Garbage Collection in a Large, Distributed Object Store

Umesh Maheshwari

September, 1997

MIT-LCS-TR-727

MIT Laboratory for Computer Science
545 Technology Square, Cambridge MA 02139

Abstract

Systems that store a large number of persistent objects over many sites in a network pose new challenges to storage management. This thesis presents a comprehensive design for collecting garbage objects in such systems. The design achieves scalability by partitioning the system at two levels: Each site traces its objects independently of other sites, and the disk space at each site is divided into partitions that are traced one at a time in main memory. To trace a site independently of other sites, and a partition independently of other partitions, objects reachable from other partitions or other sites must be treated as roots. This introduces two problems. First, maintaining up-to-date information about inter-site and inter-partition references can stall applications and increase usage of disk, memory, and the network. Second, treating these references as roots does not allow collection of cycles of garbage objects spanning multiple sites or multiple partitions. Solutions to these problems have been proposed in single-site or distributed systems, but they do not scale to many partitions or many sites. This thesis presents scalable solutions to these problems.

The thesis provides new techniques to organize and update a potentially large amount of inter-partition information such that the information is recoverable after a crash and disk time is used efficiently. It also provides efficient techniques to record inter-site references in a system with client caches and multi-server transactions. A client might cache a large number of references to server objects; therefore, we identify a minimal subset of these references that must be recorded for safe collection at servers. We use a new protocol to manage inter-server references created by distributed transactions. This protocol sends messages in the background to avoid delaying transaction commits. We use another new protocol to handle client crashes; the protocol allows servers to safely discard information about clients that appear to have crashed but might be live.

The thesis provides different schemes to collect inter-partition and inter-site garbage cycles. Inter-partition cycles are collected using a site-wide marking scheme; unlike previous such schemes, it is piggybacked on partition traces, does not delay the collection of non-cyclic garbage, and terminates correctly in the presence of modifications. Inter-site cycles, on the other hand, are collected using a scheme that minimizes inter-site dependence. It is the first practically usable scheme that can collect an inter-site garbage cycle by involving only the sites containing the cycle. The scheme has two parts: the first finds objects that are highly likely to be cyclic garbage, and the second checks if they are in fact garbage. The first part uses a new heuristic based on inter-site distances of objects. It has little overhead and it can be made arbitrarily accurate at the expense of delay in identifying garbage. We provide two alternatives for the second part. The first migrates a suspected garbage cycle to a single site, but unlike previous such schemes, it avoids migration as much as possible. The second traces backwards from a suspect to check if it is reachable from a root. We provide the first practical technique for back tracing in the presence of modifications.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

This thesis was guided and criticized by my research advisor, Prof. Barbara Liskov. I have benefited from her keen mental vision in problems of technical nature and presentation. She provided prompt feedback on the thesis.

My thesis readers, Prof. John Guttag and Prof. Dave Gifford, helped with bringing out the significance of this thesis. Phil Bogle, Dorothy Curtis, and Robert Ragno proofread parts of the thesis on short notice. Much of the work presented here is derived from papers published previously [ML94, ML95, ML97b, ML97a]. I am grateful to the anonymous referees who judged these papers. I am also grateful to various members of LCS who proofread these papers and gave useful comments: Atul Adya, Chandrasekhar Boyapati, Miguel Castro, Sanjay Ghemawat, Robert Gruber, Frans Kaashoek, Andrew Myers, Liuba Shrira, Raymie Stata, and James O'Toole. In addition to these people, others gave me comments during practice talks: Kavita Bala, Dorothy Curtis, John Guttag, Paul Johnson, Ulana Legedza.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research, under contract N00014-91-J-4136.

There are some people whose help transcends this thesis; to thank them here is neither necessary nor sufficient.

Contents

List of Figures

Chapter 1

Introduction

Object stores, also known as object databases, allow user applications to store and share objects. Emerging distributed systems will allow applications to access objects stored at a large number of sites, and each site may store a large number of objects on disk. The scale of such systems poses new challenges for efficient use of storage, especially for reclaiming the storage of unusable objects. This thesis presents techniques to meet these challenges. This chapter describes the context and the contribution of the thesis.

1.1 The Storage Problem in Object Stores

Applications view an object store as a collection of objects that contain data and references to each other. An application may invoke operations that read or modify existing objects or create new objects. Objects created by one application may be accessed by other applications, even after the application has finished. To facilitate such access, certain objects are assigned user-level names by which they can be accessed from any application and at any time; these objects are known as *persistent roots*. Thus, an application may begin by accessing a persistent root and then traverse references to reach other objects. This view of an object store is illustrated in Figure 1.1.

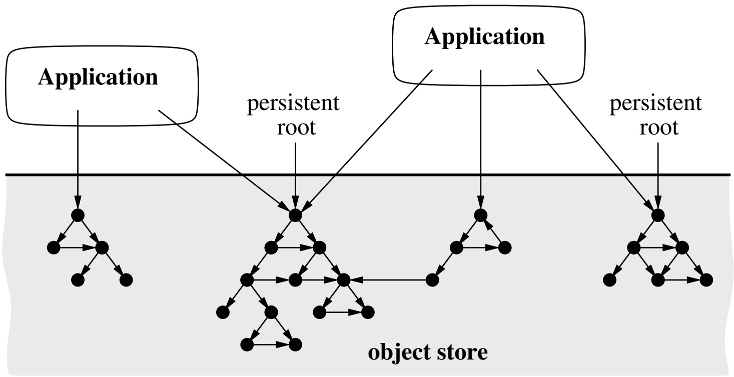

Figure 1.1: An object store provides a shared collection of objects.

New objects require resources such as disk space and object names. Since these resources are limited, the continual demand for them must be met by reclaiming objects that are of no further use to applications. These objects are said to be *garbage*, while others are said to be *live*. Reusing the storage of garbage objects does more than avoid running out of disk space: it achieves higher spatial density of live objects. Higher spatial density results in better utilization of memory caches and disk and network bandwidth.

There are two ways of reclaiming garbage objects:

1. *Explicit deletion* by the application programmers, based on their knowledge of object semantics and usage.
2. *Garbage collection* by the underlying runtime environment, based on the reachability of objects from applications.

Explicit deletion is used in many programming languages. It identifies garbage with little runtime overhead; it has the potential to identify more garbage than automatic detection—and sooner—because the programmer may exploit program semantics to decide when an object is not useful. However, explicit deletion is a burden on programmers and is prone to errors. Two kinds of errors are possible. If a garbage object is not deleted, its storage will be lost. On the other hand, if a live object is deleted, an application may attempt to use the object and fail. References to such an object are called *dangling references*. Explicit deletion becomes even more problematic in object stores for the following reasons. Objects may be shared by many applications and there may be many different ways of accessing them. Therefore, it is very difficult for a programmer to decide when to delete an object. Further, objects are persistent, so the effects of deleting live objects or retaining garbage objects will last for ever.

Therefore, object stores need garbage collection. An underlying component, the garbage collector, automatically finds and reclaims objects that cannot be reached from applications. An object is reachable from an application only if it is reachable from a persistent root or an application variable. Reachability is determined by the existence of a path of zero or more connecting references. This model for storage management is often known as "persistence by reachability" [ABC$^+$83].

In a distributed object store, objects may contain references to objects at other sites. Objects are expected to be clustered at sites such that remote references are infrequent compared to local references. However, garbage collection must account for local as well as remote references to determine reachability.

Garbage collection might not be suitable for a network of sites owned by many users with little common interest, such as the Internet. In such networks, the need for autonomous control of storage overrides the need to prevent dangling references from other sites. For examples, the administrator of a Web server might want to remove local documents although other sites might contain references to those documents. However, garbage collection *is* suitable for a network of sites owned by a corporation or by users with a shared interest. In such networks, there is an interest in preventing dangling references. Note that even corporate networks may be geographically distributed and may span thousands of sites.

Indeed, many Internet-wide object stores are not based on the object model described above. For example, most objects in the Web are relatively coarse-grained since they are meant for direct viewing by humans, and these objects have user-level names (URL's) by which they can be accessed provided they have not been deleted by the administrator. In contrast, this thesis applies to object stores that are programmable, that store fine-grained objects for use by application programs, and that provide reliable access to data. Thus, it applies to newly popular distributed object systems such as Java Remote Method Invocation.

1.2 Goals for Garbage Collection

Ideally, the garbage collector should have the following properties:

Safe It should collect only garbage objects. In general, the garbage collector should not change the application semantics.

Complete It should collect all garbage objects eventually. This is particularly important in persistent systems because even small amounts of uncollected garbage can accumulate over time and cause significant loss of storage.

Timely It should collect most garbage objects quickly enough that applications are not blocked for free space when creating new objects. Timely collection is also required for better spatial locality of live objects.

Efficient It should run in the background so that it does not stall the applications. Further, it should have low utilization of resources such as the processor, memory, disk, and network.

Fault Tolerant It should tolerate site crashes and lost messages. Sites should collect as much garbage as possible even when other sites or parts of the network are down or slow.

Scalable The garbage collector should retain the above properties as more sites are added or as more objects are added to a site.

However, as is evident in the discussion below, garbage collection in a large, distributed store is difficult because many of these requirements conflict with each other.

A simple method to collect garbage is to *trace* all objects reachable from the roots and then collect objects not visited by the trace. However, a trace of the global object graph would require all sites to cooperate and finish the trace before any site can collect any garbage. Such a trace would not be timely or fault tolerant because every site must wait until the global trace has finished. Timeliness and fault tolerance require that a site trace local objects independently of other sites. This is the approach taken in many distributed systems [Ali85, Bev87, SDP92, JJ92, BEN+93, ML94]. Local tracing minimizes inter-site dependence. In particular, garbage that is not referenced from other sites is collected locally. Further, a chain of garbage objects spanning multiple sites is collected through cooperation within the sites holding the chain. We call this feature the *locality property*.

The problem with a global trace reappears within a site in a different context. Each site may have a large disk space and a relatively small main memory. Often, the memory size is only a hundredth of the disk size. Therefore, a trace of the site's objects is likely to thrash on disk due to poor locality of references [YNY94]. In particular, a disk page might be fetched and evicted many times as different objects on the page are scanned. This is undesirable for two reasons. First, the trace might take a long time to finish, preventing timely collection of garbage. Second, which is worse, the trace would use significant disk bandwidth and memory space, which are crucial resources for good performance of applications. Thus, timeliness and efficiency require that the disk space be divided into smaller *partitions* that are traced independently. This is the approach taken in many single-site systems with large, disk-based heaps [Bis77, YNY94, AGF95, MMH96, CKWZ96]. Ideally, partitions should be large enough to be an efficient unit of tracing, and small enough to fit in a fraction of main memory.

However, to trace a site independently of other sites, and a partition independently of other partitions, objects reachable from other sites or other partitions on the same site must not be collected. Therefore, references to objects from other sites or other partitions must be treated as roots for tracing. This introduces two problems:

Efficiency Maintaining up-to-date information about inter-site and inter-partition references might delay applications and increase utilization of memory, disk, and network.

Completeness Treating inter-site and inter-partition references as roots fails to collect cycles of garbage objects spanning multiple sites or multiple partitions.

While some solutions to these problems have been proposed in single-site or distributed systems, they do not scale to many partitions or many sites. For example, they are not efficient in managing large numbers of inter-partition references, and they do not preserve the locality property in collecting inter-site garbage cycles.

1.3 Contributions

This thesis presents a comprehensive design for garbage collection in a large, distributed object store. The design includes a collection of new techniques that meet the challenges posed in the previous section. The focus of the thesis is the management of inter-partition and inter-site references.

1.3.1 Inter-Partition References

The information about inter-partition references is recorded on disk for two reasons. First, if this information is not persistent, recomputing it after a crash by scanning the entire disk would take a long time; until the information is retrieved, no partition can be collected. Second, a large disk may contain enough inter-partition references that recording them in memory would take up substantial space. Maintaining inter-partition information persistently requires care in keeping the disk utilization low, both for the garbage collector to perform well and, more importantly, to avoid degrading service to applications. We present new techniques to organize and update this information with the following benefits:

1. Information about inter-partition references is recovered quickly after a crash.
2. Disk accesses to add new inter-partition references to the information are batched.
3. Unnecessary references in the information are dropped efficiently.

