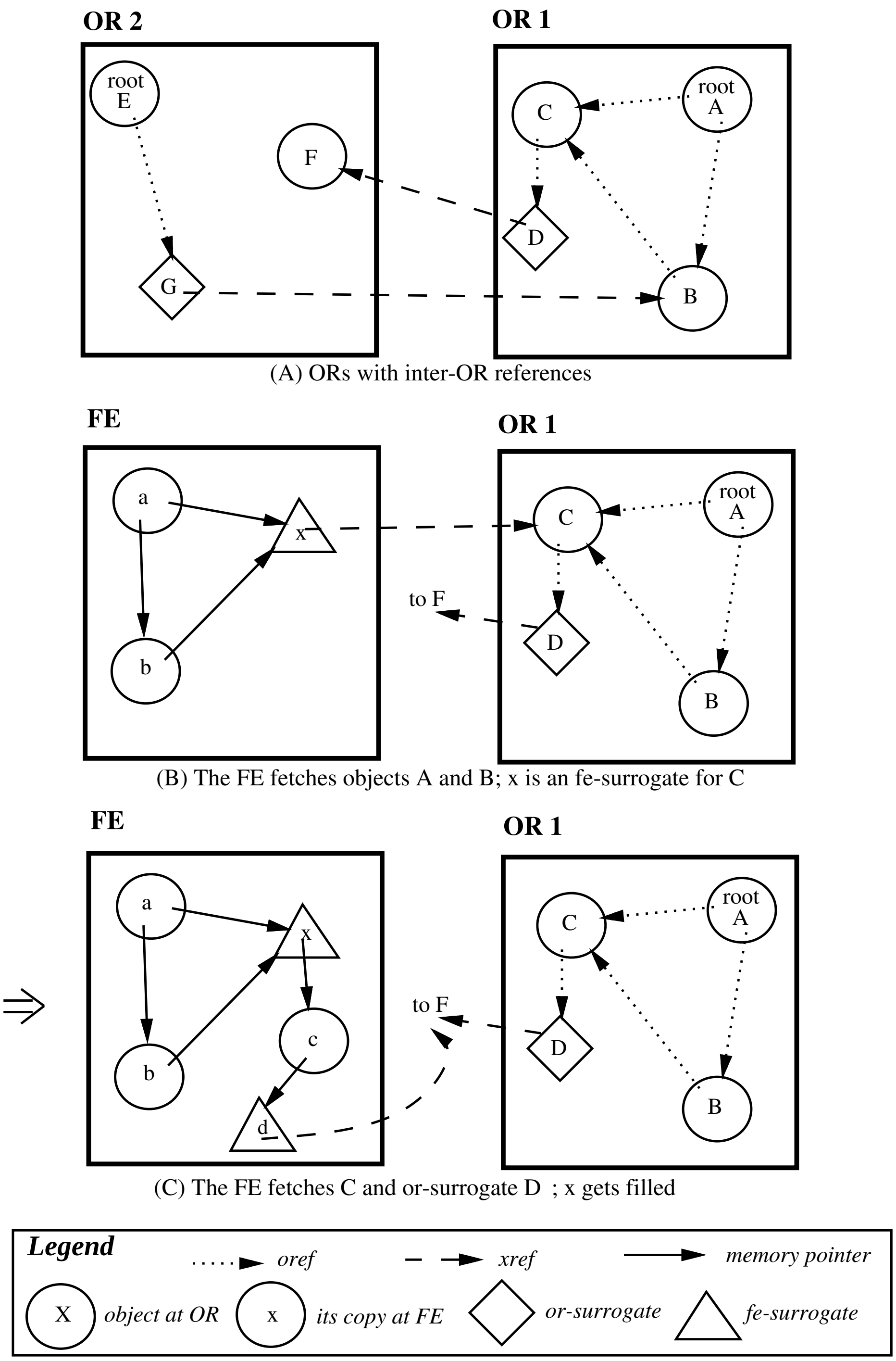


Figure 3-2: Inter-node references between ORs and Front Ends

end maintains an *unswizzle table* (or equivalent information in a different form) that maps a cached object to its xref. As noted earlier, an object at the front end may have multiple xrefs associated with it: one at the OR where the object actually resides, and one each for the or-surrogates fetched from other ORs that point to it. In order to avoid indirections in the unswizzled references, the unswizzle table provides the xref where the object actually resides (Figure 3-3).

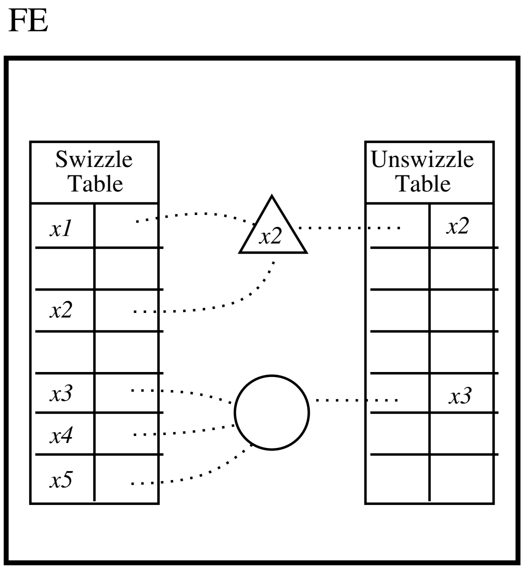


Figure 3-3: Swizzle table and unswizzle table

In unswizzling a memory pointer, if the xref of the containing object and that of the referenced object have the same OR-id, the pointer can be replaced by the oref part of the xref of the referenced object. Otherwise, the front end creates a new surrogate to be sent along with the containing object, and replaces the memory pointer with a reference to that surrogate. The new surrogate contains the xref of the referenced object. When the OR receives such a surrogate, it may create a new or-surrogate, or, if it already has an or-surrogate for that xref, it may use the existing one instead.

3.3 Clients

A client program, which may be written in any programming language, interacts with Thor via a front end. If the language is type-safe, the client and the front end may execute in the same address space. But programs written in unsafe languages could possibly corrupt the front end such that the errors reach the ORs. Since this is unacceptable, the two must run in separate address spaces and communicate through messages.

If the final result of an operation invoked by the client is a reference to an object, the front end turns it into a *handle* before returning it to the client (Figure 3-4). Handles are short lived: they are local to the lifespan of the front end. The front end maintains a *handle table* mapping handles to objects. The client refers to objects through handles. It can also ask for the root directory of an OR, and that is precisely how the client begins to get acquire handles.

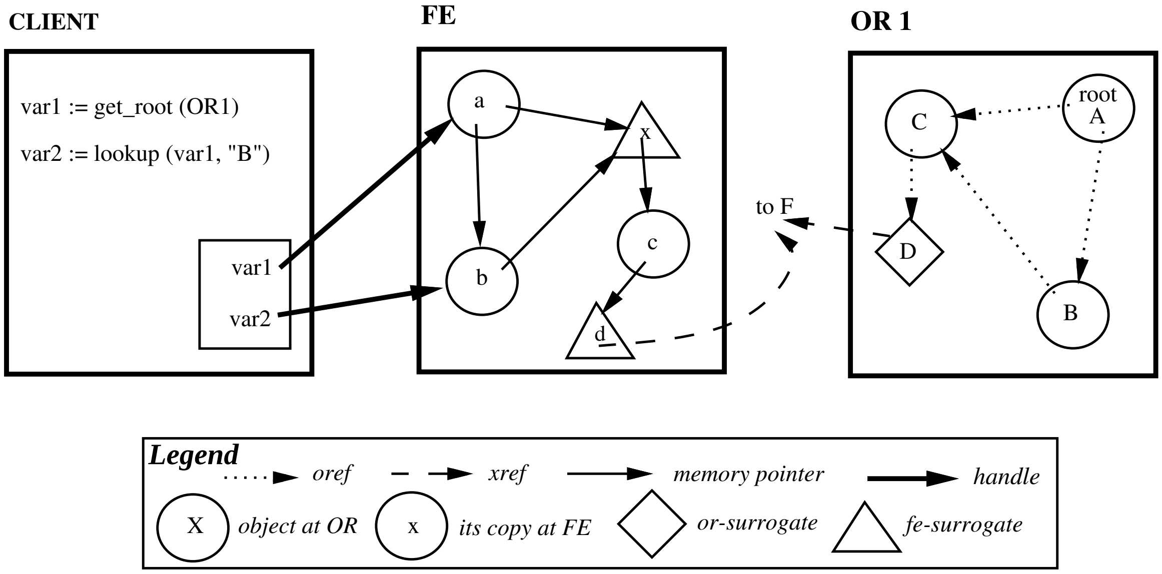


Figure 3-4: The client accesses objects at the front end through handles

3.4 Transaction Commit

The client may direct the front end to commit or abort the current transaction, which includes the series of operations since the last commit or abort. Transactions are serialized using optimistic concurrency control: the front ends might fetch inconsistent data, so the transactions have to be *validated* at commit time [KR81]. A transaction aborts if the validation fails.

The front end tracks the persistent objects read and modified during the current transaction. At commit time, it collects the (to-be) newly persistent objects by tracing for temporary objects reachable from the set of modified objects. For each such object, the front end indicates its preference for the OR where it should be placed.

The ORs whose objects were read or modified by the transaction constitute the *participants* of the transaction. The front end selects a participant as the *coordinator*, and sends it the necessary information about the transaction, including copies of newly persistent objects and modified objects. The coordinator coordinates the commit with other participants. The 2-phase commit protocol proceeds as follows [Gra78, Ady93]:

Phase 1 (or, the prepare phase)

initiate : The coordinator assigns the transaction a *tid* that is unique across Thor. All messages sent on behalf of the transaction are stamped with its tid.

The coordinator sends out *prepare* messages to the participants. The message to a participant contains the information required to validate and commit the transaction at that participant, including its share of newly persistent objects and modified objects.

prepare : When a participant receives the prepare message, it attempts to validate the transaction. If validation fails, it sends a negative ack to the coordinator. Otherwise, it forces a record of the information, called the *prepare record*, to stable storage and sends a positive ack. There might be other information that needs to be returned with the ack; *e.g.*, the orefs assigned to the newly persistent objects. We shall call it *phase 1 information*.

decide (commit/abort): The coordinator waits for acks from all participants. If it receives a negative ack, or if it times out while waiting (after having retried a few times), it decides to abort the transaction. Otherwise, it decides to commit the transaction, and saves a record of this decision and the collected phase 1 information on stable storage.

The coordinator can respond to the front end about the fate of the transaction as soon as it has decided. It also sends along the phase 1 information useful to the front end, including the xrefs of the newly persistent objects. The second phase is then executed in the background.

Phase 2 (or, the install phase)

install : The coordinator distributes the decision and the collective phase 1 information to the participants. When a commit decision is reported to a participant, it logs the decision and the phase 1 information on stable storage, and sends back a *done* message to the coordinator. At some later point, the modifications suggested in the prepare record are *installed* in the object store. If an abort decision is reported, the prepare record is invalidated.

done : When the coordinator receives a done message from all participants, it can truncate all records of the transaction.

We will talk of transactions as having *initiated*, *prepared* at a participant, *committed*, *aborted*, *installed* at a participant, or *done* if they have passed the corresponding stage.

Having summarized the architecture of Thor, we note that there are two distributed operations that lie in the critical path of operations invoked by clients (*i.e.*, they affect the latency visible to clients):

1. Fetching an object from an OR.
2. First phase of the 2-phase commit.

Therefore, a performance requirement on the distributed GC in Thor is that it minimize the delay it adds to these operations.

Chapter 4

FE-OR Garbage Collection

This chapter first presents an overview of our design for distributed GC in Thor, and then focuses on the GC protocol between a front end and an OR. The GC protocol to be followed among the ORs is described in the next chapter. Section 4.1 redefines the problem of distributed garbage collection in the specific context of Thor. Section 4.2 provides an overview of the proposed solution. The remaining sections provide detail on the FE-OR GC protocol. Section 4.3 describes the part of the protocol that ensures the safety of accessible objects, while Section 4.4 describes the part that guarantees progress in collecting garbage. Section 4.5 deals with providing safety despite OR crashes.

4.1 The Problem

The task of garbage collection in Thor is to reclaim as many objects at the ORs as possible such that no client ever finds anything amiss. It follows that the objects accessible from client programs need to be protected against collection. A client can refer to objects in two ways: either specify the root directory object at any OR, or use a handle given it earlier by the front end (Section 3.3). Therefore, the primary roots for garbage collection in Thor consist of the root directories of all ORs and the handle tables at the front ends running at the time.

The distributed GC is responsible for protecting remotely accessible objects and exposing inaccessible ones to local collection. The problem arises because of the roundabout manner in which the exchange of references might take place in Thor (Figure 1-3), which we recapitulate below:

1. The front end fetches objects from an OR referred to as the sender.
2. The front end reads data from these objects and copies it into objects fetched from another OR, referred to as the receiver.
3. When the time comes to commit the transaction, the front end sends necessary information, including the modified versions of receiver's objects, to an OR designated as the coordinator.

4. The coordinator executes the 2-phase commit protocol with the participants, which include the receiver, and notifies the front end of the result (Section 3.4).