We have implemented these techniques, and we present a performance study to evaluate their benefits. The study shows that if not enough memory is available to cache the inter-partition information, the disk-time overhead of maintaining the information can be significant compared to the overhead of installing modified objects on disk. Our techniques reduce the disk-time overhead of maintaining inter-partition information by a factor of 100. These techniques exploit a stable log of modified objects; most object stores contain such a log to support transactions. (A transaction is a sequence of operations that behaves atomically with respect to other transactions and crashes [Gra78].)

Partitioned garbage collection involves a wide range of other issues such as the choice of the tracing algorithm (marking, copying [Bak78], replicating [ONG93], etc.), the selection of partitions to trace [CWZ94], the rate of starting traces [CKWZ96], handling transactional rollback [AGF95], etc. This thesis either does not cover the above issues or addresses them only marginally. These issues have been discussed by other researchers and most of their proposals can be applied in conjunction with the techniques proposed in this thesis.

1.3.2 Inter-partition Garbage Cycles

Treating inter-partition references as roots retains inter-partition garbage cycles and objects reachable from them. Inter-partition cycles are common in practice; for example, in many CAD applications a container object points to its parts and the parts point back to the container—thus creating a huge tree with doubly linked edges [CDN93].

We present a site-wide marking scheme for collecting cycles between partitions on the same site. Complementary global marking has been proposed earlier to collect cyclic garbage in partitioned schemes. We present the first such scheme that has the following features [ML97b]:

1. It piggybacks site-wide marking on partition tracing, adding little overhead to the base scheme.
2. It does not delay the collection of non-cyclic garbage.
3. It terminates correctly in the presence of concurrent modifications.

The scheme also preserves the disk-efficient nature of partition tracing. As with all global schemes, it can take a long time for site-wide marking to terminate, but that is acceptable assuming cyclic garbage spanning partitions is generated relatively slowly. The techniques underlying the scheme are applicable to all partitioned collectors.

1.3.3 Inter-Site References

We present separate techniques for recording inter-partition and inter-site references to match their efficiency and fault-tolerance requirements:

1. A site is allowed to configure its partitions dynamically, but other sites are shielded from such reconfigurations: they view the site as a single partition.
2. Information about an inter-partition reference is shared between the source and the target partitions, but information about an inter-site reference is maintained on both the source and the target sites to avoid unnecessary messages.
3. Newly created inter-partition references can be processed lazily because a single thread traces the various partitions at a site. However, newly created inter-site references must be processed eagerly because different sites trace partitions asynchronously.

Most of the previous partitioned collectors are designed for either a distributed system with one partition per site or a single-site system with multiple partitions. Few collectors handle multiple partitions within multiple sites [FS96], but they do not differentiate between inter-partition and inter-site references.

We present techniques to support client-server mechanisms such as client caching and multi-server transactions. Server sites store objects on disk, while a client runs at each application site and executes operations on local copies of objects. A client fetches objects from servers into its cache as required. New and modified objects in the client cache are sent to the servers when an application commits the current transaction. This architecture is illustrated in Figure 1.2. Here, inter-site references include references from client caches to server objects as well as references between server objects.

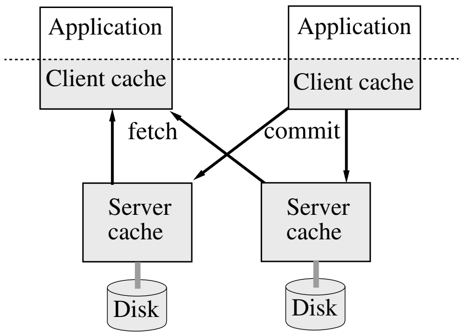

Figure 1.2: A client-server implementation of an object store.

One problem is that a client may fetch and cache many objects from servers, potentially containing millions of references. Recording all such references would be a large overhead and might even prevent the collection of some garbage objects at servers. We present an efficient technique that allows a server to record a minimal set of references held by a client. We show that it is necessary and sufficient to record this set for safe collection. We also describe how this set is maintained without adding overhead when clients fetch objects. This technique is applicable to cache-coherent or transactional systems, which either ensure that objects cached by clients are up to date or abort client modifications based on out-of-date objects.

Another problem concerns recording inter-server references. Previous distributed collectors were mostly designed for RPC-based (or control-shipping) systems, where inter-server references were created through remote procedure calls [BN84]. However, in client-caching (or data-shipping) transactional systems, inter-server references are created when clients commit transactions. We present a new protocol for recording inter-server references created this manner. This protocol is piggybacked on the commit protocol, but it does not delay the commit by sending extra foreground messages or by adding extra participants. All extra messages are sent in the background, so they may be deferred and batched.

We present techniques for tolerating client and server crashes. Servers are expected to recover from crashes while clients might not recover. Servers recover inter-partition and inter-site information by storing it on disk. A problem in handling client crashes is that a live client might appear to have crashed because of communication problems. We present a protocol that allows servers to discard state for clients that appear to have crashed. In particular, the protocol ensures that a live client that was presumed dead does not introduce dangling references into objects at servers.

1.3.4 Inter-Site Garbage Cycles

Inter-site cycles are relatively uncommon, but they do occur in practice. For example, hypertext documents often form complex cycles spread over many sites. A global marking scheme is not suitable for collecting inter-site cycles because sites may fail independently or, equivalently, become overloaded and unresponsive. Thus, global marking might never complete in a large system. The challenge in collecting an inter-site garbage cycle is to preserve the locality property, that is, to involve only the sites containing the cycle. This has proven difficult: most previous schemes do not preserve locality [Ali85, JJ92, LQP92, MKI$^+$95, RJ96]. The few that do seem prohibitively complex or costly [SGP90, Sch89, LC97].

We present the first practical scheme with locality to collect inter-site cycles. It has two parts. The first part identifies objects that are highly likely to be cyclic garbage—the *suspects*. This part is not safe in that it might suspect live objects, although a performance requirement is that few suspects be live. Suspects are found by estimating the minimum inter-site *distances* of objects from persistent roots. This technique preserves locality, has very little overhead, and guarantees that all cyclic garbage is eventually detected.

The second part checks if the suspects are in fact garbage. This part has the luxury of using techniques that would be too costly if applied to all objects but are acceptable if applied only to suspects. We present two alternatives for checking suspects. The first *migrates* the suspects such that a distributed cycle converges to a single site [ML95]. Unlike previous migration-based proposals [SGP90], it avoids migration as much as possible: both in the number of objects migrated and the number of times they are migrated. The second technique traces *back* from the suspects to check if they are reachable from a persistent root. Unlike forward global tracing, this approach preserves locality and scalability. Back tracing was proposed earlier by Fuchs [Fuc95]. However, this proposal assumed that inverse information was available for references, and it ignored problems due to concurrent mutations and forward local traces. We present efficient techniques for conducting back tracing that handle these and other practical problems. We show that the scheme is safe and collects all inter-site garbage cycles.

These techniques are applicable to both client-caching and RPC-based distributed systems, and to transactional and non-transactional systems.

1.3.5 Limitations

Popular use of distributed object stores is in its infancy and little factual information is available about them, especially about the distribution of garbage objects in such systems. Therefore, this thesis provides robust techniques that will work in a wide range of workloads. It also provides a performance analysis of some techniques using synthetic workloads that allow us to control the distribution of object references. However, a factual assessment of the full design would require either implementing it in a distributed system in popular use, or simulating the distribution of garbage in such a system. This thesis does not provide such an assessment.

1.4 Outline

The rest of this thesis is organized as follows:

- Chapter 2 describes the system that is the context of this work.
- Chapter 3 presents techniques for maintaining information about inter-partition references within a site and presents a performance evaluation.
- Chapter 4 presents a marking scheme to collect inter-partition garbage cycles within a site.
- Chapter 5 presents techniques for maintaining information about inter-site references, including client-to-server and inter-server references.
- Chapter 6 presents a scheme with locality to collect inter-site garbage cycles, including a technique for finding suspected cyclic garbage and two techniques for checking them.
- Chapter 7 contains conclusions on the overall design, some guidelines for evaluating the performance, and some directions for future research.

Chapter 2

The Context

This thesis was developed in the context of Thor, a state-of-the-art distributed object database [LAC+96]. This chapter describes the parts of Thor that are relevant to the garbage collector. Some details are presented for the sake of completeness; most techniques proposed in this thesis are applicable to other distributed object stores.

2.1 The Application Interface

Applications view a Thor database as a shared collection of objects. An application identifies objects by *handles*. It begins its interaction with Thor by asking for a handle to some persistent root object using a user-level name. It may then invoke operations—passing handles or numeric data as parameters and receiving handles or numeric data as results. The application may release some handles when it has finished using them. An application releases all handles when it finishes its interaction with Thor. A handle is meaningful only within an application and only until it is released, just like file handles in Unix operating systems.

An application groups its operations into transactions to tolerate failures and concurrent modifications by other applications. A transaction behaves as if its operations happened atomically with respect to failures and other transactions [Gra78]. An application specifies transaction boundaries by requesting a *commit*, which ends the ongoing transaction and starts another. A transaction might fail to commit, or *abort*, for various reasons. In this case, the updates made during the transaction are undone. Further, new handles given to the application during an aborted transaction are revoked and cannot be used subsequently.

Applications may use handles obtained during one transaction in subsequent transactions. Some other systems constrain applications to release all handles with each transaction [AGF95], so an application must start each transaction from the persistent root. This constraint implies that garbage collection need not account for the handles held by applications. However, it makes writing applications inconvenient. More importantly, it does not allow use of volatile objects across transactions; volatile objects are a useful optimization discussed in Section 2.2. Therefore, our model does not have this constraint.

2.2 Objects at Clients

Thor is implemented as a client-server system. A Thor client runs at an application site and executes application requests on local copies of objects. This may require fetching objects from servers into a local cache. New objects created when executing an operation are kept in a *volatile* heap within the client. A volatile object becomes persistent when a transaction makes it reachable from a persistent root. The notion of volatile and persistent objects is illustrated in Figure 2.1.

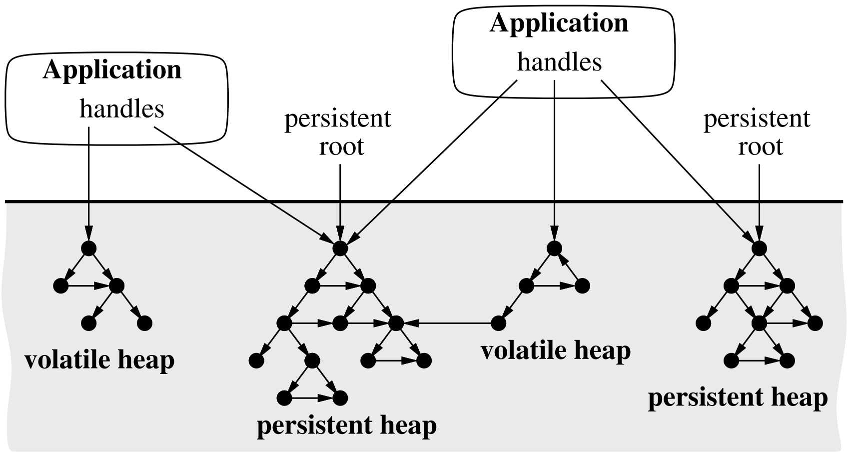

Figure 2.1: Volatile and persistent objects.

When a transaction commits, the client does the following:

- It sends copies of persistent objects that were modified during the transaction to the servers where they reside.
- It sends copies of all volatile objects reachable from some modified persistent object to chosen server sites.

For example, in Figure 2.2, the client has cached persistent objects a and b. It modifies b to point to the volatile object e. When the transaction is committed, e and f become persistent and are copied to the server along with the modified copy of b. Note that the committed state of a persistent object never points to a volatile object, although a volatile object may point to a persistent object.

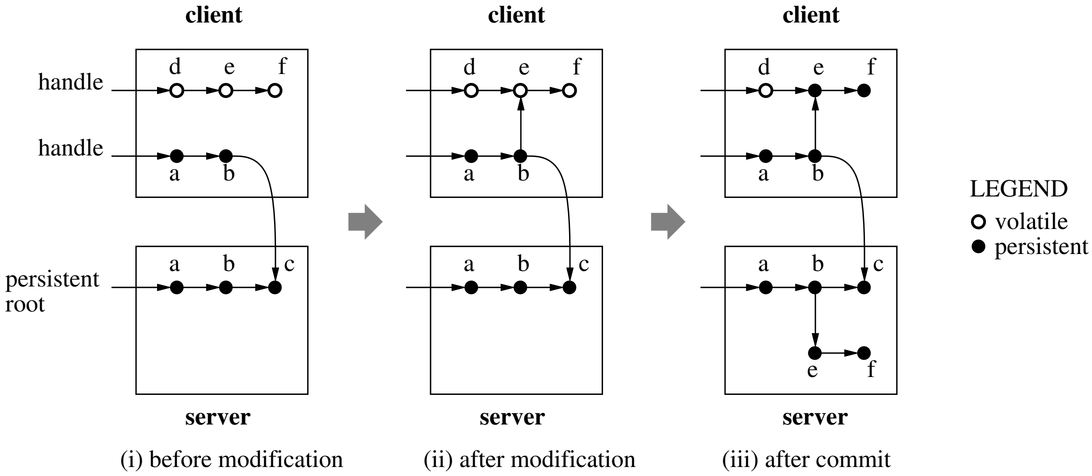

Figure 2.2: How volatile objects become persistent.