Thus, when the sender sends the objects, it does not know which ORs, if any, will receive references extracted from those objects. The sender may or may not be among the participants, depending on whether any of its objects were read or modified during the transaction in which the receiver's objects were modified. For example, the front end might have read the sender's objects in an earlier transaction, stashed away some of the contained data in temporary objects, begun a new transaction, and copied the data from the temporary objects into the receiver's objects.

4.2 The Overall Plan

The ORs protect remotely accessible objects by tracking incoming remote references. Each OR maintains an *FE table* for every front end it has sent objects to; the FE table records a conservative estimate of references the front end holds to objects in that OR. Further, each OR maintains an *OR table* for every other OR, which records a conservative estimate of incoming references from that OR. If OR_1 has a surrogate pointing to an object in OR_2, the OR table for OR_1 at OR_2 has an entry for the referenced object. The use of FE tables and OR tables is a form of reference listing (see Section 2.4.4 for a detailed analysis), where all inlist entries from the same area have been grouped together into a table. As discussed in Section 2.4.4, the main advantage of using reference listing is that the removal of entries from FE tables and OR tables can be carried out by unreliable messages, called trim messages, sent in the background.

The roots for the local collection at an OR include its root directory object, its FE tables, and its OR tables. (To be complete, the roots also include the modified versions and new objects stored in the prepare records of transactions that are yet to be installed.) It is argued below that this protects all objects accessible from the primary roots. An object reachable from the local root directory is trivially protected against local collection. An object that is reachable from the handle table of an FE is protected by an FE table followed by a sequence of zero or more OR tables. An object that is reachable from a remote root directory via objects on intervening ORs is protected by the OR tables on those ORs.

The argument above is illustrated by the scenario in Figure 4-1. FE_1 has a session with OR_1, from where it has fetched the root directory A and an or-surrogate D. It is questionable how FE_1 got hold of D in the first place, since it is not reachable from any root directory. One possibility that can explain the situation is that the root directory A contained a reference to D when A was fetched by FE_1, and later, some other front end modified A such that the reference to D was deleted. Now consider object H, which is reachable from a handle at FE_1. The FE table for FE_1 at OR_1 protects the or-surrogate D, which contains an xref to H. Therefore, the OR table for OR_1 at OR_2 protects H. As another case, consider object J in OR_3, which is reachable from the root directory A in OR_1. The root directory A protects or-surrogate B, which contains an xref to F. Therefore, the

OR table for OR_1 at OR_2 protects F. F protects the or-surrogate G, which contains an xref to J. Therefore, the OR table for OR_2 at OR_3 protects J.

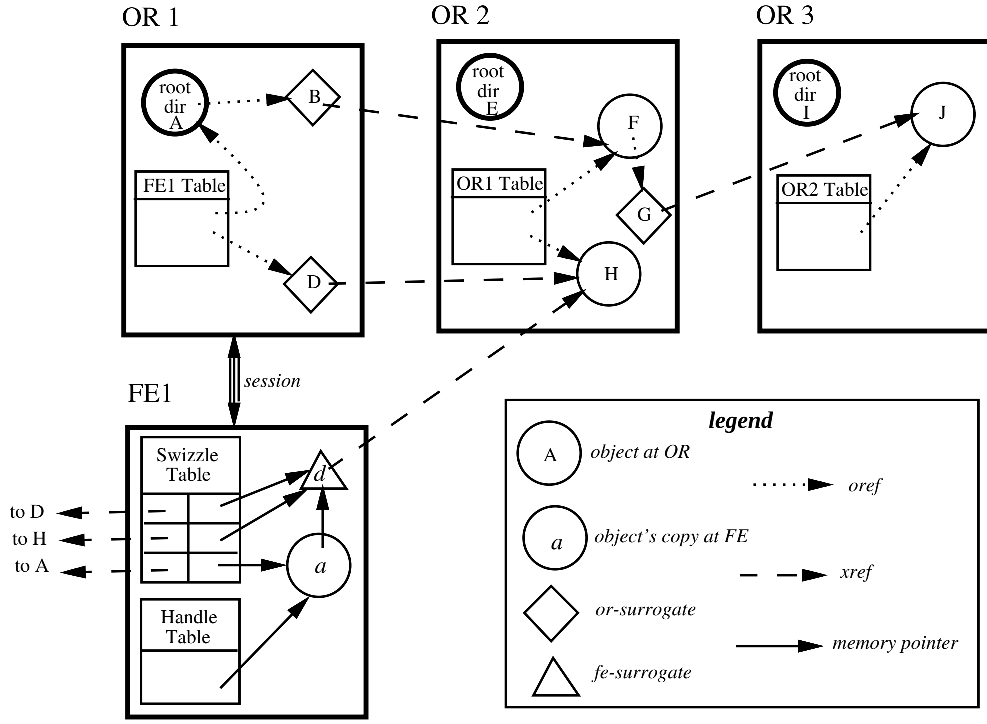


Figure 4-1: OR tables and FE tables protect objects accessible from primary roots.

It is the responsibility of distributed GC to insert and delete entries in OR tables and FE tables. The timely insertion of entries ensures the safety property that live objects will not be collected; the entries must therefore be made in synchronization with the operations that call for the insertion. On the other hand, deletion of entries ensures that garbage will be collected ultimately; therefore it can be delayed and executed in the background. The events that result in the insertions and deletions are briefly indicated below:

1. Entries are inserted in an FE table at an OR when the OR sends a block of objects to the front end in response to a fetch request.
2. Entries in an FE table are deleted when the FE closes its session with the OR, or when it sends a trim message, indicating which objects at the OR it continues to hold references to.
3. Entries are inserted in the OR table for OR_1 at OR_2 when OR_1, as a participant of a committing transaction, receives new xrefs to OR_2, which results in the creation of new or-surrogates.
4. Entries in the OR table are deleted when OR_1 sends a trim message to OR_2 indicating which objects at OR_2 it continues to hold references to.

The handle table is essentially an inlist maintained by the front end for the client. The front end makes entries in this table whenever it returns a new handle to the client. The client can help trim the handle table by sending trim messages to the front end, indicating the handles that are useful to it (or a delete message indicating the handles it is done with). The communication between a client, its front end, and two ORs is illustrated in Figure 4-2.

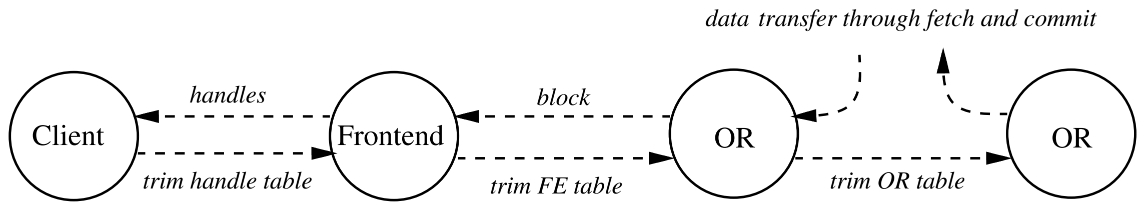


Figure 4-2: GC interaction between the client, the front end, and the ORs

We show below how OR tables and FE tables stitch together to protect the objects whose references are in transit between the sender and the receiver ORs. For simplicity, attention is focused on only one reference in transit. Further, to describe the most general scenario, the sender, the owner, and the receiver involved in the transfer are assumed to be different ORs. A series of events that tests the integrity of the scheme is described below, and is illustrated in Figure 4-3:

1. Before the sender sends the reference to the front end, there must exist an or-surrogate at the sender containing an xref to the object at the owner. The owner must therefore have an entry for the referenced object in its OR table for the sender.

2. After the reference is sent to the front end, the surrogate at the sender is protected by the FE table there. Following this, it does not matter if the preexisting protection for the surrogate is withdrawn. (Say, the surrogate was earlier reachable from the root directory, but becomes disconnected due to modifications made to the root directory.)

3. The front end copies the reference into an object fetched from the receiver OR. At commit time, it sends the modified object to the coordinator.

4. The coordinator, the receiver, and the owner communicate with each other, resulting in the creation of an entry for the referenced object in the owner's OR table for the receiver. The coordinator then informs the front end of the commit.

5. Subsequently, the front end may do a local collection that gets rid of the reference to the surrogate fetched from the sender. It then sends a trim message to the sender. The sender, noticing that the front end no longer references the surrogate, deletes the entry in the FE table that protected the surrogate. Alternatively, the front end could close its session with the sender, which results in the deletion of the entire FE table.

6. The surrogate at the sender can then be reclaimed when the sender next performs a local collection. After the local collection, the sender sends its outlist to the owner in a trim message.

7. The owner, noticing that the sender no longer references its object, will delete the corresponding entry in its OR table for the sender.

Note that the protection of the referenced object at the owner switched from the OR table for the sender to that for the receiver without any gap in between. The FE table ensured the protection of the or-surrogate at the sender, so that the entry in the OR table for the sender was maintained long enough.

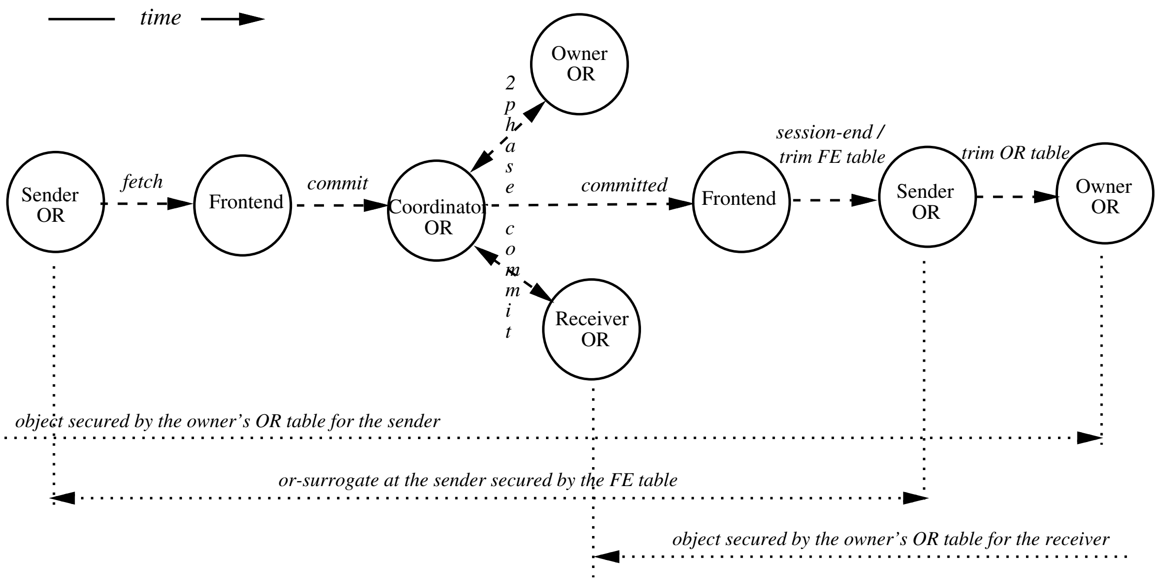


Figure 4-3: The overall plan: How objects with references in transit are protected.

The next four sections provide the complete design for insertion and deletion of entries in FE tables and OR tables.

4.3 Inserting Entries into FE Tables

When a front end opens a session with an OR, the OR creates an empty FE table for it. Subsequently, whenever the front end makes a fetch request for some object, the OR sends back a block of objects. What needs to be entered in the FE table?

At the bare minimum, the orefs of all objects in the block must be recorded in the FE table. So long as the stable versions of these objects are not modified, the references contained in the objects are also protected against local collection at the OR. The problem arises when a stable version *is* modified — by a concurrently executing front end or even the same front end — so that one of the original references contained in it is eliminated. If that was the only reference to its target object, that object might be reclaimed. But the front end may still have the reference cached. Therefore, it is not sufficient to record only the objects in the block: the references *contained* in those objects must be recorded too.

An or-surrogate included in the block is a special case because it contains an xref to an object at another OR. As with other objects in the block, the oref of the or-surrogate is recorded in the FE table. As long as the or-surrogate exists, the object referenced by it is protected by an entry in an OR table at the OR where it resides.[1]

Consider, again, the scenario in Figure 4-1. When the front end receives the or-surrogate D from OR_1, it acquires two references: the xref of the or-surrogate D, and the xref of the referenced object H at OR_2. It uses the xref of H to directly refer to that object, as when fetching the object or unswizzling a memory pointer to it. But it also retains the xref of D — until H is fetched (if ever) — so that the entry for D in the FE table at OR_1 is not removed[2], which in turn protects the referenced object H through the OR table at OR_2. This technique is similar to the strong-weak pointers technique described in Section 2.4.5. Here, the strong pointer is the xref of the or-surrogate D, while the weak pointer is the xref of the referenced object H. The technique protects the referenced object without sending an insert message to the OR where it resides.

In addition to the blocks sent to the front end, there is another source of entries in the FE table. When a new object created by the front end is installed at the OR (in the install phase of the commit protocol), its oref must be entered into the FE table because the front end has a reference to the object.

Although the insert protocol does not involve any extra messages, there is a different problem that needs to be solved: the recording of all references contained in the block increases the latency of the fetch operation and inflates the FE table. Blocks are relatively big because that allows more objects to be prefetched. The OR has to scan each object to check for references, and insert all references in the FE table. We refer to the naive scheme that requires the objects to be scanned before sending the block as *scan-ahead*. Two ways of optimizing the process are described below.

4.3.1 Scan-Behind

Before sending the block to the front end, the OR records only the objects composing the block, and it *locks* the stable versions of those objects against modification. The stable versions are scanned after the block has been sent. The locks guarantee that all contained references are indeed entered in the FE table before the containing object is modified.

[1]Note that or-surrogates are immutable. They may be modified to reflect migration of the referenced object, but such modification is benign.

[2]This remark will become clearer when the removal of FE table entries is discussed in Section 4.4.

This scheme removes the scanning of objects from the critical path of the fetch operation, but *increases* the total amount of work done because of the added complexity. Further, the scheme does nothing to reduce the size of the FE table.

4.3.2 Scan-on-Need

The approach is similar to scan-behind in that only the objects composing the block are recorded before they are sent to the front end. But the main idea here is not to scan an object *unless* it is going to be modified.

Each FE table needs to be divided into two:

- an *obj-table*, which records the references to the objects sent to the front end
- a *ref-table*, which records the references contained in the objects sent to the front end

References to or-surrogates sent to the front end can be entered in either obj-table or ref-table, because or-surrogates are immutable objects, and their contents need never be entered into the FE table.

Before a block is sent to the front end, its objects are entered in the obj-table. When the stable version of any object is about to be modified during the installation of a transaction (Section 3.4), the OR checks every FE obj-table for that object. If an entry for the object is present in any obj-table, the references contained in the object are included in the ref-table for the same front end.

In order to avoid re-scanning an object for the same FE table as it gets modified multiple times, a boolean $scanned$ is included with each entry in the obj-table, which indicates whether the object's references have been included in the ref-table. Initially, when an object is sent to the front end, $scanned$ in the obj-table entry is set to false. When an object about to be modified is discovered in an obj-table, $scanned$ is checked. If it is false, the object's references are included in the ref-table and $scanned$ is set to true. Nothing need be done if $scanned$ is already true. If the same object is sent to the front end again (as part of another block), the $scanned$ bit must be reset to false because the object might have been modified since the last fetch and may contain additional references.

Another optimization is possible. If the modified version of an object contains a superset of the references contained in the old version, the object need not be even searched for in obj-tables. This will hold, for instance, when the modifications are limited to the non-reference data part of the object, or when some extra data is appended to the object.

The virtue of this scheme is that it scans an object and adds its references into the FE table only when needed, namely, when the object is about to be modified and one of the contained references is about to be overwritten. Given that modifications are rarer than reads, we expect that most objects entered in obj-tables will not have to be scanned. The scheme also keeps the FE tables from exploding in size.

One hitch is that before modifying an object, it must be searched for in *all* FE tables at the OR. The situation worsens as the number of FE tables increases. But note that objects at the OR are modified when a transaction is being *installed*. Since this is a background process, delaying it is probably an acceptable overhead — especially in the view of the fact that it expedites the fetches, which lie in the critical path of client-invoked operations, and are more common.

Further, the search for the FE obj-tables to which an object belongs can be expedited if auxiliary data-structures are maintained to store this information. As an extreme, an additional table may be maintained that maps an object at the OR to the front ends it has been sent to. [3]

4.4 Removing Entries from FE Tables

An FE table is removed when the front end closes its session with the OR, which typically happens when the front end is about to terminate. For short sessions, it may be acceptable to let the FE table grow monotonically as more entries are added to it. For long-lived ones, however, the FE table may grow very large and need some pruning.

4.4.1 Redundant Entries

There are various sources of redundant entries in FE tables. First, if the OR composes the block with the objects physically clustered around the one asked for by the front end, some of those objects may not be accessible from references at the front end at all. (This will not happen if prefetching of objects is done according to their reachability from the requested object.)

Second, the front end will be doing its own garbage collection to make space in its local cache. The roots for this collection include the entries in the handle table. In addition, the front end must not collect any persistent object that was modified by the current transaction, regardless of whether it is reachable from the handle table. The reason is that when the transaction commits, the modified copy of the object needs to be sent to the coordinator. So the roots also include the modified object set for the current transaction (Section 3.4).

Besides deleting unreachable objects, the front end can create extra space by turning reachable objects that are *unlikely* to be accessed into fe-surrogates. This is the reverse of caching; we call it *shrinking* because it reduces actual objects to surrogates [Day93]. (Only persistent objects that have not been modified during the current transaction can be shrunk in this way.) When an object is turned into a surrogate, its contents are removed, which in turn may cause other objects to become inaccessible.

[3] There are other uses for such a table in Thor. One is the *invalidate-notification service*: when an object is modified as a result of a transaction at some front end, the OR notifies all other front ends that might have cached the object.

4.4.2 The Outlist

To each OR it has opened a session with, the front end sends a set of references it continues to hold to objects at that OR, so that the OR may trim its FE table for the front end. This is typically done soon after a local collection at the front end. Continuing with the terminology set up in Chapter 1, we refer to the set of references as the front end's outlist for the OR.

The front end gathers the outlist by making use of its swizzle table, which maps each xref known to the front end to a memory pointer for the corresponding object — actual or fe-surrogate (Section 3.2). To be useful for the purpose of computing the outlist, the functionality of the swizzle table is extended. First, the swizzle table is organized hierarchically: it maps an OR-id to an *oref table*, which in turn maps an oref to a memory pointer. In addition, the oref table can enumerate all pairs entered into it. A list of all orefs entered in the oref table for an OR can serve as the outlist for that OR.

As noted in Section 3.2, an object may be entered in the swizzle table at multiple xrefs. If the front end fetches an or-surrogate stored at xref x_1 in OR_1, which contains the xref x_2 pointing to an object in OR_2, it makes two entries in the swizzle table: one at x_1, and another at x_2. The entry for x_2 is in the oref table for OR_2. If the front end does not have a session open with OR_2, the oref table for that OR is not used as an outlist at all. Nonetheless, the object at x_2 is protected against GC because the or-surrogate in OR_1 is protected by the FE table there, which in turn ensures that the OR table for OR_1 at OR_2 protects the object. On the other hand, if the front end does have a session open with OR_2, the entry for x_2 is included in the outlist sent to it, even though the reference x_2 may have never been fetched from OR_2 and does not exist in the FE table for the front end at OR_2. This extra entry is redundant in that it protects an object that was already protected through another FE table and OR table. We expect that the occurrence of such entries will be small enough that the extra slots occupied in FE tables can be ignored.

Since the OR uses the scan-on-need scheme and maintains the FE table as an obj-table and a ref-table (Section 4.3.2), the outlist must be separated into an *obj-list* and a *ref-list*. This separation can be performed by checking the object paired with the oref in the oref table: if it is an fe-surrogate, the oref is inserted in the ref-list, otherwise it is inserted in the obj-list. The handling of the or-surrogates fetched from the OR needs to be validated. The scheme as described above inserts the reference to the or-surrogate into the ref-list if the actual object was never fetched, and into the obj-list if the object was; but this is acceptable because or-surrogates can be recorded in either obj-table or the ref-table at the OR (Section 4.3.2).

If the outlist computed for an OR is empty, the front end has an opportunity to close its session with that OR. This allows the OR to reclaim all resources it had maintained for that front end.

4.4.3 A Timestamp Protocol

When an OR receives a trim message from a front end, it replaces its FE table with the outlist in the message. In the context of the scan-on-need scheme, this means that the OR replaces its obj-table and ref-table with the obj-list and the ref-list contained in the trim message. Further, it can set all $scanned$ bits in the obj-table to true, because obj-list and ref-list together cover all references the front end continues to hold to objects in that OR.

Trim messages are sent in the background — asynchronously and using unreliable delivery. This, however, gives rise to the problem of delayed trim messages: what if the OR receives an old outlist that does not capture the results of the recent fetches made by the front end? Replacing its FE table with such an outlist will lead to erroneous deletion of some entries that were made following the recent fetches.

Any of a number of commonly-known, simple timestamping techniques can be applied to solve the problem. It is desirable to use a scheme that allows the OR to send a block spontaneously, that is, without a fetch request from the front end. This is useful for streaming extra blocks in the background after the requested object has been sent (Section 3.2). Such blocks can be sent using unreliable messages. A timestamping scheme suited for this purpose is given below. It is similar to the one used in [SDP92].

The OR timestamps each block sent out. It maintains the timestamp of the latest block sent to a front end as $T_{max[FE]}$, for each front end it has a session with. The timestamp required for the purpose is simply a monotonically increasing identifier. On the other side, the front end maintains the maximum timestamp of the blocks it has received from an OR as $T_{max[OR]}$, for each OR it has a session with. Just before the front end computes the outlist for an OR, it records the current value of $T_{max[OR]}$ for that OR as $T_{trim[OR]}$, and includes this timestamp in the trim message. This protocol requires the front end to drop blocks received later from the OR that are timestamped below $T_{trim[OR]}$. As a simplification, the front end could drop all out-of-order blocks received from any OR, that is, those timestamped below $T_{max[OR]}$; this may result in more blocks being dropped than if $T_{trim[OR]}$ was used as a threshold.

When the OR receives the trim message, it compares the timestamp included ($T_{trim[OR]}$) with $T_{max[FE]}$ it has maintained for that front end. It discards the message if $T_{trim[OR]} < t_{max[FE]}$ because this indicates that the outlist does not account for a block the OR sent recently.

The technique described above will be ineffective if block-sends are too frequent and trim messages are usually delayed, because the OR may have to discard all trim messages. The problem here is that when a trim message is late, it makes *no* contribution to trimming the FE table. A well-known fix for this is that each entry in the FE table be timestamped separately — with the timestamp of the last block that created or affirmed the entry. Now, even a late trim message can be partially effective: entries in the FE table that are missing from the outlist can be deleted provided the timestamp included in the trim message is as big as the timestamp associated with the entry. We chose *not* to use this scheme because it increases the size of the FE table and requires extra processing of trim messages. Further, we do not expect the problem to arise often enough to merit a fix that has an extra cost in the common case. Our solution is that, if an OR does not receive a valid trim message from a front end for a long time so that the FE table grows very big, the OR can send a query message to prompt the front end, and delay further block sends till a trim message is received.

4.5 Stability of FE Tables

In this section, we first argue why FE tables should be stable. However, maintaining an up-to-date stable copy of the FE table is a costly proposition. Therefore we introduce alternative mechanisms that substitute for stable storage operations.

Suppose that ORs do not keep any stable information for the front ends they have a session with. What might go wrong with a front end if an OR it has a session with crashes? When the OR recovers, it will have lost all knowledge of the front end, including its FE table. Subsequently, the objects that were protected only by the FE table might be reclaimed by the local GC at the OR. Further, if an or-surrogate that was sent to the front end is reclaimed, the actual object at another OR, to which the front end holds a reference, might be reclaimed too. With respect to Figure 4-1, the reclamation of or-surrogate D at OR_1 will withdraw the protection provided to object H at OR_2 and a subsequent local GC at OR_2 will collect H.

At first sight the situation seems manageable: whenever the front end attempts to use a reference to a deleted object (in either fetch or commit), the OR raises an exception, which forces the front end to abort the current transaction. However, aborting the transaction is not a cure for the situation because subsequent transactions may have to abort for the same reason. Conceivably, all objects containing a reference to the deleted object could be shrunk into fe-surrogates. This does not help, for instance, when the client holds a handle to the deleted object.

The problem is even more severe in the particular context of Thor because, as described below, the deletion of a referenced object may pass undetected at fetch or commit time. Usually, the OR will detect that the object has been deleted and raise an exception. However, in Thor, ORs are free to reuse the orefs of reclaimed objects. If another object has come to reside at the same oref the front end asked for, the front end will end up fetching the *wrong* object instead of finding it *deleted*, which in turn could lead to serious system errors.

We therefore conclude that having the ORs keep no stable information about front ends and allowing the front ends to run until an OR raises an exception is not acceptable. We discuss a number of alternatives below, starting with simple-minded schemes and evolving more elaborate ones in a bid to improve performance.

4.5.1 Stable FE Tables

A stable copy of the FE table is kept up-to-date at all times. Upon a fetch request, the OR updates the FE table and forces the update to stable storage before sending the block to the front end. This guarantees that if the OR crashes at any time, the FE table can be readily reconstructed on recovery.

The penalty paid is that the fetch operation is delayed by a write to the stable storage. Note that the stable update cannot be simply deferred and done in the background, because that introduces the possibility of the OR crashing after it sends the block but before the stable update could be made.

4.5.2 Stable Reconnect Information

Instead of keeping a stable copy of the FE table, the OR maintains only enough stable information that will allow it to reconnect with the front end when it recovers after a crash. The OR stores this information when a front end begins a session with it. The information may include the net address and the process id of the front end; we shall call it the *FE-id*.

When the OR recovers from a crash, it retrieves the stable set of FE-ids, and for each entry, checks if the front end is still running. If so, the front end is asked to send its outlist, which is then used to reconstruct the FE table at the OR. The OR must defer its local collection until, for each FE-id in the set, it has either confirmed that the front end is not running or the front end has sent it the outlist.