Note that an application's volatile objects cannot be reached from other applications. Further, when the application finishes, the volatile heap becomes unreachable (garbage). The volatile heap is a useful optimization because it provides cheap storage for temporary objects used by an application. The application may hold handles to volatile as well as persistent objects and access them in subsequent transactions. Without this facility, *all* objects that an application needs to access across transactions must be copied to servers, which would delay applications and waste resources.

Modifications might make a persistent object e unreachable from persistent roots while it is reachable from an application handle. For simplicity, Thor does not revert the status of such an object to volatile: e continues to be stored at the server. The application from which e is reachable might make it reachable from persistent roots again.

The volatile as well as the persistent heaps need to be garbage collected. Since the volatile heap in a client is not referenced from other sites and is not persistent, its collection is similar to that in conventional environments. This thesis focuses on the garbage collection of the persistent objects at the servers. However, the garbage collection at the servers must account for references from volatile heaps and handles at clients.

2.3 Objects at Servers

Persistent objects are stored in disk pages at server sites. A *page* is a fixed-sized unit for reading and writing the disk. Objects do not cross page boundaries; objects that would be larger than a page are implemented as a collection of smaller objects. Further, an object does not grow or shrink in size after it is created, which simplifies storage management. Abstractions such as resizable arrays are implemented as a tree of fixed-size blocks.

When an object becomes persistent, it is assigned a globally unique name by which other objects, possibly at other sites, may refer to it. These names are designed to locate objects efficiently. A name identifies the server site, the page where the object resides, and an index within the page. A header within each page maps indexes to the offsets of the objects within the page. Thus, an object may be moved within its page without changing the name by which it is referenced. This organization is illustrated in Figure 2.3; it is similar to *slotted pages* used in databases.

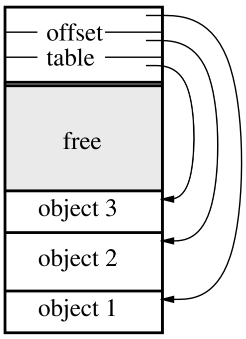

Figure 2.3: A slotted page.

Object names are recycled to save space. That is, after a garbage object is deleted, its name may be assigned to a new object. Successive objects assigned the same name are called *incarnations*. If an incarnation number were used to make names unique across time, each reference would use more space. A small incarnation number would not suffice because of the danger of its wrapping around.

Objects may contain references to objects at other server sites. We assume that objects are clustered such that remote references are infrequent compared to local references. This assumption is likely to be valid because a server stores a large number of objects, potentially billions, which provides ample opportunity for clustering objects to improve locality.

In most cases, a server treats an object as a vector of untyped fields. However, the garbage collector needs to distinguish references and numeric data stored in an object in order to trace objects. This information is found in a *class* object that describes the layout of all objects belonging to the class. The header of each object contains a reference to its class object.

A server maintains a cache of pages to serve fetch requests from clients. Besides a page cache, the server contains a *modified object buffer* to store newly persistent or modified copies of objects returned by clients [Ghe95]. This buffer allows the server to defer and batch the installation of new and modified objects into their disk pages.

2.4 Concurrency Control

The techniques described in this thesis are applicable to a wide range of transactional systems. In particular, a client need not lock objects before using them; locking objects would require contacting the server and possibly other clients [CFZ94]. Not locking objects puts a greater demand on garbage collection because clients might cache old copies of objects that are invalid. For concreteness, we describe the transactional mechanisms employed in Thor.

Thor uses optimistic concurrency control [AGLM95]. A transaction does not lock the objects it uses; instead, it is validated against previous transactions when it commits. Validation requires information about the objects read and modified by the transaction. This information is logged by the client and sent to the servers at commit time. In the current design, the client is dedicated to a single running application, so the client runs one transaction at a time.

In order to validate transactions, a server keeps a conservative record of the pages that any given client may have cached. When a client X commits a transaction modifying object b, the server puts b in the *invalid set* of all clients other than X that might have cached b. Further, the server sends an *invalidation message* to each such client in the background[1]. When a client receives an invalidation for an object b, it evicts b from its local cache. Also, if the client has used b in its current transaction, it aborts the transaction. When the client acknowledges the invalidation message, the server removes b from the invalid set of the client. Finally, when validating a transaction, if the server finds that an object used by a transaction is in the client's invalid set, the server aborts the transaction.

A client may evict pages that it read during the current transaction, but proper invalidation requires that the server keep information about these pages until the end of the transaction. Therefore, clients inform servers of evicted pages only at the end of transactions.

When a client learns that its transaction was aborted, it revokes the handles created during the transaction and reverts the modified objects to their states when the transaction started. To aid this, the client makes a *base copy* of an object before it is modified in a transaction. It retains the base copies of volatile objects until the transaction ends. It need not retain base copies of persistent objects, since they are available at the servers.

2.5 Commit Protocol

The commit protocol is relevant to garbage collection because it dictates how object modifications are applied. At commit time, the client finds the *participant* servers whose objects were read or

[1] A *background* message is buffered in the send queue until an opportune moment, such as when another message must be sent immediately or when many background messages have accumulated.

written during the transaction. If there is more than one participant, it uses the following two-phase commit protocol [Gra78]. It sends the copies of new and modified objects to a participant chosen as the *coordinator*. The coordinator sends a *prepare* message to each participant. A participant tries to validate the transaction. If the transaction cannot be committed for some reason, the participant sends a negative vote to the coordinator; otherwise, it sends a positive vote. If the coordinator receives a negative vote, it aborts the transaction and sends an *aborted* message to the client and the participants where objects were created or modified. Otherwise it logs a commit record and sends a *committed* message.

The flow of new and modified objects from the client cache to disk pages is shown in Figure 2.4. When a server prepares a transaction, it stores the new and modified objects into a prepare record in a stable log. Conventional systems use a disk-based log, but Thor uses an in-memory log replicated on several machines [Ghe95]. Modifications stored in prepare records are not visible to clients. When a transaction commits, the server *applies* the objects in the prepare record by moving them into the modified object buffer, which is also a part of the in-memory stable log. Modifications of single-site transactions are applied immediately after validation, but those of distributed transactions are applied after receiving a committed message.

Since committed messages for prepared transactions might not be received in the order in which the transactions are serialized, their modifications might be applied out of order. Further, objects in the modified buffer are installed into the disk pages in the background to optimize disk utilization. Again, modifications might be installed out of order. Applying and installing modifications out of order makes it difficult for the garbage collector to obtain a safe view of the object graph.

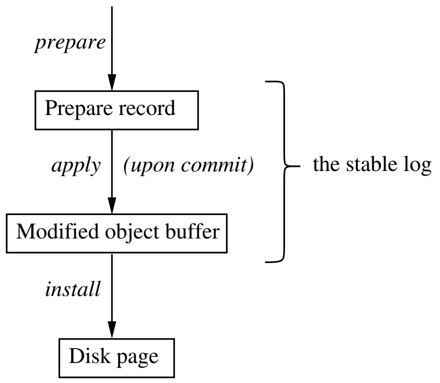

Figure 2.4: The flow of new and modified objects.

Some object stores use *steal* buffer management [EH84], which allows uncommitted modifications to be installed on disk pages. Thor does not use this policy for reasons provided in [Ghe95]. Not installing uncommitted data also simplifies garbage collection. Other researchers have solved garbage-collection problems due to uncommitted data that is installed [AGF95]. Their solutions can be used in conjunction with the techniques suggested in this thesis.

2.6 Failure Model

A server is expected to recover from crashes without much delay. When a server crashes, it loses the data in its main memory but retains the data on its disk. Further, the server log is stable through replication [LGG$^+$91]. This thesis does not account for disk failures or other non-recoverable failures in servers.

A Thor client might crash at any time and never recover. We assume that clients are well-behaved and their crashes do not cause arbitrary or malicious behavior. We account for the possibility that a network partition might be mistaken for a client crash even when the client is alive. This is because network failures are practically indistinguishable from failures of the communicating sites.

Clients and servers may be distributed over a wide-area network. We assume the presence of a reliable channel for ordered delivery of messages between any pair of sites. Such delivery can be provided efficiently even if the underlying network is unreliable [Pos81, LSW91, Mah97]. However, all practical protocols for reliable delivery might lose the last message if an end-site crashes. In this case, the end-site is aware of this possibility, but it cannot tell whether a message was lost for sure.

Chapter 3

Partitioned Garbage Collection

This chapter describes techniques for independent tracing of disk partitions at a server. Partitions are needed because a trace of the entire disk using a relatively small main memory would thrash on the disk. For a partition trace to be safe, references to its objects from other partitions and other sites must be treated as roots. The chapter focuses on maintaining information about inter-partition references such that the information is recoverable and efficient to use and update. The chapter ignores the management of inter-site references; they are discussed in Chapter 5. The chapter is organized as follows:

- Section 3.1 gives the rationale for partitions and describes their structure.
- Section 3.2 describes the basic scheme for partitioned collection.
- Section 3.3 describes how inter-partition information is maintained.
- Section 3.4 describes techniques for tracing a partition in the presence of transactions.
- Section 3.5 presents a performance study to evaluate some of the techniques.

3.1 Partitions

A server partitions its disk such that a partition can be traced independently within the available main memory. These partitions are for the purpose of separate tracing only; they are different from disk partitions for file systems and they are mostly invisible to the rest of the system.

3.1.1 Why Partitions

Partitioned tracing provides several benefits over tracing the entire disk as a unit. First, the main memory available for tracing is a small fraction, often less than a hundredth, of the disk space. Therefore, a trace of the entire disk is likely to have poor locality of reference with respect to available memory. It would thrash on disk, *i.e.*, fetch and evict a disk page many times in order to scan different objects in the page [YNY94, AGF95]. On the other hand, a partition trace fetches the pages of one partition into memory and scans all live objects in them before fetching another partition. This saves disk bandwidth, which is a performance bottleneck in many object stores and is crucial for good performance of many applications [Ghe95].

Second, avoiding thrashing on disk also results in faster collection of garbage. However, faster collection through partitioned tracing is based on the assumption that most garbage is either local to a partition (*i.e.*, not reachable from outside the partition), or is reachable from garbage chains passing through few partitions. As described later in Section 3.2, collection of a garbage chain passing through many partitions might take a long time. Similarly, collecting inter-partition garbage cycles requires another technique, which might also take a long time, as described later Chapter 4. The assumption that most garbage is local or on short chains is likely to be valid if partitions are constructed such that inter-partition references are relatively infrequent.

Third, partitions provide an opportunity for faster and more efficient garbage collection because the collector can focus on partitions that are likely to contain the most garbage. This is similar to generational collectors which collect newer partitions more frequently [Ung84]. Although the age-based heuristics of generational collectors are not applicable to persistent stores [Bak93], other heuristics are available for selecting partitions [CWZ94].

Fourth, if the server crashes in the middle of a trace, the work done during the trace is wasted. The loss of work is insignificant when tracing a partition, but it might be significant when the disk is traced as a unit. (The trace of a 10 gigabyte disk might take a day to finish.) It is possible to conduct a trace such that it can be resumed after a crash, but this adds substantial complexity [KW93].

3.1.2 The Structure of Partitions

A partition may contain many pages, possibly non-adjacent. This approach has important advantages. First, the page size is chosen for efficient fetching and caching, while a partition is chosen to be an efficient unit of tracing. Second, it is possible to configure a partition by selecting pages so as to reduce inter-partition references without reclustering objects on disk. For example, a partition can represent the set of objects used by some application, and the size of the partition can be chosen to match the set. This thesis does not prescribe a heuristic for configuring partitions, but it assumes that some heuristic is used to find clusters of inter-connected objects and to reduce inter-partition references.