The problem arises when the OR is unable to access a front end, say, because of a network partition. If we are to uphold the guarantee that the front end will not run into a deleted object, the OR must not do a local collection without its FE table. But then, the local collection cannot be deferred for an unbounded span of time. One solution to this is to use *leases* [GC89] as discussed below.

4.5.3 Leased FE Tables

In this scheme, the OR guarantees to maintain the FE table for a limited time. We say that the "lease" for the FE table expires after that time. If the front end intends to continue using the references fetched from the OR, which will normally be the case, the lease must be renewed before it expires.

When a front end starts a session with an OR, the OR informs it of the time until when it will maintain the session. The OR computes this time by adding the lease period, T_{lease}, to the current time. Both the front end and the OR keep a record of the lease expiration time. Subsequently, a background thread at the front end sends a *ping* message to the OR to renew the lease. Ping messages are sent every T_{ping} seconds, where T_{ping} is much smaller than T_{lease}. They can often be piggybacked on the fetch and commit requests sent to the OR. When the OR receives a ping message, it extends the lease and sends back the new expiration time. On receiving the ack, the front end extends its own copy of the lease.

If the ping messages aren't acknowledged, the lease at the front end will gradually run out. A background thread at the front end checks the time that remains before the lease will expire. If this time falls below a certain interval, $T_{threshold}$, the thread triggers a shut down of the front end to avoid the possibility of a system error. This condition must be checked every T_{check} seconds, where T_{check} is smaller than $T_{threshold}$. On the other side, if the lease runs out at the OR, it can reclaim the resources devoted to the front end, including the FE table.

Now consider what happens if the OR crashes and loses its FE tables. When the OR recovers, it need wait for only time T_{lease} before performing a local GC; this ensures that the old leases granted to the front ends will expire before any object referred to from those front ends is reclaimed. If the OR receives a ping message from a front end during this period, it asks the front end to send the outlist so that it can reconstruct the FE table. If T_{lease} is chosen to be larger than the failover time, most of the sessions that were started before the crash will be able to continue in this way. This is particularly applicable in Thor because ORs are replicated so that failover is fast. Note that the OR need not even maintain the FE-ids on stable storage. The scheme preserves the following invariant:

> If a front end holds the lease for a session opened with an OR, then either the OR has an FE table for the front end, or the OR is crashed, or the OR has recovered but has not performed a local GC.

Besides maintaining sessions across OR crashes, leases solve another important problem, namely, partitions and undetectable front end crashes. Without the use of leases, the OR would be required to maintain the FE table for an indefinitely long time. While the stability of FE tables is a safety issue, the removal of useless FE tables is required to reclaim the space. The use of leases solves both of these problems without introducing stable storage operations.

There is a small probability that the lease at the front end expires while it is waiting for a commit request to execute. To see why this is an unsafe situation, consider again the scenario depicted in Figure 4-3. If the lease for the front end's session with the sender expires while the commit is pending, the sender might remove the FE table before the referenced object is protected by the OR table for the receiver. This opens up a window of time during which the referenced object is exposed to garbage collection. Even though the front end will shut down as required, the commit it issued cannot be revoked. The commit might result in the creation of a reference to a deleted object, or the "wrong one" if the oref of the object has been reused. Although this scenario is highly unlikely to occur, a simple fix can be used to safeguard against it. The front includes a *commit deadline* with each commit request, which is the minimum of the expiration times of all leases it holds. The commit must succeed by this time; otherwise the coordinator aborts the transaction.

For the use of leases to be safe, the lease at the front end must expire no later than it expires at the OR. Otherwise, the OR might assume the lease to have expired when the front end believes that it still holds the lease. Then the front end could try to fetch an object after the OR had discarded the

FE table and reclaimed that object. Our scheme relies on *loosely synchronized clocks* to guarantee correct behavior. Bounded differences in clock times can be handled by conservatively adjusting the lease period at the two ends. [Lis93] suggests a technique to implement leases using loosely synchronized clock *rates* instead of the absolute time maintained by the clocks. We could not employ that technique because the commit deadline included in commit requests needs to be an absolute time limit. We believe it is practical to rely on loosely synchronized clocks; the probability that the clocks will differ by greater than a generously chosen bound are remote.

The scheme we have just described keeps one lease for the whole of the FE table. Conceivably, leases could be kept for each entry in the FE table and the FE could renew some leases but not others. We do not consider this alternative because it is computationally expensive. Instead, the FE table at the OR is pruned using explicit trim messages sent by the front end (Section 4.4).

4.6 Summary

Each OR maintains an FE table for every front end it has sent objects to; the FE table records a conservative estimate of the references the front end holds to objects in that OR. Entries are made in the FE table when the OR sends copies of its objects to the front end.

The protocol is designed to minimize the latency of the fetch operation. Inter-OR references in or-surrogates sent to the front end do not generate insert messages to the owners because the or-surrogates are also protected by the FE table. Prefetching of large blocks, however, poses a problem: recording all references contained in the block would delay the fetch and increase the size of the FE table. The scan-on-need scheme records only the objects composing the block; the references contained in such an object are included only when the object is about to be modified.

Entries are removed from the FE table when the front end closes its session with the OR or sends a trim message containing an outlist of the references it continues to hold to objects in that OR. Trim messages are sent in the background using unreliable message delivery. A simple timestamp protocol is used to handle late trim messages.

The FE tables are maintained on a lease basis; that is, the OR guarantees to maintain the FE table only till the lease expires. Normally, the lease is renewed before it expires by ping messages sent in the background. In the event of a partition or a crash of the front end, the lease does not get renewed, and the OR can reclaim the FE table when the lease runs out. On the other side, if the front end is up, it shuts itself down when it detects that the lease is about to expire.

When the OR recovers from a crash, it waits for the lease period to pass before doing a local GC, so that there is no danger of reclaiming an object accessible from an active front end. The scheme allows most of the sessions started before the crash to re-establish in this duration. Further, the scheme avoids the need to maintain FE tables stably, thus eliminating a stable-storage write from the fetch operation.

Chapter 5

OR-OR Garbage Collection

An overview of our design for distributed GC in Thor was presented in the last chapter. It introduced the use of OR tables as inlists maintained by ORs to track the incoming references from other ORs. This chapter provides a detailed design of the GC protocol to be followed among the ORs. Section 5.1 describes the part of the protocol that ensures the safety of accessible objects by inserting entries into OR tables, while Section 5.2 describes the part that guarantees progress in collecting garbage by removing entries from OR tables. Section 5.3 deals with maintaining GC information on stable storage.

5.1 Inserting Entries into OR Tables

ORs receive new remote references in the prepare phase of the commit protocol. For simplicity, we will focus on a single remote reference contained in the modified version of an object that belongs to the receiver. To begin with, we consider the most general case, wherein the sender, the receiver, the owner, and the coordinator are all different ORs.

Before the commit, the referenced object is protected by the combination of an FE table at the sender and the OR table for the sender at the owner (Section 4.2). As shown in Figure 4-3, the front end might discard the reference soon after the commit is reported, thus withdrawing the protection provided by the FE table. Therefore, the referenced object must be provided additional protection to account for the creation of the new remote reference at the receiver.

Conceivably, the front end could refrain from discarding the reference until it had notified the owner in the background. Such a scheme employs a translist at the front end, as described in Section 2.4.2, to avoid synchronous insert messages to the owner. The scheme is unacceptable, however, because the front end is a *temporary* entity, and the protection provided by the FE table at the sender is temporary too. What if the front end is unable to contact the owner after the transaction has successfully committed? The front end could possibly contact the sender instead, so that the sender maintains a translist entry until the owner is notified. This seems plausible because the sender is a persistent entity, and it could wait indefinitely before the owner is notified. But, what if the front end is unable to contact the sender too? It would be an error if, after the front end and the FE table ceased to exist, the referenced object were reclaimed while the receiver holds a reference to it.

The upshot is that it is necessary to notify the owner (or the sender) as a part of the commit protocol itself, so that it is possible to abort the transaction if the owner (or the sender) is not accessible. It may be that the owner and the sender are not among the participants of the commit protocol (Section 4.1), therefore one of them has to be contacted solely for GC purposes. Now that extra messages must be sent anyway, it is preferable to communicate with the owner directly rather than use a translist entry at the sender. Further, unless the front end maintains extra information, the sender of a reference might not be known at commit time. If such information *is* maintained, it is possible to avoid extra insert messages when the sender is a participant while the owner is not; special cases like this are described in Section 5.1.4.

We describe two schemes for sending insert messages to the owner: a simple-minded one that adds a message round trip in *series* with the rest of the prepare phase, and another that adds the round trip in *parallel* with other prepare messages. The second scheme, however, requires a trickier timestamp protocol to take care of hazardous interplay of insert and trim messages (Section 5.2).

5.1.1 Serial Insert

The prepare phase proceeds as follows (Figure 5-1(A)):

1. The coordinator sends the receiver a prepare message containing modified versions of objects together with surrogates created by the front end for non-local references in those objects (Section 3.2).
2. If the receiver finds a non-local reference (xref) in any object, it checks whether it already has an or-surrogate for that xref. If so, it replaces the reference with the oref for the existing or-surrogate. Otherwise, it sends an insert message to the owner indicated in the xref.
3. When the owner receives the insert message, it makes an entry for the referenced object in its OR table for the receiver, and sends back an ack.
4. When the receiver gets the ack, it creates a new or-surrogate and replaces the non-local reference with its oref. Note that the or-surrogate is created only after the creation of a matching entry in the owner's OR table is confirmed. If the rest of the prepare work has also succeeded, the receiver sends a positive ack to the coordinator.
5. If, however, the receiver times out waiting for the ack from the owner, it sends a negative ack to the coordinator. The flip side is that if it cannot validate the transaction on other grounds, it need not bother about the insert message.

The additional message round trip adds to the commit latency observed by the client. Moreover, as discussed later in Section 5.3, updating the OR table involves stable storage writes, which further adds to the latency.

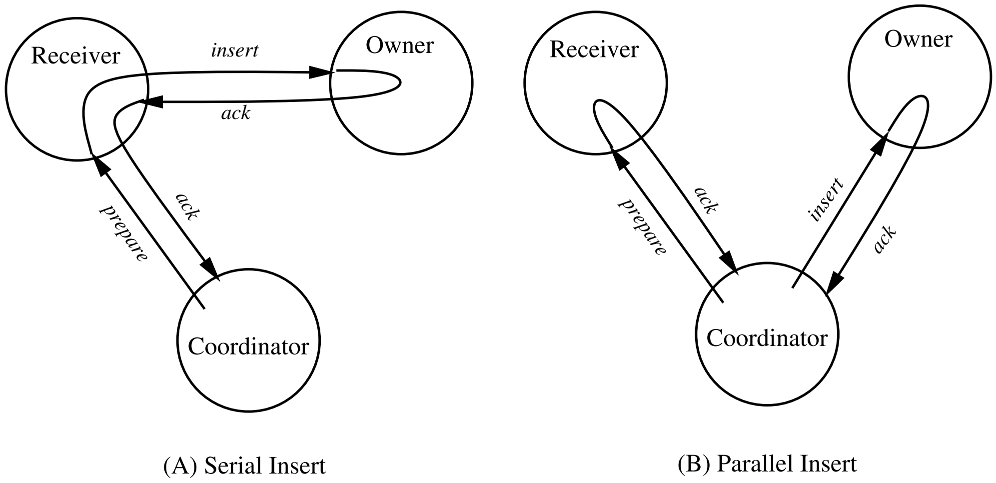


Figure 5-1: Two Insert Protocols

5.1.2 Parallel Insert

The prepare phase proceeds as follows (Figure 5-1(B)):

1. The the coordinator sends the insert messages to the owners in parallel with the prepare messages to the participants. The coordinator can figure out a conservative estimate of the references that need insert messages by searching for non-local references in the complete set of modified objects.

2. An insert message includes the identity of the receiver OR. When the owner receives the insert message, it makes an entry for the referenced object in its OR table for the receiver, and sends back an ack.

3. When the receiver finds a non-local reference in the prepare message, it checks whether it already has an or-surrogate for that xref. If so, it replaces the reference with the oref for the existing or-surrogate. Otherwise, the receiver creates a new or-surrogate, but marks it as *unconfirmed* because there is no certainty that a matching entry exists in the owner's OR table. The or-surrogate is *confirmed* when it becomes known that a matching entry was made in the OR table. The commit decision from the coordinator serves to notify the receiver that the or-surrogate is confirmed.

4. The coordinator waits for acks from all participants and owners, and commits the transaction only if all acks are received and those from the participants are positive.

This scheme is expected to reduce the latency of the commit protocol because the insert messages are sent in parallel with prepare messages. However, it has some drawbacks of its own. First, the coordinator might send insert messages for references that didn't require any. If the receiver were to send the insert messages itself, it would send one only if it did not already have an or-surrogate for the reference. If the creation of new inter-OR references is rare, a significant fraction of insert messages sent by the parallel scheme might be redundant. Section 5.1.4 suggests ways to minimize the extra messages. Second, it is simpler to design a remove protocol when the receiver itself sends the insert message. This issue is addressed in Section 5.2. Since the parallel scheme holds a promise of better performance than the serial scheme, this chapter will mainly focus on solutions to these problems.