There is a tradeoff in selecting the size of a partition. Small partitions mean more inter-partition references, which has two disadvantages. First, the space and time overheads for maintaining and using inter-partition information are higher. Second, there are likely to be more inter-partition garbage chains and cycles, which take longer to collect. On the other hand, big partitions mean more memory space used by the collector, which reduces the memory available for the server cache. Further, if the partition does not fit in the available memory, the trace would thrash on disk [AGF95]. Therefore, partitions should fit in a fraction, say, a tenth, of main memory. Given current memory sizes, a partition may be a few megabytes to tens of megabytes. Since pages are about ten kilobytes, a partition may contain about a thousand pages, and a ten gigabyte disk may contain about a thousand partitions.

A reference identifies the page of the referenced object, but not its partition. Therefore, we use a *partition map* to map a page to its partition and a partition to its pages. The partition map is stable, but it is cached in memory for efficient access.

3.2 Basic Scheme

The basic scheme for partitioned collection works as follows. For each partition P, the server records the set of objects in P that are referenced from other partitions; this set is called the *inset* of P. For simplicity, persistent roots are treated as references from a fictitious partition. To collect garbage objects in P, the server traces objects from the inset of P, but does not follow references going out of P. Objects in P not visited by the trace are known to be garbage and are deleted. For example, in Figure 3.1, the server will trace P from a and f and will collect object d.

The insets of partitions must be updated to reflect the creation and removal of inter-partition references. When a reference to an object in partition Q is stored into an object in P, safety requires adding the reference to the inset of Q so that a trace of Q does not miss it. On the other hand, as

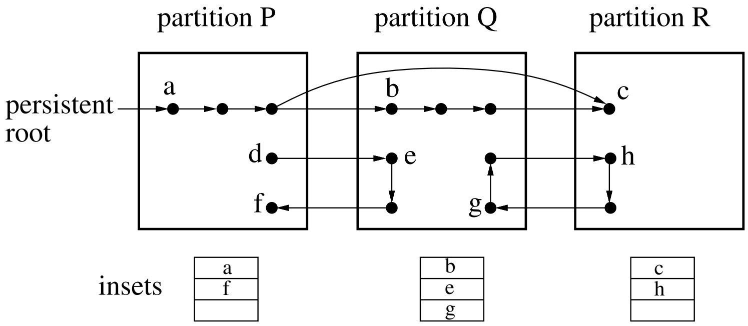

Figure 3.1: Insets.

inter-partition references are removed by applications and by the collector, an object in the inset of Q might not be referenced from other partitions anymore. Complete collection requires removing such references from insets, but this may be done lazily. A convenient time to detect and remove unnecessary references is when partitions are traced. A trace of partition P visits all references going out of P that are reachable from P's inset. Therefore, any reference e that is in the inset of another partition on account of P alone and that was not visited by the trace of P is removed. However, identifying such references requires additional information, which varies among existing schemes for partitioned or distributed collection, *e.g.*, remembered sets [Ung84, Sob88], reference counts [Bev87, Piq91], and reference source-listing [Bis77, SDP92, BEN$^+$93, ML94, AGF95, MMH96]. Section 3.3 presents a new form of reference source-listing that makes it efficient to both add and remove references in insets, while using less space than any previous scheme.

The above scheme successfully collects chains of garbage objects across multiple partitions. In the example in Figure 3.1, tracing P removes e from the inset of Q, so tracing Q the next time will collect e, and tracing P after that will collect f. In general, consider a garbage object f that is reachable from a chain of garbage objects through a sequence of partitions $\overline{P}$ at some point in time. Suppose the sequence in which partitions are selected for tracing after this time is $\overline{T}$. Then, f will be collected when $\overline{T}$ contains $\overline{P}$ as a subsequence. For example, in Figure 3.1, $\overline{P}$ for f is PQP, so one possible $\overline{T}$ that will collect f is $PQRP$. If a garbage object is reachable from multiple chains, then it will be collected when $\overline{T}$ contains all of them as subsequences.

Collection of garbage chains passing through many partitions might take a long time. Suppose a chain passes through m partitions and there are n total partitions. If the partitions are traced in round-robin order, the chain might be collected in m traces in the best case (when the round-robin order matches the chain order), and in $m \times n$ traces in the worst case (when the round-robin order is opposite to the chain order). Collection of garbage chains can be accelerated by using suitable heuristics for selecting partitions. It is desirable to select a partition that contains a lot of garbage or whose trace will lead to the collection of a lot of garbage in other partitions. We suggest the *reduced-inset heuristic* as one of the heuristics for selecting partitions:

> *Increase the trace-priority of any partition whose inset has reduced since the last time it was traced.*

This heuristic favors tracing partitions that contain the head-ends of garbage chains; therefore, it is likely to make quick progress in collecting garbage chains. Cook *et al.* have suggested some other heuristics for selecting partitions [CWZ94]. This thesis does not examine these heuristics any further; this issue is an important direction for future work.

3.3 Insets

This section describes new techniques for organizing and updating insets. The inset of a partition Q contains references to objects in Q that are persistent roots or are referenced from other partitions. For efficiency in adding and removing references in the inset, it is represented as a union of *translists* from other partitions to Q. A translist from partition P to Q contains the set of references in P to objects in Q; we call P the *source* and Q the *target* of the translist. Persistent roots in Q, if any, are stored in a translist from a fictitious source partition denoted as #. The inset of Q is implemented as a data structure containing pointers to the translists from other partitions to Q. In Figure 3.2, the inset of Q contains pointers to the translist from P to Q and the translist from R to Q.

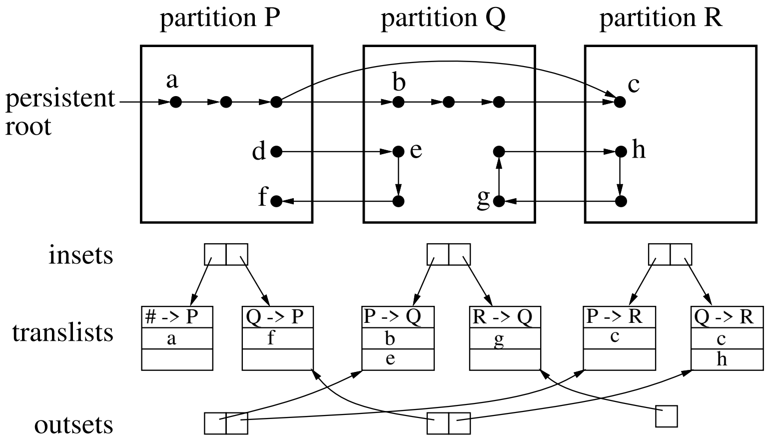

Figure 3.2: Inset, outset, and translists.

The set of references going out of a partition P is called the *outset* of P. It is implemented as a data structure containing pointers to the translists from P to other partitions. It is not necessary to record outsets for tracing partitions; however, the outset of a partition provides an efficient mechanism to remove unnecessary references in the insets of the target partitions. Specifically, when the partition is traced, the references in its outset that were not visited are removed, which by itself removes the references from the insets of the target partitions. For example, in Figure 3.2, when P is traced, e is removed from the outset of P, which removes e from the inset of Q. Arranging inter-partition references into translists and sharing them between insets and outsets provides a cheap mechanism to maintain both insets and outsets.

3.3.1 Comparison with Previous Work

Most other single-site partitioned collectors implement insets as a sequence of pairs ⟨reference, source-partition⟩ and do not provide mechanisms to find the outset [AGF95, MMH96]. The scheme by Amsaleg et al. scans the insets of all partitions after a trace to remove untraced references [AGF95]. This approach would not scale to a large number of partitions. The scheme by Moss et al. computes the outset whenever a page is fetched and also when a modified page is evicted, and applies the differences to insets [HM92, MMH96]. This approach would delay applications when they fetch pages.

Some generational collectors implement insets as *remembered sets* [Ung84, Sob88]. They record the *locations*—at the level of a word, object, or page—that may contain inter-partition references. The location information allows the collector to update references when an object is moved for compacting storage. However, this scheme is not suitable for disk-based heaps because tracing a partition requires examining and updating locations in other partitions. Further, storing locations at a fine granularity results in more information if multiple locations contain the same reference. Some other generational schemes use *card marking*, which remembers a single set of pages that might contain inter-partition references—undistinguished by the target partition [Sob88]. Any partition trace scans all of these pages for relevant roots.

On the other hand, many distributed collectors maintain both insets and outsets for inter-site references [SDP92, BEN+93, ML94]. However, insets and outsets at different sites do not share translists. The outset of a site contains the translists to other sites, while the inset either contains copies of the translists from other sites or it consists of a single set of incoming references with a count of the source sites for each reference [Bev87, Piq91]. If this organization were used for inter-partition references, it would have a higher space overhead and require safety invariants between insets and outsets. We experimented with such a technique earlier [ML96]; it was much more complex than the current technique due to the need to maintain invariants.

In Emerald distributed system, the inset stores only the set of incoming references with no additional information, making it impossible to determine when to remove references from it without doing a global trace [JJ92]. Such systems rely on a complementary global trace to collect inter-partition cycles as well as chains.

3.3.2 Maintaining Insets

Translists, insets, and outsets are kept on disk for two reasons. First, if this information is not persistent, recomputing it after a crash by scanning the entire disk would take a long time. Until the information is retrieved, no partition can be collected. Second, a large heap may have enough inter-partition references that recording them in memory would take up substantial space. For example, in a heap where every tenth word is a reference and every tenth reference points outside the containing partition, the inter-partition information would occupy a hundredth of the disk space. This space, although a small fraction of the disk, may be large compared to main memory.

Therefore, we maintain translists, insets, and outsets as persistent objects. Maintaining translists persistently requires care in keeping the disk-time utilization low, both for the garbage collector to perform well and, more importantly, to avoid degrading application performance. Translists are accessed for three methods:

1. Tracing a partition P needs to use references in the translists to P as roots.
2. After tracing a partition P, untraced references in the translists from P need to be removed.
3. New inter-partition references need to be added to the appropriate translists.

We reduce disk accesses for the first by clustering translists to the same partition together. Removing references from translists is described in Section 3.4. Below we describe techniques for reducing disk accesses when adding new references. This part is important because addition of new references is a steady-state activity—unlike tracing partitions, which can be scheduled occasionally. The significance of these techniques is evaluated in Section 3.5.

New inter-partition references are found lazily by scanning modified objects in the log. (Other garbage collectors have used the log to process modified objects lazily for various purposes [ONG93].) We refer to scanning objects for inter-partition references as *inter-scanning* to distinguish it from scanning objects as part of a trace. An object in the log must be inter-scanned before it is installed, since information about the modification is lost at that point.

Adding references to translists requires either caching them in main memory or reading them from disk. We save cache space and defer reading or writing the disk as follows. When a modified object in partition P is inter-scanned, a reference to another partition Q is recorded in an in-memory *delta list* from P to Q. Translists and delta lists maintain the following *inter-partition invariant* when the log is fully inter-scanned:

$$(\text{References from } P \text{ to } Q) \subseteq (\text{translist from } P \text{ to } Q \cup \text{delta list from } P \text{ to } Q)$$

The translist and the delta list from P to Q might contain some unnecessary references because deletion of references contained in P are not reflected into the translist or the delta list until P is traced. Further, references in a delta list might already be in the corresponding translist. Systems where old copies of modified objects are available can reduce such references in delta lists by not recording references already present in the old copies. However, a delta list might still contain references already in the translist because those references might be contained in other objects in the source partition.

Eventually, a delta list is merged into its translist. This happens either when it grows too big or when there is potential of losing its information in a crash. We discuss these possibilities below.

The total memory provided for delta lists is limited. When this space fills up, we select the largest delta list, fetch the corresponding translist, and merge the delta list into the translist. The sequence of events is depicted in Figure 3.3. If there is no corresponding translist, one is created and added to the appropriate inset and outset. Since translists to the same partition are clustered together, we merge all other delta lists to that partition at this point. Modifications to translists, insets, and outsets are handled like other objects; they are logged using an internal transaction.