5.1.3 Special Cases

The schemes given above describe the most general case, where the coordinator, the receiver, the sender, and the owner were all different ORs, and the sender and the owner were not among the set of participants. Special cases, like one where two of the roles are played by the same OR, result in some straightforward simplifications, which are listed below. We expect such cases to be common in practice due to spatial locality at ORs.

Special Case: $owner \in participants$

If the owner is a participant, the coordinator sends it one message containing both the prepare and the insert information; this saves an extra round trip of messages. This optimization is not possible in the serial insert scheme, where the receiver must send the owner a separate insert message.

Special Case: $owner \equiv coordinator$

If the owner is the coordinator itself, no insert message is necessary in either serial or parallel scheme, but the serial scheme needs to be slightly modified. The reason for this has to do with the remove protocol, and the slight modification required is described in Section 5.2.1.

Special Case: $receiver \equiv coordinator$

If the receiver is the coordinator itself, the serial and the parallel insert schemes are equivalent: an insert message is sent to the owner only if the receiver (coordinator) does not already have an or-surrogate for the purpose. However, in the parallel scheme, an insert message is still required if the or-surrogate is marked unconfirmed, because it is not certain that previous insert messages for such a surrogate succeeded in creating an entry in the OR table at the owner.

5.1.4 Tracking the Senders

In the model discussed so far, the sender of a reference is not known at commit time. If the front end tracks the senders of references and passes this information to the coordinator, many insert messages can be avoided. However, such a strategy adds some overhead at the front end (primarily space overhead). Therefore we discuss it as an optional extension to the scheme proposed so far. The design of the extension is such that, in the same Thor world, a front end may or may not employ it. The front ends that do are likely to experience somewhat shorter delays on distributed commits.

As described in Section 3.2, the front end maintains an unswizzle table that maps a cached object to the xref where it actually resides, although the object may have been entered in the swizzle table at multiple xrefs (Figure 3-3). The xref where the object resides indicates the owner of the object. Any other xref in the swizzle table that maps to the same object is for an or-surrogate that points to that object. The ORs where such or-surrogates were fetched from can be viewed as the senders in our model, because they sent to the front end a remote reference to an object at a different owner. For actual objects cached in the front end, the owner is technically a sender itself because it sent the object to the front end; however, for the sake of uniformity in discussions, we shall exclude the owner from the set of senders.

The front end could track the senders by storing in the unswizzle table the complete set of xrefs associated with each object. One of these xrefs is distinguished as the actual xref; the others denote the or-surrogates at the sender.

Figure 5-2 illustrates a situation where the modified version of an object x, which was fetched from R (the receiver), is to be sent to the coordinator. Object x contains a pointer to object y, which has to be unswizzled. Object y was fetched from O (the owner), and an or-surrogate pointing to y was fetched from S (a sender). The unswizzle table provides the following xrefs (an xref is denoted as an OR-oref pair):

1. the xref of the containing object at the receiver: $\langle R, a \rangle$
2. the xref of the referenced object at the owner: $\langle O, b \rangle$
3. the xrefs of the or-surrogates for the referenced object at the senders: $\{\langle S, c \rangle\}$

As before, if the receiver and the owner are the same, the front end replaces the memory pointer with the oref part of the xref of the referenced object. In the context of Figure 5-2, if $R \equiv O$, the pointer is unswizzled into the oref b. Otherwise, the front end checks if the following special cases are applicable.

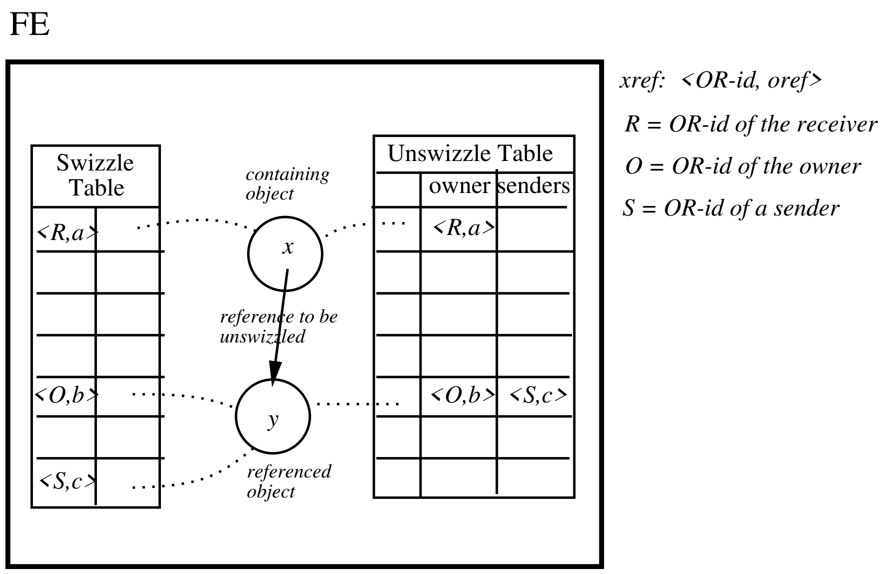


Figure 5-2: The front end remembers all xrefs associated with an object.

Special Case: $receiver \in senders$

This condition holds when the front end knows of an or-surrogate at the receiver that points to the referenced object; therefore the front end uses the xref of the or-surrogate in place of the actual xref of the object at the owner. The pointer to the referenced object is unswizzled with the oref part of the xref of that surrogate. In the context of Figure 5-2, if $R \equiv S$, the pointer is unswizzled into the oref c.

Unswizzling this way has a particularly benevolent effect because the receiver would have had to replace the reference with the oref of its surrogate anyway. Further, since the unswizzled reference is local to the receiver, no GC work need be done for it at commit time.

This technique helps the parallel insert scheme avoid many of the redundant insert messages that the serial insert scheme does not send. (In the serial insert scheme, the receiver never sends an insert message if it already has a suitable or-surrogate.) With this technique in place, the parallel insert scheme will send a redundant insert message only if the front end does not know of the or-surrogate at the receiver. In many common cases, the front end does know of such an or-surrogate, *e.g.*, when the field containing the reference to be unswizzled was never modified since the containing object was fetched from the receiver.

Special Case: $sender \in participants$

If one of the senders is a participant of the commit protocol but the owner is not, the insert message to the owner can be avoided by suitably preparing the sender instead. (The techniques described earlier are preferable if the owner itself is a participant, or if the receiver is a sender.) The idea here is to pass the responsibility of protecting the referenced object at the owner to the sender. The sender already has an or-surrogate for that object, which is protected by the FE table until commit time, which in turn protects the referenced object through the owner's OR table for the sender. Now, the sender has to extend this protection until it is sure that the owner has made an entry in its OR table for the receiver. To this end, the sender sends a background insert message to the owner on behalf of the receiver (Figure 5-3). The technique is detailed below.

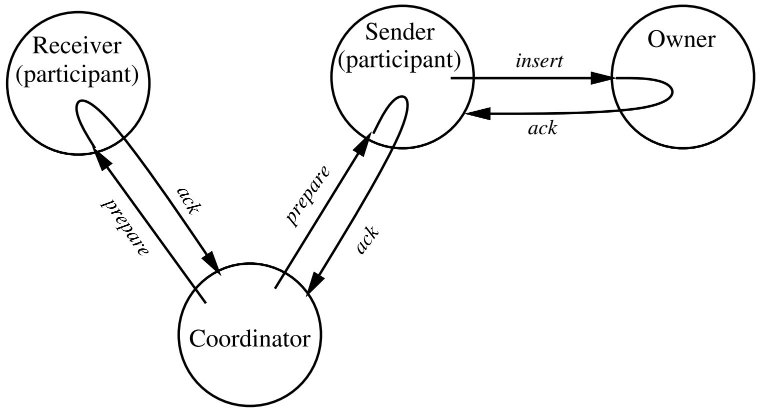


Figure 5-3: The sender sends an asynchronous insert message to the owner.

When the front end unswizzles a pointer, it first checks if the owner is same as the receiver, or if the receiver is a sender, or if the owner is a participant. If not, it checks if any sender is among the participants. If so, it creates a surrogate for the reference that contains both the actual xref of the referenced object at the owner, and the xref of the or-surrogate at the sender. In the context of Figure 5-2, the result of the unswizzling is shown in Figure 5-4. The containing object and the surrogate are sent to the coordinator.

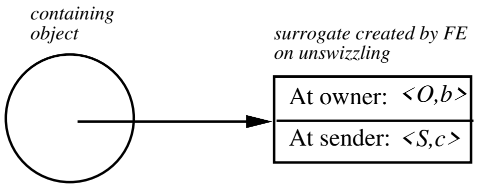


Figure 5-4: The surrogate sent to the coordinator contains xrefs at both the owner and the sender.

The prepare message from the coordinator to the sender contains a copy of the surrogate as well as the OR-id of the receiver. The sender records this information as an entry in its *translist* for the owner. The idea is similar to the one discussed in Section 2.4.2. The translist entry, which has a reference to the or-surrogate at the sender, is used to protect the or-surrogate against local collection. In order to expedite the prepare phase, the sender defers communication with the owner until after it has sent the ack for the prepare message. Later, it sends an asynchronous insert message to the owner, informing it of the new remote reference at the receiver. In fact, the sender can batch together asynchronous insert messages for entries pending in its translist for the owner into a single message. A further possibility is to merge asynchronous insert messages with trim messages: such a message contains the sender's translist and the outlist for the owner.

The owner processes the translist received from the sender as follows: for each translist entry, it makes an entry in its OR table for the indicated receiver. The owner sends the translist back in its ack to the sender. When the sender receives the ack, it can remove the entries in its translist that are included in the ack. (The sender may have made more entries in its translist since it last sent the translist to the owner, which should not be removed.)

An alternative way of protecting the or-surrogate at the sender would be to use the strong-weak pointer scheme instead of the translist, which is discussed in Section 2.4.5. It is not clear whether one is substantially better than the other.

This technique avoids the need to send a synchronous insert message at commit time. Only the parallel insert scheme can benefit from this. The same technique can also be used for another purpose: when the owner is not accessible at commit time, the coordinator can contact a sender instead, even if it is not a participant; this is discussed in the next subsection.

5.1.5 Unavailability of the Owner

In the general case, when an insert message is sent to the owner, the transaction has to be aborted if there is no ack from the owner in due time. On the other hand, conventional distributed systems are able to avoid dependence on the owner for the transfer of a mutator message by using GC protocols that do not involve a synchronous insert message to the owner. We argued earlier that the same is not possible in Thor because the sender does not know the future receiver(s) when it sends objects to a front end.

This problem arises in both the serial and the parallel insert schemes, but is slightly more severe for the parallel scheme because it sends some redundant insert messages that the serial scheme does not. Actually, when the insert message is redundant, it is possible to avoid the dependence on the owner as follows. The receiver, along with its ack for the prepare message, indicates which non-local references in the modified versions did not require an insert message (because the receiver already had confirmed or-surrogates for them). Then the coordinator does not have to wait for the acks of those insert messages.

The dependence on the owner can be further reduced if the front end tracks the senders of references, as described in Section 5.1.4, and passes on the information to the coordinator even when none of the senders is a participant. If the owner does not respond to an insert message, the coordinator tries to contact an OR that has a confirmed or-surrogate for that reference. It is possible, though not very likely, that the coordinator itself has such an or-surrogate. Otherwise, it tries to contact the sender indicated in the surrogate sent by the front end. The sender (or the coordinator itself if it has an or-surrogate to the referenced object) makes an entry in its translist to protect its or-surrogate until the owner is notified. This allows the commit protocol to proceed while the owner is unavailable.

5.2 Removing Entries from OR Tables

An OR trims its OR table for another OR when it receives a trim message containing a fresh outlist from that OR. An OR can also send a query message to another OR to prompt it to send its outlist.

Each OR keeps an outlist for every OR to which it contains references. The outlist for OR_1 at OR_2 can be viewed as an abstraction over the set of all or-surrogates at OR_2 containing xrefs to objects in OR_1. Entries in the outlists correspond with the or-surrogates: they are created and deleted together with or-surrogates. As discussed in Section 5.1, or-surrogates are created during the prepare phase of the commit protocol; they are deleted by the local collector. The roots for the local collection are the root directory object at the OR, and its FE tables and OR tables, and the modified versions and new objects stored in the prepare records of transactions that are yet to be installed. If translists are used, as suggested in Section 5.1.4, they too are used to protect or-surrogates against collection.

Trim messages are sent asynchronously, typically, soon after the OR has completed a local collection. A trim message that arrives late could erroneously undo the effect of insert messages sent later. In order to study the interplay of insert messages and trim messages, we use the same setup as used for the insert protocol: insert messages are sent on behalf of the receiver to the owner, resulting in new entries in the owner's OR table for the receiver. Therefore, for the remove protocol, we consider the trim messages that the receiver sends to the owner, which result in removal of entries from that OR table.

The serial insert scheme requires a simple remove protocol. The parallel insert scheme requires a more elaborate protocol, but keeping in mind the performance benefits of this scheme, we focus on the design of such a protocol, and argue that it has sufficient safety and liveness properties.

5.2.1 The Serial Scheme

Under the serial scheme, the receiver itself sends the insert and the trim messages to the owner. Therefore it is simple to keep late trim messages from getting re-ordered behind insert messages sent later. One way is to have the receiver timestamp each insert or trim message. The timestamp is simply a monotonically increasing identifier. The owner discards trim messages timestamped earlier than the latest insert message it has received from the receiver. When it receives a valid trim message, it replaces its OR table for the receiver with the outlist included in the message.

Consider the insert protocol again: when the coordinator is the owner itself, the receiver need not send a separate insert message (Section 5.1.3). Instead, the receiver simply includes its timestamp in the ack sent to the coordinator (owner) in response to the prepare message. The coordinator then makes new entries in its OR table for the receiver and records the timestamp sent by the receiver as that of the latest insert message.

This protocol is safe because it maintains the following invariant:

> For every or-surrogate at an OR, there exists a matching entry in the owner's OR table for that OR.

The insert protocol creates an or-surrogate only after the owner confirms the creation of an OR table entry. The OR table entry is deleted only after the or-surrogate is deleted and a trim message is sent that does not contain a corresponding outlist entry. The timestamp protocol ensures the orderly delivery of insert and trim messages.

5.2.2 The Parallel Scheme

The situation here is complicated by the fact that the insert messages are sent by the coordinator to the owner (on behalf of the receiver), while the trim messages are sent by the receiver itself. Therefore it is tricky to synchronize the insert and trim messages. Figure 5-5 illustrates the possible race condition described below:

1. The receiver sends its outlist for the owner in a trim message (Figure 5-5(A)).
2. The coordinator sends a prepare message to the receiver that results in the creation of a new (unconfirmed) or-surrogate pointing to an object, say x, in the owner. Also, the coordinator sends the owner an insert message on behalf of the receiver. The owner makes an entry for x in its OR table for the receiver (Figure 5-5(B)).
3. The old trim message sent by the receiver reaches the owner. Since the contained outlist does not have an entry for x, the owner deletes x from its OR table for the receiver (Figure 5-5(C)).

The aim is to design a protocol such that a trim message does not result in deletion of OR table entries that were created on behalf of a transaction whose effects are not captured by the outlist contained in the trim message. A timestamp protocol like the one designed for the serial scheme cannot be applied here because the insert and trim messages are sent by two different nodes without any prior communication between those two nodes.

To resolve the problem, we exploit two attributes of the distributed commit protocol in Thor [Ady93]:

1. Each transaction is assigned a *tid* by its coordinator that is unique across the Thor world. The tid consists of a timestamp generated locally by the clock at the coordinator's node, which is

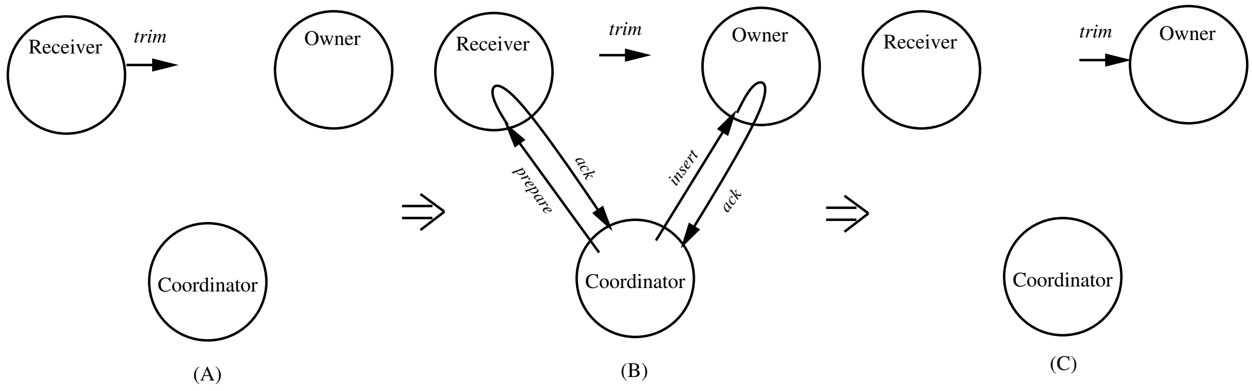


Figure 5-5: Race condition between insert and trim messages in the parallel insert scheme

augmented with the OR-id of the coordinator for distinctness. All messages sent on behalf of the transaction include its tid. The tids are used to serialize the distributed transactions.

2. A participant OR accepts to prepare a transaction only if its tid is above a certain threshold, called the *prepare threshold*, maintained by that OR. Otherwise, the participant asks the coordinator to retry the commit protocol with a larger timestamp.

The clocks used to generate the timestamps at different nodes need not be synchronized, but they should be loosely synchronized for good performance, for instance, to minimize the retries of the commit protocol.

The timestamp protocol given below makes use of the total ordering of the tids to synchronize insert and trim messages sent by different ORs:

1. The coordinator includes the transaction's tid in all prepare and insert messages.

2. Each entry in the OR tables includes the largest tid of any transaction that created it or *reaffirmed* it. We say an entry is reaffirmed when it would have been created if it did not already exist.

   When the owner gets an insert message, it checks for the indicated reference in its OR table for the receiver. If an entry does not exist, it creates one with the timestamp included in the message. Otherwise, it simply updates the timestamp of the existing entry to the larger of the existing timestamp and the one in the message.

3. When the receiver sends a trim message, it timestamps the message with the current prepare threshold, or more precisely, with the value of the prepare threshold just before it gathered the outlist. The outlist accounts for both confirmed and unconfirmed surrogates at the receiver.

4. When the owner receives a trim message, instead of replacing its old OR table for the receiver with the outlist contained in the message, it scans the OR table and removes an entry if:

   (a) the reference does not exist in the outlist, *and*

   (b) the timestamp of the trim message is at least as large as that of the entry.

Admittedly, this protocol requires more time to process the trim message, and also more space to keep the timestamps with each OR table entry. In Section 5.2.3, we discuss why this protocol is safe, and in Section 5.2.4, we discuss why it is effective in collecting garbage. Before that, we provide the protocol to be employed if a translist at the sender is used to avoid synchronous insert messages to the owner (Section 5.1.4):

1. Each entry in the translist is timestamped with the largest tid of any transaction that created it or reaffirmed it.

2. When the owner receives a translist from the sender, it treats each translist entry as it would a similar insert message directly from the coordinator; that is, it either creates a new entry in its OR table with the timestamp included in the translist entry, or merely updates the timestamp of the entry if it exists.

3. As before, the ack from the owner contains the translist it just processed. When the sender receives the ack, it scans the translist for the owner and removes an entry if:

   (a) the entry also exists in the ack, *and*

   (b) the timestamp of the entry in the ack is as large as that of the entry in the sender's translist.

5.2.3 The Parallel Scheme: Safety

In this section, we show that the protocol for the parallel scheme is safe. Ignoring the use of translists for the time being, the safety requirement is: if a transaction creates a remote reference at an OR, an entry in the owner's OR table for that OR must protect the referenced object from the point of commit until whenever the OR holds the reference.

The parallel scheme maintains two invariants which together guarantee the safety requirement:

1. A transaction commits only after the or-surrogates it created (or made use of, if already existing) have been confirmed.

2. For every or-surrogate that was confirmed by a committed transaction, there exists a matching entry in the owner's OR table (for as long as the or-surrogate exists).

Invariant 1 is maintained because the coordinator sends out insert messages for all references that the receivers might not have confirmed surrogates for, and commits the transaction only after receiving acks from the owners.

Invariant 2 requires that trim messages not remove an OR table entry for which there exists an or-surrogate at the receiver that was confirmed by a committed transaction. An informal proof that the given protocol has this property is provided at the end of this subsection; before that, we provide some insights on the interaction of trim messages with other events.

Given that a trim message is timestamped with the prepare threshold at the receiver, it accounts for the or-surrogates created by all transactions that will ever be prepared at the receiver with a lower or equal timestamp. This claim is based on the subtle assumption that the unconfirmed or-surrogates created by a transaction are visible in the outlist as soon as the transaction is prepared, and that if a transaction is in the process of being prepared, the prepare threshold is not raised above the transaction's tid until after it is prepared.

Now consider an entry for the object o in the owner's OR table for the receiver, which is timestamped with the tid of transaction X. A trim message sent by the receiver can result in the deletion of this entry only if it does not contain a reference to o, and if its timestamp is as large as the tid of X. This condition can arise from two different kinds of situations, which are discussed below.

The first kind of situation arises when the trim message was sent *after* transaction X was prepared at the receiver. It represents the normal course of events leading to the removal of a useless entry in an OR table. It develops as follows (Figure 5-6):

1. During the prepare phase of X, an or-surrogate pointing to object o was created at the receiver, and an OR table entry for o was created at the owner (Figure 5-6(A)).

2. Since the trim message from the receiver did not contain a reference to o, the or-surrogate must have been collected by the local GC before the trim message was sent (Figure 5-6(B)). (For instance, this could happen if another transaction, Y, which was serialized after X, made the the or-surrogate unreachable.)

3. Later, when the trim message was sent, the prepare threshold at the receiver happened to be higher than the tid of X, so it was effective in removing the OR table entry at the owner (Figure 5-6(C)).

The second kind of situation arises when the trim message was sent *before* transaction X was prepared at the receiver. In this case, the entry removed from the OR table was created by a transaction the receiver had not accounted for when it sent the trim message. Nonetheless, the removal is safe because the distributed commit protocol will not allow the transaction to prepare at the receiver. The situation develops as follows (Figure 5-7):

1. The insert message sent on behalf of transaction X resulted in the creation of an OR table entry for o (Figure 5-7(B)).

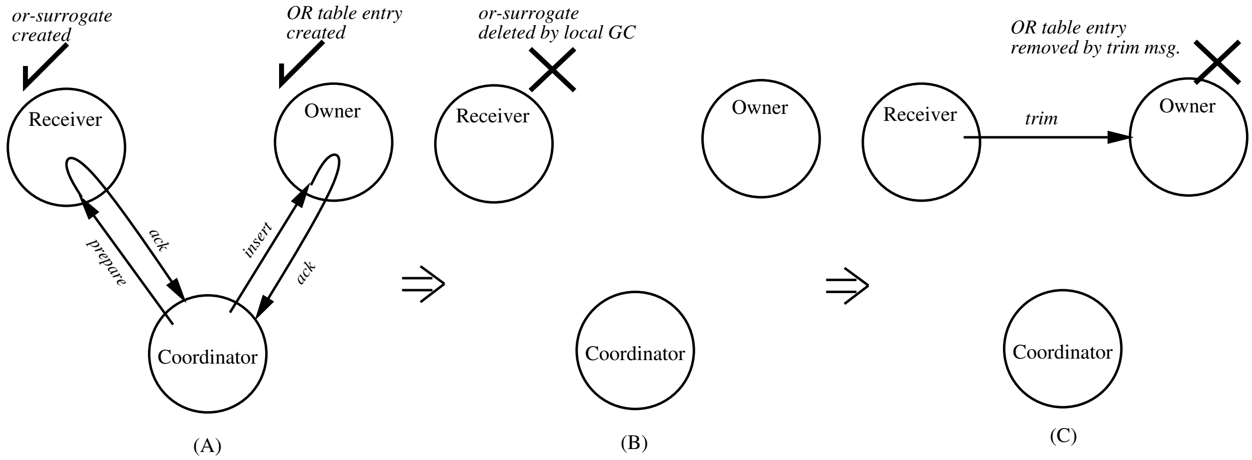


Figure 5-6: The trim message deletes a useless OR table entry.

2. The receiver sent a trim message while its prepare threshold was greater than the tid of transaction X. The trim message resulted in the removal of the OR table entry for o.

3. The prepare message sent on behalf of transaction X reached the receiver. Since the prepare threshold of the receiver is above the tid of X, the receiver refused to prepare X and sent back a retry message (Figure 5-7(C)).

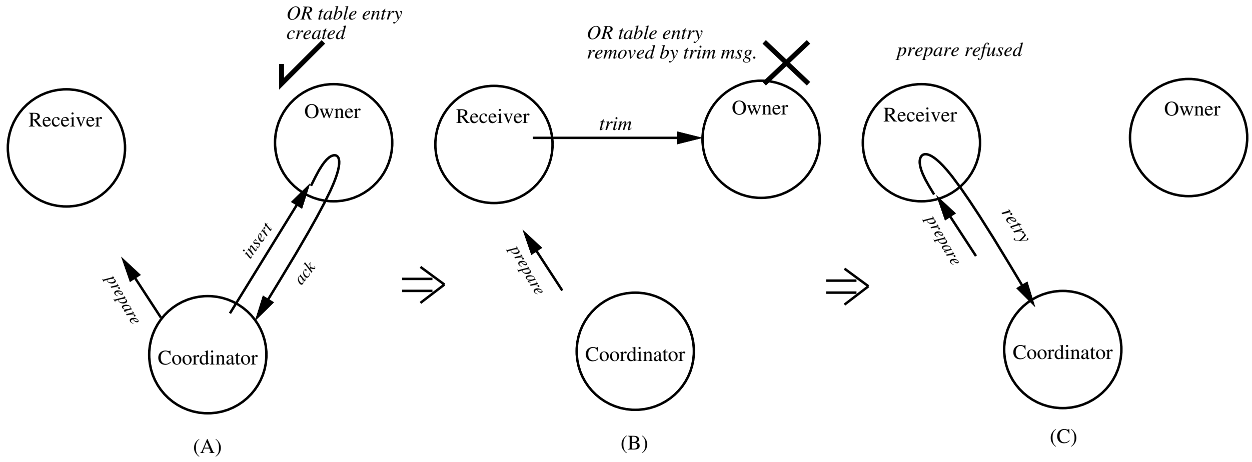


Figure 5-7: The trim message deletes an entry it did not account for.