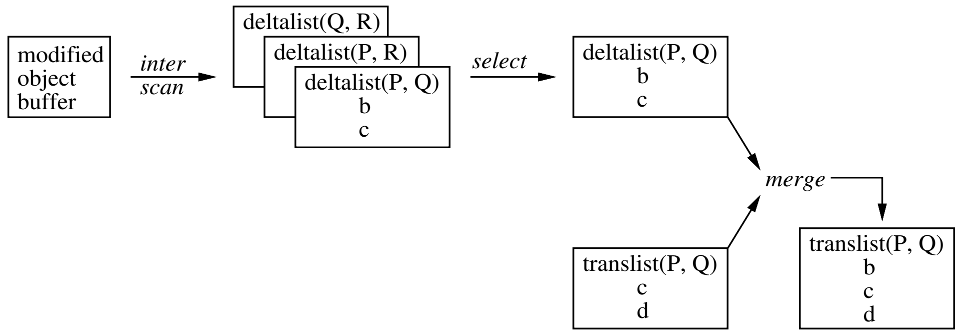

Figure 3.3: Delta lists.

If a server crashes, its delta lists are lost, but it can recover them by scanning the stable log. However, information in the log is truncated when modified objects are installed; therefore, information about inter-partition references in modified objects must be made stable by then. This is achieved by first finding the delta lists that might contain information from the installed objects. These delta lists can be made stable by either logging them or merging them into the translists. These actions need to be taken infrequently because the log is truncated only when the modified object buffer fills up.

The upshot of this scheme is that delta lists defer and batch disk accesses for updating translists. Delta lists may be viewed as a summary of inter-partition references in the log that are not yet added to the translists. The delta lists to a partition are treated as roots when tracing the partition. Thus, they avoid the need to *re*-inter-scan the entire log whenever a partition is traced.

3.4 Tracing a Partition

Most techniques described in this chapter can be used with a variety of tracing algorithms. We chose a mark-and-sweep scheme for tracing. A copying collector would use twice the space used by mark-and-sweep. Further, persistent stores often have little garbage to be collected [Bak93], so a copying collector would spend significant time copying many live objects. Finally, the use of slotted pages allows a mark-and-sweep scheme to compact live objects by sliding them within their pages, as discussed later in Section 3.4.4. Amsaleg et al. also chose a mark-and-sweep scheme for these reasons in the context of another object database [AGF95].

3.4.1 Safe Tracing

A trace that runs concurrently with ongoing transactions needs to see a safe view of the object graph such that it does not miss any references due to concurrent modifications.

> *A view is safe if it reflects the creation of all references up to some point in time, and does not reflect the deletion of any reference after that time.*

We refer to the point in time as the *snapshot time* of the trace. Objects that were created before the snapshot time and are not visited by the trace must be garbage because they were unreachable from the roots at snapshot time.

There are two basic ways of obtaining a safe view: *old view* and *new view*. The snapshot time of the old view is when the trace starts. An old-view trace scans an object at most once—the state of the object when the trace started—and it need not scan new objects created during the trace because they are created marked. Such a trace is also known as "snapshot at the beginning" [WLM92]. A new-view trace sees the most recent view of the heap. It must scan new and modified objects, possibly multiple times, such that when the last object is scanned, no further modifications are pending. Such tracing is also known as "incremental update" [WLM92].

An old-view trace is more conservative because garbage generated during the current trace is not collected until the next trace. While this might be too conservative in a server-wide trace, it is acceptable for partition tracing because a partition trace is short and because the garbage generated in a partition while it is being traced is a small fraction of the garbage generated in the whole system. Further, the old-view is simpler since it does not need to pursue modifications like the new-view trace does. Therefore, we use old-view tracing.

3.4.2 Safe Tracing in a Transactional System

Finding a safe view in a transactional system is difficult because modified objects might not be applied and installed in the order the modifying transactions are serialized. As described in Section 2.5, modified objects are processed as follows:

$$\boxed{\text{Prepare record}} \text{ —apply}\rightarrow \boxed{\text{modified buffer}} \text{ —install}\rightarrow \boxed{\text{disk page}}$$

An example of how applying or installing modifications out of order may result in erroneous collection is illustrated in Figure 3.4 Here, a persistent root a points to object b. Transaction τ_1 creates a reference from another persistent root c to b, and then τ_2 removes the reference from a to b. If modifications of τ_2 are applied or installed before those of τ_1, b might be collected. Note that this problem might arise in both optimistic and lock-based systems.

We find a safe view of the partition to trace, P, as follows. First, we load the pages of P in the memory; this is possible without flushing the server cache since a partition is a small fraction

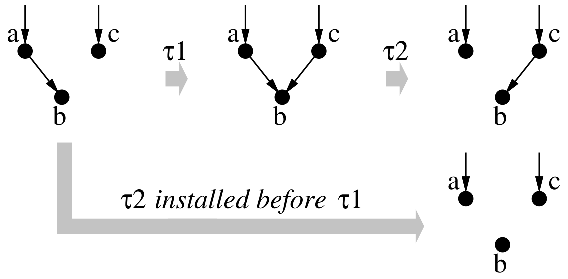

Figure 3.4: Problem with applying or installing modifications out of order.

of main memory. Then, to take a snapshot, the collector halts applying transactions and does the following work:

- Record the sequence number, L, of the last log record so far.
- Inter-scan all objects in prepare records up to L.
  This handles the case where c is in another partition and the modification to c is not applied.
- Inter-scan all objects in the modified object buffer.
  This handles the case where c is in another partition and the modification to c is not installed.
- Scan P's objects in prepare records up to L.
  This handles the case where c is in P and modification to c is not applied.
- Install P's objects in the modified object buffer into pages.
  This handles the case where c is in P and the modification to c is not installed.
- Halt installations to pages of P until the end of the trace, so that the trace will see the snapshot view of the partition.

After this work, the server may resume applying transactions, but installations to P are halted until the end of the trace. In order to reduce the time for which applying transactions is halted, we *preprocess* the log as much as possible beforehand and then finish the work atomically by halting applying transactions briefly. The preprocessing involves inter-scanning objects in prepare records and the buffer. (Of course, objects in the log that were inter-scanned before a previous trace need not be inter-scanned again.) The preprocessing also scans P's objects in prepare records and installs P's objects from the buffer into the in-memory pages. The objects in the log are indexed by the page they are in, so it is efficient to find objects belonging to P.

After taking a snapshot of the partition, the collector initializes the root set for tracing P. This set consists of the translists in the inset of P and any delta list to P. (This is a good opportunity to merge delta lists to P into the translists.) Inter-scanning the prepare records and the modified buffer ensure that all inter-partition references to P created before the snapshot are found.

3.4.3 Marking

Marking may be done using any graph traversal algorithm. It uses a *scan-set* to store references to objects that need to be scanned; the scan-set may be a queue or a stack to implement breadth-first or depth-first traversal. It also uses a *markmap* containing a mark bit per potential object name in the pages of the partition. Initially, only the roots are marked and entered in the scan-set. When scanning an object, if the collector finds an *intra*-partition reference b, it checks whether b is marked; if not, it marks b and enters it in the scan-set. If the collector finds an *inter*-partition reference b, it adds b to a new version of the appropriate translist in memory. The trace runs until the scan-set is empty. We give the marking algorithm below as a base line for modifications in later chapters.

```
while (scanset ≠ empty) Scan(scanset.remove())

proc Scan(object b)
      for each reference c in b
          if c ∈ P
              if not c.mark
                  c.mark := true
                  scanset.add(c)
          else (c ∈ Q)
              add c to the new translist from P to Q
      endfor
endproc
```

Objects are scanned at most once, except objects in prepare records that are scanned when taking the snapshot. If the trace reaches such an object later, the installed version of the object must be scanned again because prepared transactions may abort. Therefore, when objects in prepare records are scanned, they are not marked.

While P is being traced, clients may commit modifications to objects in P or create new objects in P. These modifications and creations are applied to the modified object buffer, but they are not installed into the pages of P, so the trace sees the snapshot view of the partition. If a client fetches a page of P in the meanwhile, the server sends an up-to-date copy of the page by installing any modifications from the buffer into the copy sent. Thus, commits and fetches to P are not blocked by the trace.

3.4.4 Sweeping

After marking is over, the collector sweeps the pages in the partition one at a time. All unmarked objects in the page are known to be garbage. If there is no such object, no work is needed. Otherwise, the collector slides all marked objects to one end of the page. It also updates the page's offset table to point to the new locations of the objects. This is illustrated in Figure 3.5. Sliding objects within the page retains the names by which they are referenced because they retain their indexes in the offset table. Object indexes of unmarked objects are flagged as free, so these indexes may be allocated to new objects. When the collector is done sweeping a page, it resumes installations of modified and new objects into the page. After such modifications have been installed, the page is flushed to the disk.

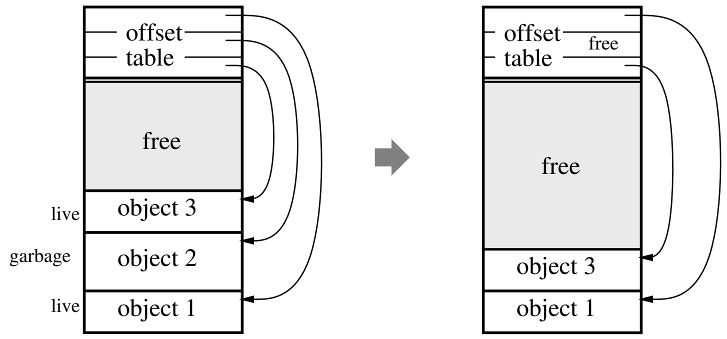

Figure 3.5: Compaction by sliding live objects within their page.

If the collector compacted storage by moving objects such that their names changed, it would be necessary to update references to these objects—including those that are stored in other partitions and in the clients. These references could be updated by recording locations containing inter-partition references, as in some generational collectors. However, such mechanisms add substantial cost and complexity in a disk-based heap. Further, reclustering of objects across pages should not be bound to the traversals of the collector. Such reclustering should be open to independent policies and application hints.

After sweeping the pages, the collector updates the translists in the outset of P. The old versions of these translists are replaced with their new versions that were generated while tracing P. Note that the old versions need not be fetched in from disk. If there is no new version of a translist, the translist is removed from the containing inset and outset. Updated values of translists, insets and outsets are logged as regular objects. Further, the collector deletes the delta lists from P, since they are accounted for in the the new versions of the translists.

3.4.5 Disk Accesses

The disk accesses involved in tracing a partition P are summarized below:

1. Fetch inset of P and translists in the inset of P for the roots.
2. Fetch pages of P.
3. Flush pages of P that were modified by the collector.
4. Fetch outset of P.
5. Log updates to the translists in the outset of P. If a translist is deleted, update the containing inset and outset.

If the server crashes before the collector logs new versions of the translists from P, the new versions are lost. This is acceptable because the old versions of the translists are retained and the corresponding delta lists are regenerated from the stable log; these translists and the delta lists satisfy the inter-partition invariant upon recovery from a crash. Similarly, if the server crashes before pages swept by the collector are flushed, the old versions of the pages are retained. Thus, a crash might only waste the work done during a trace. The wasted work is small because a trace is limited to a partition.

3.5 Performance

This section provides some details of the implementation and then presents some experiments to evaluate our technique for maintaining insets. The performance of maintaining insets is particularly important because it is a steady-state activity that must be carried out as objects are modified—unlike tracing partitions, which can be scheduled occasionally.

3.5.1 Implementation Details

The techniques described in this chapter have been implemented in Thor, except those for crash recovery. Work related to garbage collection is performed by a collector thread, which is run at low priority to avoid delaying client requests. Besides tracing partitions, the collector inter-scans modified objects in the buffer every so often.

Insets, outsets, and translists are implemented as linked lists of fixed-sized *block* objects. A translist block stores a compact array of references in no particular order. An inset (outset) block stores references to the contained translists paired with the source (target) partition ids. Blocks allows these lists to grow or shrink without copying; we use relatively small blocks (64 bytes) to reduce fragmentation. When updating a list, we log only the modified and new blocks. These blocks are logged using transactions, except that we bypass the concurrency control mechanism since only the collector accesses them. One problem with using blocks is that the blocks of a list may be mixed with those of other lists and should be consolidated periodically.

A delta list is implemented as a hash table. To merge a delta list into a translist efficiently, we iterate through the translist: for every reference x in the translist, we remove x from the delta list, if present. At the end, we append remaining references in the delta list into the translist. We have not yet implemented crash recovery and the actions needed on truncating the stable log described in Section 3.3.2. In our experiments, however, we accounted for the expected log overhead from these actions; they were found to be negligible.

3.5.2 Workload

Amsaleg *et al.* pointed out the lack of a standard benchmark for database garbage collectors [AFFS95]; such a benchmark remains absent today. Therefore, we designed a micro-benchmark specifically for evaluating the overhead of maintaining insets.

The benchmark database consists of a homogenous collection of small objects, each of which has a single reference and some data fields. This is similar to the benchmark suggested by Amsaleg et al., except that the objects are not linked into a list as in their case. Instead, the benchmark allows us to control the distribution of reference modifications systematically: both the spatial locality of references, *i.e.*, where they point, and the temporal locality, *i.e.*, which references are modified in time order.

The workload consists of selecting a *cluster* of objects and initializing their references; a cluster comprises a fixed number of contiguous objects in the database. Clusters are selected from the database randomly until the database is fully initialized. Random selection of clusters simulates the effect of concurrent applications creating or modifying groups of clustered objects. The cluster size is a measure of temporal locality in creating references. Bigger clusters mean that successive reference creations are more likely to happen in the same partition.

Spatial locality is determined by how the target object of a reference is chosen. Objects are numbered sequentially, and object n refers to a random object $n + i$ using a chosen probability distribution for i. (Object numbers wrap around when they overflow or underflow the bounds of the database.) The database is partitioned linearly; each partition contains p objects.

We used the following distribution of references. With some probability s, a reference points within the containing page. With probability $1 - s$, a reference points to an object that is i apart according to the exponential distribution, which is illustrated in Figure 3.6:

$$prob(i) = \frac{1}{2d} e^{-\frac{|i|}{d}}, -\infty < i < \infty$$

We chose the exponential distribution because it is simple to analyze and involves only one variable, d. We call d the *deviation*. While s is the major factor governing the number of inter-partition references, d governs their spread: a small deviation keeps them to nearby partitions, while a large deviation spreads them further. All experiments reported here used a deviation equal to the number of objects in a partition, which resulted in each partition containing references to its 16 neighbors on the average. Moreover, given this deviation, a fraction $1/e$ of references under the exponential distribution fall in the same partition. Thus, the overall *cross fraction* of references f that cross partitions is $(1 - s)(1 - 1/e)$. Figure 3.7 summarizes the parameters employed for the workload.

In our database of 256 Mbyte, a cross fraction of 10% leads to about 4 Mbyte of translists, insets, and outsets. The lists represent 1.5% overhead with respect to the database size, which is a

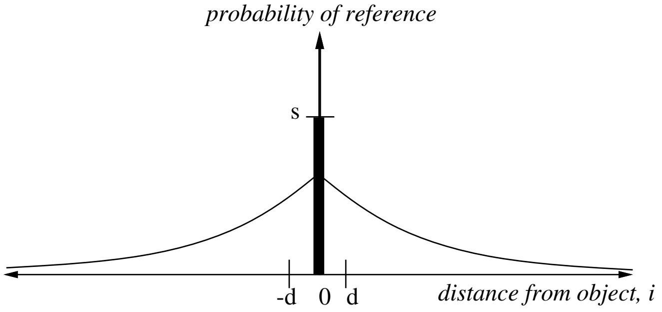

Figure 3.6: Probability distribution for references to nearby objects.

Parameter	Value(s) [Default]
Object size	30 bytes
Page size	32 Kbyte, 1 K objects
Partition size	1 Mbyte, 32 K objects
Database size	256 Mbyte, 8 M objects
Cluster size	32–8192 [1024] objects
Cross fraction f	0–15% [7.5%]
Deviation d	32 K objects

Figure 3.7: Workload parameters.

small overhead on disk. However, for realistic database sizes, the amount of this information would be significant compared to primary memory.

3.5.3 Experimental Configuration

We designed experiments to compare the *relative* performance of our scheme with and without delta lists for a range of workload parameters. We refer to the scheme with delta lists as DELTA, and that without delta lists as NODELTA.

The experiments ran on a DEC Alpha 3000/400, 133 MHz, workstation running DEC/OSF1. The database disk has a bandwidth of 3.3 Mbyte/s and an average access latency of 15 ms. The log is not stored on disk; however, we model it as a separate disk with a bandwidth of 5 Mbyte/s and average rotational latency of 5 ms; we ignore its seek time because the log is written sequentially.

To compute the correct overhead of maintaining insets, care is needed so that the collector's work is not hidden in idle periods such as disk accesses due to application fetches and commits. We ensured this by avoiding an external application and generating work for the collector within the server.

The collector is given a fixed amount of main memory to store lists. It is also given space in the modified buffer for storing modified lists. In our experiments, we allocated 1 Mbyte each for the list memory and the modified object buffer for storing lists.

In NODELTA, the collector uses the list memory to cache translists, insets and outsets. It also uses a small delta list (1 Kbyte) for efficient processing when adding references to translists.

In DELTA, the collector uses the allocated space to store delta lists and to cache translists. We determined how to best divide this space between the two experimentally. Figure 3.8 shows the variation in collector overhead as delta lists are allocated a bigger fraction of available memory. (Here, the cross fraction was fixed at 7.5%.) The overhead is the lowest when three quarters of the space is allocated to the delta lists, so we used this division in our experiments.

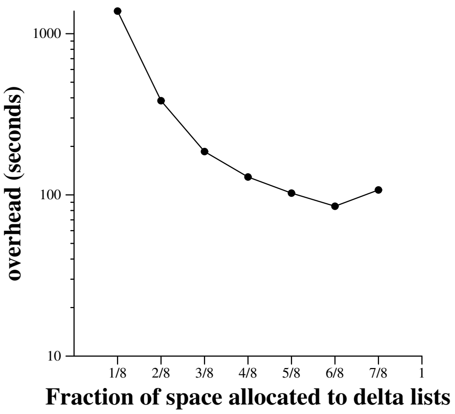

Figure 3.8: Effect of space allocated to delta lists. (Note log scale.)

3.5.4 Performance Results

This section presents the results of running the benchmark. We show how the collector overhead varies with the cross fraction, the cluster size, and the amount of list memory available. For each of these parameters, we compare the performance of DELTA and NODELTA.

Fraction of References Crossing Partitions

Figure 3.9 shows the results of running the benchmark in DELTA and NODELTA as the fraction of references crossing partitions is increased. The processor overhead comprises scanning overhead, which is a constant cost due to scanning objects, and list processing, which is mostly due to manipulating various lists in memory. The disk reads are due to fetching pages containing translist and inset/outset blocks, and the disk writes are due to installing modified blocks on the disk. The log forces are due to committing transactions containing modified blocks; the log overhead is too small to be visible in the figure. Most of the overhead is due to processing and disk reads.

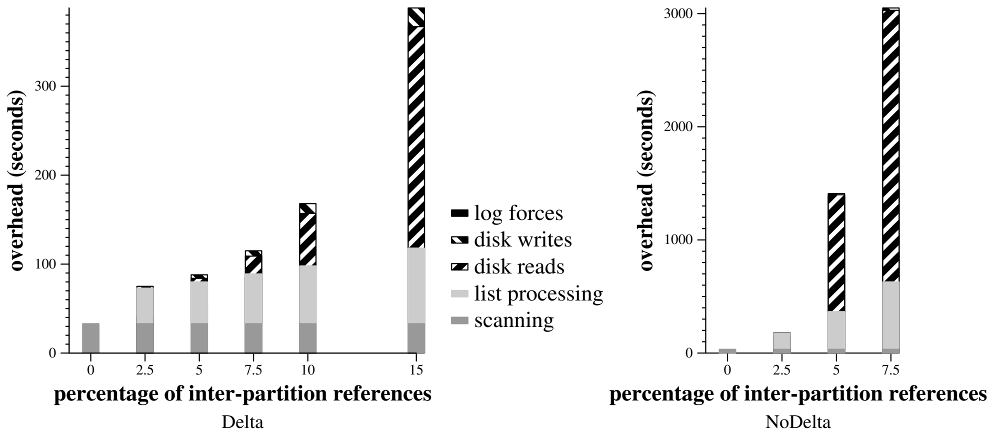

Figure 3.9: Breakdown of collector overhead with increasing inter-partition references.
Note the different scales in the two graphs.

Both the processor and the disk overhead of DELTA is much lower than that of NODELTA. The processor overhead of DELTA is lower because delta lists provide an efficient mechanism for adding a set of references to translists. However, we discount this advantage because other mechanisms could be used in NODELTA to improve processing time, such as B-trees. Figure 3.10 segregates the disk read overhead and compares this overhead in DELTA and NODELTA. When only 2.5% of the references cross partitions, the translists occupy about 1 Mbyte, which fits in the memory allocated to the collector. Therefore, both DELTA and NODELTA have negligible disk overheads. However, when the fraction of inter-partition references is a little higher, such that the translists do not fit in the allocated memory, NODELTA begins to thrash on the disk. At 7.5% cross fraction, when the translists occupy about 3 Mbyte, the disk read overhead for NODELTA is higher than that of DELTA by a factor of more than 100.

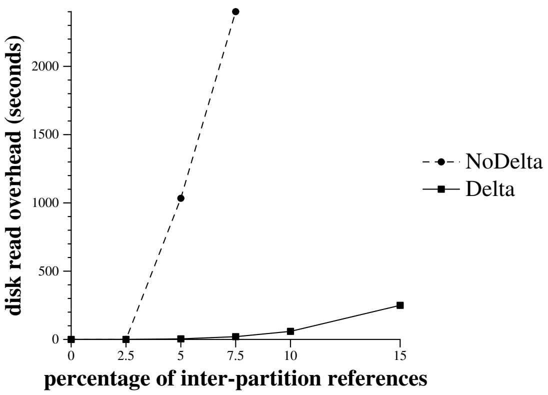

Figure 3.10: Effect of spatial locality on disk reads.

To put the collector's disk overhead in perspective of the remainder of the system, consider the following. Installing the database objects modified by the workload takes less than 400 seconds of disk time. This is the basic disk overhead in the absence of the collector. Thus, the collector's disk overhead in NODELTA can be significant compared to the basic overhead, while that of DELTA is much less.

Cluster Size

Figure 3.11 shows the overheads for DELTA and NODELTA as the cluster size varies from 32 objects to 8K objects. The cluster size controls the temporal locality in creating references. Smaller clusters mean that successive reference creations are more likely to happen in different partitions. Therefore, small clusters cause NODELTA to fetch and evict translists more frequently, increasing its disk overhead, while DELTA retains good performance. In the results shown, the fraction of inter-partition references was fixed at 7.5%.

In fact, DELTA has lower overhead for a cluster size of 32 objects than for 1K objects. This is because a low cluster size results in a large number of small delta lists. Since we merge delta lists to the same partition at the same time, a lower cluster size provides greater opportunity to create translists to the same partition close together. This results in fewer disk reads on later merges. For very small cluster sizes, this advantage seems to outweigh the disadvantage from poor temporal locality. The results indicate that good placement of translists is important to performance.

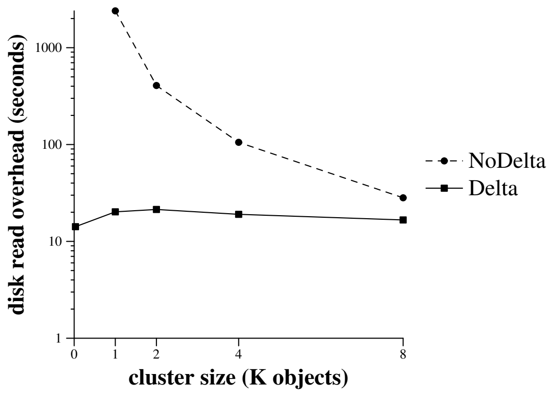

Figure 3.11: Effect of temporal locality on disk reads. (Note log scale).

Memory Allocated to the Collector

Figure 3.12 shows how the performance varies with the amount of memory allocated to the collector. The fraction of inter-partition references was fixed at 7.5%. As more memory is allocated to hold lists, the difference in the performance of DELTA and NODELTA is reduced. When the collector is given enough space to store all translists in memory (3 Mbyte), NODELTA has lower overhead than DELTA by a small margin. This is because DELTA allocates some of the space for delta lists and therefore has a smaller translist cache. However, when the collector is given enough space, the disk overhead of maintaining translists is negligible (about 1 second for both DELTA and NODELTA) compared to the disk overhead of installing the modified database objects (400 seconds). Therefore, a small difference in the disk overhead of maintaining translists is irrelevant.

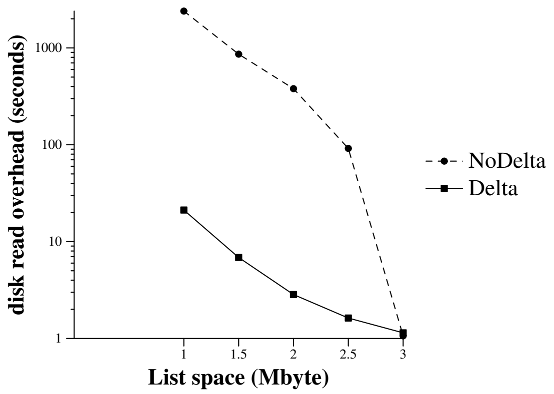

Figure 3.12: Effect of list memory on disk reads. (Note log scale.)