However, it is possible for an *unconfirmed* or-surrogate to exist at the receiver while there is no matching entry in the OR table at the owner; this is illustrated in Figure 5-8. Suppose the prepare message results in the creation of an or-surrogate at the receiver, but the insert message fails because, say, the owner is not accessible. The coordinator times out waiting for a reply from the owner and aborts the transaction. As a result, the receiver is left with an or-surrogate that has no matching OR table entry. It follows that it is possible for a trim message to contain an entry for which there is no matching entry in the OR table at the owner. Note that this is not problematic because the entry is ignored when the trim message is processed at the owner. Unconfirmed or-surrogates are ultimately reclaimed by the local collection.

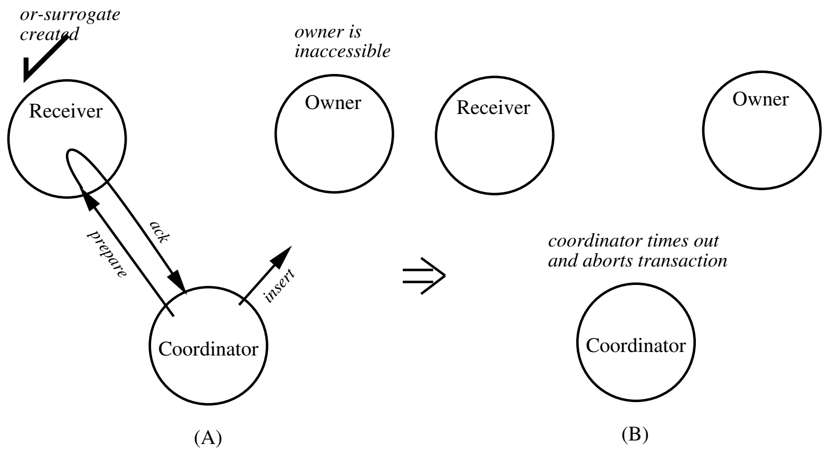


Figure 5-8: An unconfirmed or-surrogate may not have a matching OR table entry.

Now we give a short, informal proof that the given protocol preserves Invariant 2, namely:

> For every or-surrogate that has been confirmed by a committed transaction, there exists a matching entry in the owner's OR table.

Proof (by contradiction):

Consider an or-surrogate at the receiver that was confirmed on behalf of transaction X. This means that an insert message timestamped with the tid of X was delivered to the owner, with the result that an entry was created or reaffirmed in the owner's OR table for the receiver. The protocol ensures that the timestamp of this entry is at least as large as the tid of X.

For the sake of contradiction, assume a trim message resulted in the removal of this OR table entry. For this to be possible, the timestamp of the trim message must be as large as the timestamp of the entry, and hence, as large as the tid of X. But the trim message was timestamped with the prepare threshold at the receiver; therefore the prepare threshold when the outlist was collected was as large as the tid of X. Since we know that X prepared successfully at the receiver, X must have prepared strictly before the outlist was collected. After X is prepared, the or-surrogate in question must be visible in the outlist. Thus, the outlist sent to the owner must have an entry for the referenced object. This contradicts the assumption that the trim message was successful in removing the OR table entry for the referenced object. □

5.2.4 The Parallel Scheme: Liveness

The liveness requirement is that if the OR table for OR_1 at OR_2 has an entry for which OR_1 has no matching or-surrogate, the entry should be removed ultimately. It is assumed that ORs do a local collection and send trim messages to other ORs every so often.

The protocol, as described so far, does not quite have the liveness property. If the prepare threshold of the receiver were to remain constant, the trim messages it sends would never remove any entry with a greater timestamp. A simple fix to this problem is to keep the prepare threshold advancing, say, by imposing a constraint that it never lag behind the clock time by more than some fixed interval. Advancing the prepare threshold in this manner would imply that a prepare request arriving too late at a participant might be refused by the participant so that the commit protocol has to be retried. We believe that such a constraint is practically reasonable.

The distributed commit protocol in Thor sets the prepare threshold at a level that makes the GC protocol particularly efficient. The prepare threshold at an OR is set to the *install watermark*, that is, the highest tid of any transaction that has been installed at that OR [Ady93][1]. We show below that this leads to an efficient removal of OR table entries.

Consider an or-surrogate at the receiver and a matching entry in the owner's OR table that is stamped with the tid of transaction X. As soon as transaction X is installed at the receiver, the prepare threshold there is sure to be at least as large as its tid. If the or-surrogate ever becomes unreachable, it will be collected by the next local collection, and the subsequent trim message to the owner will not list the corresponding remote reference. Since the trim message is timestamped with the then current prepare threshold, which is at least as large as the tid of X, it will be effective in removing the OR table entry, which is timestamped with the tid of X. Thus, timestamping the trim message with the prepare threshold does not hinder it from collecting any garbage that it justifiably could. However, there might be a problem if X was never installed at the receiver because it aborted. To handle such cases, we must maintain the constraint that the prepare threshold is advanced periodically.

There is another issue regarding the liveness property. If the outlist for OR_2 at OR_1 becomes empty, when can OR_1 stop sending trim messages to OR_2? It is not sufficient for OR_1 to send only one trim message containing an empty outlist and stop, because the trim message may be lost, or because it might not result in the removal of all entries in the OR table at OR_2 (if the timestamp of the trim message is not high enough). Therefore, OR_1 must keep sending the trim message periodically, until it receives an ack from the OR_2 indicating that its OR table is indeed empty. In addition, if OR_2 has not received a trim message from OR_1 for a long time and has a non-empty OR table for it, OR_2 can send a query message to OR_1 to prompt it to send its outlist.

[1]This claim is speculative because the research on this issue was still in progress at the time this thesis was written.

5.3 Stability of OR Tables and Outlists

If OR tables are not stable, an OR recovering from a crash must reconstruct its OR tables by sending query messages to other ORs, prompting them to send their outlists. The problem here is that an OR cannot perform a local collection until it has all the OR tables (although it *can* start other work, such as servicing fetch requests). If even a single other OR is inaccessible, the recovering OR will be held up from doing a local collection. The problem can be mitigated to some extent if each OR kept stable information as to which ORs it has non-empty OR tables for. (This scheme is similar to that proposed for FE tables in Section 4.5.2.) Then a recovering OR need depend on only those ORs for their OR tables. A solution based on leases, like the one used for inaccessible front ends (Section 4.5.3), cannot be applied here because it is unacceptable to shut down an OR if the lease runs out. We conclude that the OR tables should be stable indeed.

To maintain OR tables stably, the owner must log updates due to insert messages. This delays the prepare phase, but if the parallel scheme is used, the stable-storage write at the owners is expected to be masked by that at the participants. Further, if the owner is also a participant of the commit protocol, the updates can be merged with the prepare record so that only one stable-storage write is involved. OR table updates arising from trim messages are less of a concern because they occur in the background.

Outlists, on the other hand, need not be "that stable." A recovering OR can compute its outlists by collecting information from the or-surrogates it has. Thus, the recovery of outlists does not depend on other ORs, but scanning the entire repository for or-surrogates is a problem in itself. If the OR does not have its outlists ready soon enough, the collection of distributed garbage at other ORs might be delayed.

Actually, it is cheap to maintain stable outlists. An outlist entry is created at a participant when a transaction being prepared there creates an or-surrogate. Therefore, additions to the outlists can *always* be logged together with the prepare record, thus causing only an insignificant overhead. Entries are deleted from outlist when the corresponding or-surrogate is collected by the local collection. This happens in the background, so additional stable storage writes are acceptable.

5.4 Summary

Each OR maintains OR tables for other ORs that contain references to its objects. The OR table for OR_1 at OR_2 records a conservative estimate of the references OR_1 holds to objects in OR_2.

Entries are made in OR tables as a part of the commit protocol. In the serial insert scheme, when the receiver receives a remote reference in the prepare message, it sends an insert message to the owner if it does not already have an or-surrogate for that reference. In the parallel insert scheme, the coordinator sends insert messages to the owners in parallel with the prepare messages to the participants. Further, if the owner happens to be a participant, the prepare and the insert messages can be merged, and so can be the stable logging of the prepare record and the OR table update.

However, in the parallel scheme, the coordinator sends an insert message for a reference even if the receiver already has an or-surrogate for it. Many of these redundant messages can be avoided if the front end tracks the senders of references by maintaining information regarding or-surrogates in the unswizzle table. Further, if a sender of the reference is a participant of the commit protocol while the owner is not, a synchronous message to the owner can be avoided by creating a translist entry at the sender. There is another benefit in tracking the senders: if the owner is unavailable at the time of the commit, the coordinator can instead contact the sender and use the translist scheme.

Entries in the OR tables are removed by trim messages sent in the background. Late trim messages must be prevented from removing OR table entries created by recent insert messages. This is simple to ensure in the serial scheme, because the receiver itself sends the insert and trim messages to the owner.

In the parallel scheme, however, an insert message is sent by the coordinator, while a trim messages is sent by the receiver itself. Our protocol solves this problem by exploiting some features of the distributed commit protocol in Thor. First, transactions are assigned unique timestamps, or tids, which are totally ordered. Second, each OR maintains a timestamp, called the prepare threshold, such that it will accept to prepare a transaction only if its tid is above this threshold; otherwise, the coordinator must retry the transaction with a larger timestamp. In our protocol, each OR table entry is timestamped with the tid of the transaction that created it, and a trim message sent by an OR is timestamped with the prepare threshold at the OR. A trim message can result in the removal of an entry only if the timestamp of the message is at least as large as that of the entry.

Updates to OR tables are logged on stable storage so that the tables may be reconstructed when an OR recovers from a crash. Outlists, on the other hand can be computed from the stably stored or-surrogates. Nonetheless, we maintain the outlists stably because it is cheap to do so.

Chapter 6

Conclusions

This thesis has presented a design for distributed GC in a new object-oriented database system called Thor [LDS92]. In Thor, a front end (FE) running on behalf of a client invokes operations on copies of objects fetched from multiple object repositories (ORs). Our design accounts for caching and prefetching of objects done by the front end. It also accounts for the distributed commit protocol, which involves the transfer of modified and new objects from the front end to the coordinator OR and then to the participant ORs. In addition to this design, Chapter 2 of the thesis provides a novel analysis of a wide range of distributed GC techniques.

6.1 Summary

The task of the overall garbage collection in Thor is to reclaim objects that are neither reachable from the root directory object at any OR, nor from the handles given out by the front ends to their clients.

We rejected global tracing as a means for distributed GC in Thor because it does not scale well to large systems. Instead, our design is based on each OR maintaining a conservative record of existing remote references to its objects. However, conventional techniques to track remote references are not applicable to Thor because the transfer of references between ORs occurs in a roundabout way. As illustrated in Figure 1-3, the transfer of a reference from the sender OR to the receiver OR involves a fetch by the front end and the distributed commit protocol executed by the coordinator OR.

The performance goal for distributed GC in Thor is to minimize the delay it adds to the execution of fetches and commits, which lie in the critical path of operations invoked by the client.

Our design uses a form of reference tracking we call reference listing. Each OR maintains an FE table for every front end it has sent objects to; the FE table tracks the references the front end holds to objects in that OR. The OR also maintains OR tables to track incoming references from other

ORs; if OR_1 has an *or-surrogate* pointing to an object in OR_2, the OR table for OR_1 at OR_2 has an entry for the referenced object. As discussed in Section 2.4.4, the advantage of using reference listing is that the removal of entries from FE tables and OR tables can be carried out by unreliable messages, called trim messages, sent in the background.

The local GC at the OR reclaims objects that are not reachable from any of the local root directory, the FE tables, or the OR tables. The local GC at the front end reclaims cached objects not reachable from the handle table. The distributed GC protocol is divided into two parts: the FE-OR protocol and the OR-OR protocol.

6.1.1 FE-OR GC Protocol

Chapter 4 described the FE-OR protocol, which is responsible for inserting and deleting entries in the FE tables. Entries are made in an FE table of an OR when the OR sends a block of objects to the front end. Our design employs the following strategies to expedite the fetch:

- When OR_1 sends to the front end an or-surrogate containing a reference to an object at OR_2, no insert message need be sent to OR_2. This is because the or-surrogate at OR_1 is protected by the FE table, which in turn protects the referenced object through the OR table for OR_1 at OR_2. This scheme is conceptually similar to the strong-weak pointers scheme [SDP92].
- Recording all references contained in the block into the FE table would delay the fetch request and inflate the FE table. The scan-on-need scheme described in Section 4.3.2 records only the objects composing the block. The references contained in these objects are included in the FE table only when the stable copy of the object is about to be modified.
- The FE tables are maintained on a lease basis [GC89]; that is, an OR guarantees to maintain an FE table only until its lease expires. When the OR recovers from a crash, it only needs to wait for the lease period to pass before doing a local GC. Then there is no danger of reclaiming an object accessible from an active front end. This scheme avoids the need to maintain FE tables stably, thus eliminating a stable-storage write from the fetch operation.