3.6 Summary

A server's disk space is divided into partitions that are traced independently. A partition consists of a set of pages, possibly non-adjacent, that fit in a small fraction of main memory. Partition tracing avoids thrashing the disk and provides timely garbage collection.

When tracing a partition, references from other partitions are treated as roots. References from a partition P to Q are recorded in a translist from P to Q. The inset of a partition records all translists to it and its outset records all translists from it. The outset provides an efficient mechanism to remove unnecessary inter-partition references in the insets of target partitions. Sharing translists between insets and outsets saves space and avoids the need to maintain invariants between them.

Translists are stored on disk for recoverability and to save memory space. New inter-partition references are found by scanning the log of modified objects lazily. They are stored in in-memory delta lists to avoid fetching translists. When delta lists grow too big or when modifications are removed from the log, some delta lists are merged into the translists. The result is that disk accesses are deferred and batched.

The performance advantage of these techniques was evaluated using a workload with controlled spatial and temporal locality of reference modifications. The study shows that inter-partition information may use a significant amount of space; if less space is available to cache the information, the disk overhead for updating it can be significant compared to the disk time for installing database objects. Delta lists reduce this overhead by a factor of 100.

A partition is traced using a mark-and-sweep scheme. The trace runs concurrently with transactions by taking a snapshot of the partition at the beginning. Such a trace does not collect the garbage generated during the trace, but this is acceptable in partitioned collection because a trace is short. Taking a snapshot is difficult because modifications might not be applied or installed in serialization order. The collector takes a safe snapshot by inter-scanning objects in prepare records and the modified object buffer. It halts installations to the partition until the trace is over, but this does not block fetches and commits to the partition. After the trace, the collector replaces the translists in the outset of the partition with their new, trimmer, versions. Also, pages in the traced partition are compacted by sliding live objects within their pages. The use of slotted pages avoids the need to update references to objects that are moved.

Chapter 4

Inter-Partition Garbage Cycles

Treating inter-partition references as roots when tracing partitions fails to collect inter-partition garbage cycles. This chapter presents a new site-level marking scheme for collecting such garbage: the scheme propagates marks from site-level roots to reachable objects within the site, and collects unmarked objects. We refer to the scheme as *site-marking* to distinguish it from the marking scheme for tracing each partition, which we refer to as *partition-marking*. Site-marking collects garbage cycles within a site but not between sites. Inter-site cycles are discussed in Chapter 6.

The roots for site-marking are the *site's inset*: the persistent roots in the site and the references coming in from other sites. The propagation of site-marks is piggybacked on partition-marking so that it has little disk overhead. When a partition P is traced, it propagates site-marks within P as much as possible, but it may generate more site-marks in other partitions that need to be propagated. Site-marking terminates when site-marks are known to have propagated fully in all partitions. A complete *phase* of site-marking may involve many partition traces, possibly multiple traces of each partition. After site-marking is complete, objects not site-marked are collected. This collects inter-partition garbage cycles because they are not reachable from the site-level roots.

Similar marking schemes have been proposed earlier in the context of distributed garbage collection to collect cyclic garbage. However, previous schemes either propagate marks separately from partition-marks [JJ92], or delay the collection of non-cyclic garbage [Hug85], or are not guaranteed to terminate correctly in the presence of concurrent mutations [LQP92]. Our scheme for site-marking has the following desirable properties:

- Site-marking is piggybacked on partition-marking so that it has little overhead.
- Site-marking does not delay the collection of non-cyclic garbage. In particular, it does not retain an object that would otherwise be collected by partition-marking.
- Site-marking is safe and it completes despite concurrent modifications.

Additionally, we organize the information required for site-marking to use the disk efficiently.

4.1 Data Structures and Invariants

Each partition has a *site-markmap*, which contains a site-mark per object. We say that an object is site-marked if it is marked in the site-markmap, and a reference is site-marked if the referenced object is site-marked. The site-markmap is implemented as a set of bitmaps, one per page in the partition; each bitmap contains a bit per potential object name in the page. In Thor, a 32 Kbyte page provides a name space for 2K objects; thus each bitmap is 2K bits, which represents a 0.8% overhead. Nevertheless, the aggregate size of the site-markmaps is large, *e.g.*, 80 Mbytes for a 10 Gbyte database. Therefore, we store site-markmaps on disk.

To determine when site-marking has finished, we associate a *flag* with each partition, which indicates whether site-marks have been propagated within the partition as much as possible. Specifically, the following *site-marking invariant* holds:

> *If partition P is flagged, all objects referenced from site-marked objects in P are also site-marked.*

The invariant is true of both inter- and intra-partition references contained in site-marked objects. Tracing a partition flags it, but might unflag other partitions. A marking phase terminates when all partitions are flagged. We give the precise conditions for termination in Section 4.5. We use the following rules to guarantee termination in the presence of concurrent modifications:

Rule 1. Once an object is site-marked, it is not unmarked until the end of the phase.
Rule 2. New objects are created site-marked.
Rule 3. Whenever a partition is unflagged, at least one of its unmarked objects is site-marked.

(In this chapter, "unmarked" means "not site-marked.") A site-mark bit is required for each object expressly to guarantee termination by enforcing the rules above. Otherwise, site-mark bits for just the inter-partition references would suffice, as in [Hug85, LQP92].

4.2 Starting a Phase

When a phase starts, only references in the site's inset are site-marked. The partitions containing these references are unflagged and the rest are flagged. (This satisfies the site-marking invariant.) These actions are performed incrementally as follows.

A site-level *phase counter* is incremented with each phase. In addition, a local phase counter for each partition tells the phase during which the partition was last traced. Before a partition is traced, if its local counter is one less than the site-level counter, this must be its first trace in the current phase. In this case, objects that were not site-marked in the previous phase are known to be garbage and are deleted. (If the partition's local counter is even smaller, it was not visited during the previous phase at all; so the whole partition must be garbage and can be deleted.) Then, the site-marks of all remaining objects in the partition are cleared, while those of non-existing objects are set in preparation for their possible creation in the future (Rule 2).

4.3 Processing Modifications

As applications commit new references, the site-marking invariant is preserved lazily by processing modified objects in the log; this processing is merged with inter-scanning modified objects as described in Section 3.3. A reference b contained in a modified object a ($a \rightarrow b$) is processed as follows. Suppose a is in partition P and b is in partition Q, where P and Q may be the same partition. If P is unflagged, we ignore the reference since it cannot break the invariant. If P is flagged and if a is site-marked, we have the following options to preserve the invariant:

1. Unmark a, but this would violate Rule 1.
2. Unflag P, but this would violate Rule 3.
3. Mark b.

Thus, the only viable option is to mark b. However, there are two problems. First, checking if a is site-marked requires fetching the site-markmap of P. We avoid fetching this site-markmap by site-marking b regardless of whether a is site-marked. The second problem is fetching the site-markmap of Q to update the site-mark of b. We avoid fetching this site-markmap by adding b to an in-memory *delta markmap*, which stores updates to the site-markmap. A delta markmap defers and batches disk accesses for the site-markmap like a delta list does for the translist (see Section 3.3). We say that an object or a reference is delta-marked if it is marked in the delta markmap. Delta markmaps support the following site-marking invariant:

> *If partition P is flagged, all objects referenced from site-marked objects in P are either site-marked or delta-marked.*

This invariant also avoids the need to site-mark the objects referenced in b transitively. If b is only delta-marked and not site-marked, references in b need not be site-marked.

A delta markmap is merged into the corresponding site-markmap eventually. This happens when either the aggregate size of delta markmaps grows above a certain limit, or when the log is truncated, as with delta lists in Section 3.3. Before merging, the local phase counter of the partition is checked. If the counter is one less than the site phase counter, the site-markmap must be from the previous phase; therefore, the steps indicated in Section 4.2 are taken to delete garbage objects and reinitialize the site-markmap.

If merging the delta markmap into the site-markmap causes a previously unmarked object to be site-marked, we unflag its partition to preserve the site-marking invariant. Note that unflagging the partition in this manner satisfies Rule 3.

4.4 Propagating Site-Marks

Site-marking is piggybacked on partition-marking. Before tracing a partition P, the delta markmap for P, if any, is merged into its site-markmap. The roots for tracing P are the same as that for partition-marking, as described in Section 3.4. When the trace reaches a site-marked object b, all objects in P reachable from b are site-marked. Consider an object c reachable from b. If c has not been visited by the trace before, it is both site-marked and partition-marked. If c has already been partition-marked but is not site-marked, it must be site-marked and re-scanned in order to propagate site-marks further. If c is outside partition P, it is merely delta marked. After P is traced, it is flagged since references in all site-marked objects in it are known to be site-marked or delta-marked.

We give the algorithm for propagating partition-marks and site-marks below; differences from the basic marking algorithm given in Section 3.4 are marked in *slanted text*.

```
while (scanset ≠ empty) Scan(scanset.remove())
proc Scan(object b)
      for each reference c in b
          if c ∈ P
               if not c.partitionmark
                  c.partitionmark := true
                  c.sitemark := c.sitemark or b.sitemark
                  enter c in scanset
               else if b.sitemark and not c.sitemark // need to rescan c
                  c.sitemark := true
                  enter c in scanset
          else (c ∈ Q)
               add c to the new translist from P to Q
               if b.sitemark add c to the delta markmap of Q
      endfor
endproc
```

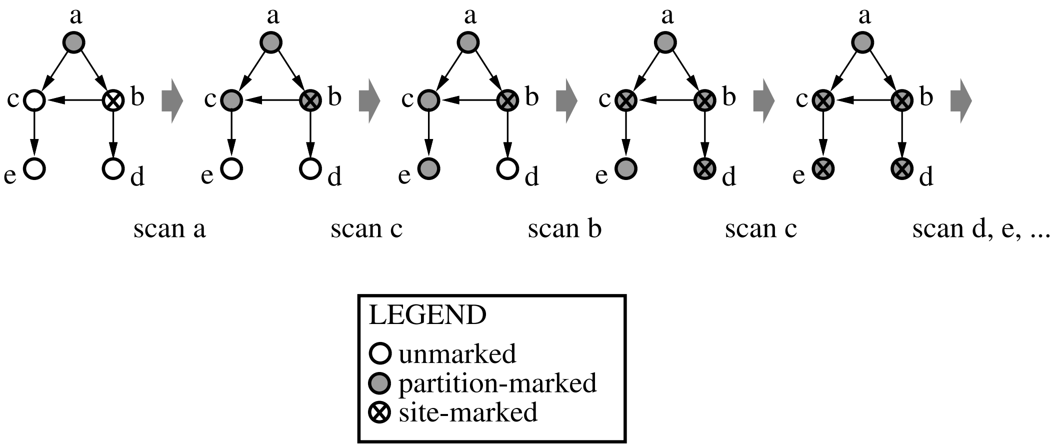

Figure 4.1: Propagation of partition-marks and site-marks during a trace.

Figure 4.1 illustrates the propagation of partition- and site-marks in a partition trace.

Note that site-marking does not retain any object that would otherwise be collected by partition-marking. This is true because site-marking does not cause any object to be partition-marked that would otherwise not be. Objects that are not partition-marked at the end of the trace are collected even if they are site-marked.

Site-marking may rescan some objects such as c in Figure 4.1. Fortunately, even in the worst case, each object will be rescanned at most once over an entire phase of site-marking. This is true because an object is rescanned only if it was not previously site-marked, and rescanning causes it to be site-marked. In practice, retracing would be even less frequent since some objects reachable from a site-marked object, such as d in Figure 4.1, are site-marked the first time they are visited. The number of retracings is reduced further by maintaining a separate scan-set for site-marked objects and selecting objects in this scan-set preferentially over the existing scan-set. While site-marking may increase the processor overhead of a trace by rescanning some objects, it does not increase the disk overhead of the trace: the partition's pages are fetched into memory for partition-marking anyway.