Entries are removed from the FE table when the front end closes its session with the OR or sends a trim message containing a list of references it continues to hold to objects in that OR. Late trim messages are handled using a simple timestamp protocol; the protocol accounts for the blocks streamed to the front end in the background. Further, the lease mechanism allows the OR to safely reclaim an FE table in the event a network partition disconnects it from the front end.

6.1.2 OR-OR GC Protocol

Chapter 5 described the OR-OR GC protocol, which is responsible for inserting and deleting entries in the OR tables. Entries are made in OR tables as a part of the commit protocol. Figure 4-3 depicts the operation of the overall GC protocol when a reference to an object at the owner OR makes its way from the sender OR to the receiver OR. Before the commit, the referenced object is protected by the combination of the FE table at the sender and the owner's OR table for the sender. After the commit, the front end might discard the reference, thus withdrawing the protection provided by the FE table. Therefore, an entry for the referenced object is made in the owner's OR table for the receiver; this necessitates sending a message to the owner.

Conventional distributed systems using reference tracking for distributed GC are able to eliminate synchronous insert messages by using techniques discussed in Section 2.4, which include the use of a translist [LL86] and strong-weak pointers [SDP92].[1] These techniques could not be applied in Thor because of the following:

1. When the sender sends a reference to the front end, it does not know which ORs will finally receive a copy. This precludes the use of conventional techniques at the time of fetch to account for the transfer of the reference to the receiver.

2. After the commit, the front end cannot be relied upon to retain the reference (say, until the owner has been notified), because it is a temporary entity and cannot be made to wait indefinitely. This precludes the use of conventional techniques at the front end at commit time.

The straightforward way to send an insert message is to use the serial scheme (Section 5.1.1): when a participant receives a remote reference in the prepare message, it sends an insert message to the owner if it does not already have an or-surrogate for that reference. This scheme adds a message round trip and a stable storage update to the commit latency. (Updates to OR tables are logged on stable storage so that the tables may be reconstructed when an OR recovers from a crash.)

Therefore, we suggested the parallel insert scheme (Section 5.1.2): the coordinator sends insert messages to the owners in parallel with the prepare messages to the participants. This masks the latency due to the GC protocol with that due to normal prepare-phase work. Further, if the owner happens to be a participant, the prepare and the insert messages can be merged, and so can the stable logging of the prepare record and the OR table update. A drawback of this scheme is that the coordinator sends an insert message for a reference even if the receiver already has an or-surrogate for it. Section 5.1.4 describes how most of these redundant messages can be avoided if the front end maintains extra information regarding the or-surrogates it has fetched.

Entries in the OR tables are removed by trim messages sent in the background, which contain a list of references one OR continues to hold to objects in another. Late trim messages must be prevented from removing OR table entries created by recent insert messages. In the parallel scheme, a race condition between the two messages cannot be avoided using simple timestamp protocols, because

[1] The use of reference listing preempts the use of some other techniques, like weighted reference counting [Wen79, Bev87]. As discussed in Section 2.4.3, weighted reference counting requires the use of reliable decrement messages, and suffers from the weight-underflow problem.

the insert message is sent by the coordinator while the trim message is sent by the receiver itself. In Section 5.2.2 we described a timestamp protocol that solves this problem by exploiting some features of the distributed commit protocol in Thor. The protocol is comparable to the one used in [SDP92].

6.2 Future Work

6.2.1 A Formal Proof

In this thesis we presented informal and operational arguments to validate parts of the proposed design. It is desirable to have a formal proof that covers the entire design — the FE-OR protocol as well as the OR-OR protocol. Besides serving an academic interest, such a proof might unearth oversights and redundancies in the current design.

6.2.2 Performance Evaluation

At the time this thesis was written, we had implemented the FE-OR protocol and part of the OR-OR protocol in a system called *TH*, a prototype of Thor. Th is implemented in a high level language called *Argus*, which supports abstract data types, garbage collection, remote procedure calls, and transactions [Lis88]. However, the layer of abstraction introduced by Argus makes Th unsuitable as a test bed for evaluating performance in terms of real time. For instance, Argus forces the use of reliable remote procedure calls where unreliable messages could have been used.

We plan to build a simulator that will allow us to better evaluate the trade-offs between some of the alternatives considered in this thesis. We are particularly interested in comparing the performance of three schemes for OR-OR garbage collection:

1. the serial scheme (Section 5.1.1)
2. the parallel scheme (Section 5.1.2 and 5.1.3)
3. the parallel scheme with the front end storing extra information regarding or-surrogates (Section 5.1.4)

The parallel scheme sends insert messages and updates OR tables in parallel with other commit time work, but it might send more messages than the serial scheme. The aim of our simulations will be to see if the cost of sending redundant messages outweighs the advantage from parallelism. The answer is difficult to obtain analytically because it depends upon the characteristics of the workload. For instance, if the creation of new inter-OR references is rare, the serial scheme will send insert messages only rarely. At the same time, if it is not so rare for modified objects sent to the coordinator to contain inter-OR references, the parallel scheme will send many redundant insert messages. The answer also depends upon the relative measures of the computational cost of preparing and accepting messages (the cost incurred in the parallel scheme) and the latency of a round trip delivery (the cost incurred in the serial scheme).

As pointed out earlier, many of the redundant messages sent by the parallel scheme can be avoided if the front end maintains extra information. Equipped with this extension, the parallel scheme is likely to beat the serial scheme. We shall estimate whether maintaining this information has any significant impact on the space and computational overhead at the front end.

6.2.3 Collection of Distributed Circular Garbage

Tracking inter-node references does not, by itself, collect distributed circular garbage (Section 1.2). Since the ORs in Thor are long lived entities, it is important that all garbage ultimately be collected; otherwise, the uncollected garbage might accumulate over time and waste a significant fraction of the storage space. Section 2.5 analyzed a number of techniques that can be used to augment reference tracking to collect all garbage. At present, we are considering two candidate techniques for Thor:

- Migration of remotely referenced objects that are not reachable from the local root directory to the ORs from which they are referenced [SGP90]; this is discussed in Section 2.5.1. The virtue of this technique is that it makes progress by pairwise communication between the ORs on which the cycle resides. The drawback is that the strategy might result in load-imbalance among the ORs.

- Hierarchical grouping of ORs such that tracing can be performed in each group independently of other ORs [LQP92]; this is discussed in Section 2.5.6. Groups are made on the basis of expected locality in inter-OR references; for instance, all ORs on the same local area network could be grouped together. When a group is traced, all circular garbage lying within the group is collected. The local tracing at each OR can be harnessed to propagate marking on behalf of the group tracing.

While distributed circular garbage in Thor must ultimately be collected, we believe that its occurrences are infrequent relative to non-circular garbage, and therefore it is feasible to collect circular garbage lazily.

Bibliography

[Ady93] A. Adya. *A Distributed Commit Protocol for Optimistic Concurrency Control*, Master's thesis, Massachussetts Institute of Technology, 1993 (forthcoming).

[Ali84] K. A. M. Ali. Garbage Collection Schemes for Distributed Storage Systems. *Proceedings of Workshop on Implementation of Functional Languages*, pages 422–428, Aspenas, Sweden, Feb 1985.

[Bar78] J. F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. *Techincal Report 88/2*, Digital Western Research Laboratory, Palo Alto CA, Feb 1988.

[Bev87] D. I. Bevan. Distributed Garbage Collection Using Reference Counting. *Lecture Notes in Computer Science 259*, pages 176–187, Springer-Verlag, Jun 1987.

[Bis77] P. B. Bishop. Computer Systems with a Very Large Address Space, and Garbage Collection. *Technical Report MIT/LCS/TR-178*, MIT Laboratory for Computer Science, Cambridge MA, May 1977.

[Che70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11), pages 677–678, Nov 1970.

[Chr84] T. W. Christopher. Reference Count Garbage Collection. *Software Practice and Experience*, 14(6), pages 503–507, Jun 1984.

[Day93] M. Day. *Managing a Cache of Swizzled Objects and Surrogates*, Ph.D. thesis, Massachussetts Institute of Technology, 1993 (forthcoming).

[DLMSS78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Schloten, and E. F. M. Steffens. On-the-fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11), pages 966–975, Nov 1978.

[DS80] E. W. Dijkstra, and C. S. Schloten. Termination Detection for Diffusing Computation. *Information Processin Letters*, Vol. 11, No. 1, Aug 1980.

[GC89] C. Gray, and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 202–210, 1989.

[Gol89] B. Goldberg. Generational Reference Counting: A Reduced Communication Distributed Storage Scheme. *ACM SIGPLAN Programming Languages Design and Implementation*, pages 313–321, portland OR, Jun 1989.

[Gra78] J. N. Gray. Notes on Database Operating Systems. *Operating Systems: An Advanced Course (Lecture Notes in Computer Science 60)*, pages 393–481, Springer-Verlag, 1978.

[HK82] P. Hudak, and R. Keller. Garbage Collection and Task Deletion in Distributed Applicative Processing Systems. *ACM Symposium on Lisp and Functional Programming*, pages 168–178, Aug 1982.

[Hug85] J. Hughes. A Distributed Garbage Collection Algorithm. *Functional Programming and Computer Architecture (Lecture Notes in Computer Science 201)*, pages 256–272, Springer-Verlag, Sep 1985.

[Juu90] N. C. Juul. A Distributed Faulting Garbage Collector for Emerald. *OOPSLA Workshop on Garbage Collection*, 1990.

[KR81] H. T. Kung, J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), pages 213–226, Jun 1981.

[LDS92] B. Liskov, M. Day, and L. Shrira. Distributed Object Management in Thor. *Distributed Object Management*, ed. M. T. Ozsu, U. Dayal, and P. Valduriez, Morgan Kaufmann, 1992.

[LH83] H. Lieberman, and C. Hewitt. A Real-time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6), pages 419–429, Jun 1983.

[LL86] B. Liskov, and R. Ladin. Highly Available Distributed Services and Fault Tolerant Distributed Garbage Collection. *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, ACM, pages 29-39, Vancouver, Canada, Aug 1986.

[LL92] R. Ladin, and B. Liskov. Garbage Collection of a Distributed Heap. *Int. Conference on Distributed Computing Systems*, pages 708–715, Yokohoma, Japan, Jun 1992.

[LLSG90] R. Ladin, B. Liskov, L. Shrira, S. Ghemawat. Lazy Replication: Exploiting the Semantics of Distributed Services. *Proceedings of the 9th ACM Symposium on the Principles of Distributed Computing*, ACM, Canada, Aug 1990.

[LQP92] B. Lang, C. Queinnec, and J. Piquer. Garbage Collecting the World. *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 39–50, Albuquerque, Jan 1992.

[LS79] B. W. Lampson, and H. E. Sturgis. *Crash Recovery in a Distributed Data Storage System.* Technical Report, Xerox Palo Alto Research Center, Apr 1979.

[Lis88] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3), pages 300–312, Mar 1988.

[Lis92] B. Liskov. Preliminary Design of the Thor Object-Oriented Database System. *Programming Methodology Group Memo 74*, MIT Laboratory for Computer Science, Mar 1992.

[Lis93] B. Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. *Distributed Computing*, 1993 (forthcoming).

[Mos90] J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Tran. on Information Systems*, 8(2), pages 103–139, Apr 1990.

[OL88] B. M. Oki, and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly Available Distributed Systems. *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, ACM, Aug 1988.

[Piq91] J. M. Piquer. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. *PARLE '91 — Parallel Architecture and Languages (Lecture Notes in Computer Science 505)*, pages 150–165, Springer-Verlag, Jun 1991.

[Ran83] S. P. Rana. A Distributed Solution to the Distributed Termination Problem. *Information Processing Letters*, Vol. 17, pages 43–46, Jul 1983.

[SDP92] M. Shapiro, P. Dickman, and D. Plainfosse. Robust, Distributed References and Acyclic garbage Collection. *Symposium on Principles of Distributed Computing*, Vancouver, Canada, Aug 1992.

[SGP90] M. Shapiro, O. Gruber, and D. Plainfosse. A Garbage Detection Protocol for a Realistic Distributed Object-Support System. *Research Report 1320*, INRIA–Rocquencourt, Nov 1990.

[Ves87] S. C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, University of Washington, Jan 1987.

[Wen79] K-S Weng. An Abstract Implementation of a Generalized Dataflow Language. *Technical Report MIT/LCS/TR 228*, MIT Laboratory for Computer Science, Cambridge MA, 1979.

[Wil92] P. R. Wilson. Uniprocessor Garbage Collection Techniques. *1992 International Workshop on Memory Management, (Lecture Notes in Computer Science 637)*, Springer-Verlag, 1992.