Some previous proposals piggybacked site-marking on partition-marking by dividing the trace into two parts [LQP92]. The first traced from site-marked roots and site-marked all objects it reached. The second traced from unmarked roots without site-marking any object. This two-part scheme does not rescan any object. However, it does not propagate site-marks from objects that are reachable only from unmarked roots, such as b in Figure 4.1. Therefore, the site-marking invariant would not hold after the trace. A possible action to preserve the invariant is to remove the site-mark of b, but that violates Rule 1 needed for termination. Another option is to include all site-marked objects as roots. However, this approach would retain all site-marked objects until the end of the site-marking phase—even when these objects would have been collected by partition-marking.

For concreteness, we present an example showing that a two-part tracing scheme without rescanning would fail to terminate safely. The example is illustrated in Figure 4.2. Here, a and d are persistent roots, and an inter-partition cycle is connected to one of them. Consider a pathological client that hides the cycle from the collector as follows. Before the collector traces partition P, the client creates the reference $d \to e$ and removes the reference $a \to b$; before the collector traces Q, the client creates the reference $a \to b$ and removes the reference $d \to e$. When tracing P, a and b are site-marked (b is site-marked because a reference to it is created from a), but b is not reachable from any site-marked root; therefore, the site-mark from b is not propagated any further. Similarly, when

Q is traced, the site-mark from e is not propagated any further. Thus, c and f are never site-marked.

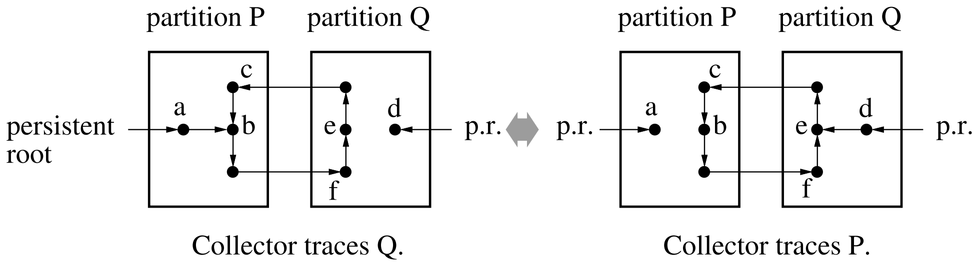

Figure 4.2: Example where a 2-part tracing scheme without rescanning fails.

4.5 Termination

Site-marking is complete when the following conditions hold:

1. All partitions are flagged.
2. All delta markmaps have been merged.
3. All references in the site's inset are site-marked.

Since modifications committed by clients may change the validity of these conditions, a phase is complete only when the conditions hold in a safe view of the site. A view is safe if it reflects the creation of all references up to some point in time, and does not reflect the deletion of any reference after that time. The following conditions provide such a view:

- All modified objects in the log have been processed up to a certain log record, L.
  So all references created until then are seen.
- Objects scanned during previous partition traces were installed from log records before L.
  So a reference deletion after L is not seen.
- No reference was removed from the site's inset after L was logged.
  So an inter-site reference deletion after L is not seen.

Such a view is available when a snapshot is taken before tracing a partition (see Section 3.4). The following policy is used to test for termination in the snapshot state. If all partitions are flagged, we merge all delta markmaps. Then we check references in the site's inset: if any is not site-marked, we site-mark it and unflag its partition. If all partitions are still flagged, marking is complete. Otherwise, we wait until all partitions are flagged again and repeat the procedure.

Safety

Here we show that, at the end of a site-marking phase, all live objects at the site are site-marked. Processing all modifications in the log up to the snapshot ensures that the site-marking invariant holds in the snapshot. Merging all delta markmaps ensures that there is no object that is delta-marked but not site-marked. Therefore, by the invariant, given that all partitions are flagged, all objects that are reachable from a site-marked object must also be site-marked. Further, all references in the site's inset are known to be site-marked. Therefore, all references reachable from them are site-marked.

Liveness

Site-marking is sure to terminate because a partition can be unflagged only a finite number of times during a phase. Every time a partition is unflagged, at least one of its unmarked objects is site-marked (Rule 3). Further, such an object must have been created before the current phase because objects created during the current phase are always marked (Rule 2). Finally, once an object is site-marked, it is not unmarked before the phase ends(Rule 1). Therefore, the number of times a partition can be unflagged is at most the number of objects in the partition that were created before this phase; there is only a finite number of such objects. Therefore termination is guaranteed even if applications are continually modifying objects or creating new ones.

Termination does require that an unflagged partition is traced and thereby flagged eventually. The relative frequency of tracing various partitions can still be governed largely by an independent policy. The following technique is used to ensure that unflagged partitions are traced eventually. When a partition is unflagged, it is inserted in a queue of partitions to be traced. When selecting a partition to trace, with probability p, the partition is selected from the queue. With probability $1 - p$, the partition is selected using the independent policy. The value of p may be controlled to balance the need for finishing site-marking (large p) and the need for using the independent policy (small p).

Although site-marking is guaranteed to terminate, the number of traces it might take in the worst case is pessimistically large—equal to the number of objects in the site at the beginning of the phase. The worst case happens when each partition trace site-marks only one object. In practice, a partition trace will propagate site-marks to many unmarked objects, so the expected length of a site-marking phase is likely to be much shorter. However, estimating the length of a phase in the presence of concurrent mutations is difficult. It is simpler to estimate this length by assuming that applications are quiescent, that is, not modifying objects. In this case, a partition is unflagged only as a result of tracing another partition. Suppose that there are n partitions and the maximum inter-partition *distance* of any object from an inset reference is l. The inter-partition distance of an object is the smallest number of inter-partition references in any path from an inset reference to the object. Another simplifying assumption is that partitions are selected for tracing uniformly, for example, in round-robin order. Then, marks will propagate fully in l rounds, or $n \times l$ partition traces. Note that this is the worst case bound given the round-robin order. (If unflagged partitions are traced preferentially over flagged partitions, fewer traces are needed.) With a thousand partitions and a maximum distance of ten, a site-marking phase would take ten thousand partition traces.

4.6 Crash Recovery

Since site-markmaps are maintained on disk to save main memory, it takes little more to make them recoverable after crashes. This allows a server to resume site-marking after a crash, which is desirable because site-marking takes relatively long to finish. The site-markmaps, flags, and phase counters of partitions are maintained as persistent objects. The site-wide phase counter is updated stably when a phase terminates. Similarly, the phase counters of partitions are updated stably when they are incremented.

A site-markmap is updated when a delta markmap is merged into it. A delta markmap must be merged into the site-markmap before any modified object that contributed to it is truncated from the log. The delta markmaps that must be merged are identified just like delta lists that must be merged, as described in Section 3.3. Further, after a partition P is traced, the delta markmaps updated during the trace are merged into their corresponding site-markmaps *before* P is flagged stably. This ensures that the site-marking invariant will hold for partition P upon recovery from a crash.

4.7 Related Work

Collection of inter-partition garbage cycles is related to collection of inter-site garbage cycles in distributed systems. We describe distributed schemes in Chapter 6. Here, we describe schemes used in single-site, partitioned systems.

Generational collectors collect inter-partition garbage cycles [Ung84]. They order the partitions, say $P_1, \ldots, P_n$ (often by age such that P_1 is the newest partition and P_n the oldest). A partition P_i is always traced along with partitions $P_1, \ldots, P_{i-1}$. Thus, a inter-partition garbage cycle that spans partitions $P_{i1}, \ldots, P_{im}$ is collected when the partition numbered $Max(i1, \ldots, im)$ is traced.

System PMOS collects inter-partition garbage cycles by grouping partitions into *trains* [HM92, MMH96]. Trains are ordered by age. When a partition is traced, objects that are reachable from a site-level root or a newer train are copied to newer trains, while other objects are copied to a new partition within the same train. Eventually, the partitions in the oldest train contain only cyclic garbage and can be discarded. As objects are copied between partitions, references to them in other partitions need to be fixed. Even if site-wide logical names were used to avoid name changes, inter-partition information would need to be updated continually as objects are copied. Further, the scheme must copy objects is one partition to multiple partitions in different trains based on where they are referenced from.

4.8 Summary

Inter-partition garbage cycles are collected by a site-level marking scheme whose roots are the persistent roots and references from other sites. Propagation of site-marks is piggybacked on partition traces. Tracing a partition propagates site-marks within the partition as much as possible, but it may generate more site-marks in other partitions that need to be propagated. A partition is flagged when the site-marks of its objects are fully propagated.

Propagating site-marks does not retain any object that would be collected otherwise; therefore, it does not delay the collection of non-cyclic garbage. Site-marking is complete when site-marks are fully propagated in all partitions. We ensure that site-marking completes despite concurrent modifications by not unmarking a site-marked object and by site-marking at least one object in a partition that is unflagged. Site-marking may require many partition traces to complete, but that is acceptable provided most garbage is collected by individual partition traces.

Site-marks of objects are stored in markmaps on disk to save main memory. Disk accesses for updating markmaps upon modifications are deferred and batched using in-memory delta markmaps. Delta markmaps are recoverable from the log, so site-marking can be resumed after a crash.

Chapter 5

Inter-Site References

A server traces its objects independently of other sites. For such a trace to be safe, the server must treat references from clients and other servers as roots. This chapter describes safe, efficient, and fault-tolerant management of inter-site references. It presents new techniques to account for client-caching, multi-server transactions, and client crashes. Clients might cache a large number of references to objects at servers, so conventional techniques for recording them would have a large space and time overhead. Further, multi-server transactions committed by clients may create a reference from one server to another without direct communication between the two, so conventional techniques designed for RPC-based systems would not work safely. Finally, clients may appear to have crashed when they are alive; conventional techniques would fail to prevent such clients from introducing dangling references in server objects. This chapter provides the solution to these and related problems:

- Section 5.1 describes the basic technique used to record inter-site references.
- Section 5.2 describes how references held in client caches are recorded efficiently.
- Section 5.3 describes how references between servers are recorded.
- Section 5.4 describes how server and client crashes are handled.

5.1 Basic Scheme

Inter-site references are recorded much like inter-partition references. The inset of a site T comprises the persistent roots in T and references from other sites. The inset is the union of translists from other sites to T. The translist from site S to T contains the references from S to T; we call S the source and T the target of the translist. For simplicity, persistent roots are treated as references from a fictitious source site.

A site treats another site as a single partition. This is desirable because identifying the exact partition of a reference to another server would require the partition map of that server. (A reference identifies only the server and the page it resides in, but not the partition.) Further, a server should be able to change its partitions dynamically without affecting other sites.

Updating a translist stored at the target site T requires messages from the source site S. To determine when to send these messages, S keeps a local copy of the translist from S to T. The copy of the translist at S is called the *outlist* of S to T, while that at T is called the *inlist* of T from S. The outlist at the source site helps avoid unnecessary messages, and the inlist at the target site provides local access to the roots. This organization is illustrated in Figure 5.1.

When a reference to an object b in T is stored into an object in S, S checks whether b is in its outlist to T and, if not, sends an *add* message to T. When T receives this message, it adds b

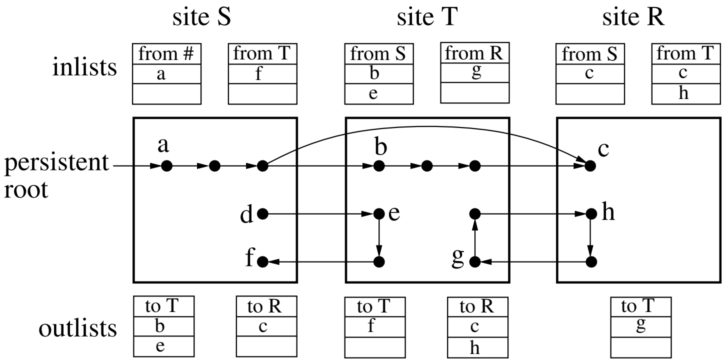

Figure 5.1: Recording inter-site references.

to its inlist from S. The add message must reach T in time such that a trace at T does not miss the reference. This marks a difference between recording inter-partition references within a site and inter-site references. Although partitions in a site are traced independently, they are traced by a single collector thread, and the collector traces a partition only after it has finished processing sufficient modified objects. Therefore, it is safe for the collector to process inter-partition references in modified objects lazily. On the other hand, different sites conduct local traces asynchronously, so inter-site references in modified objects must be processed eagerly. Nonetheless, there are various protocols for deferring or avoiding add messages without compromising safety and for tolerating lost messages [SDP92, BEN$^+$93, ML94]. Sections 5.2 and 5.3 describe new protocols to add client-to-server and server-to-server references.

When S does a local trace, it removes the references in its outlists that were not traced, and sends *remove* messages for them to their target sites. When a site receives such a message, it removes the indicated references from its inlist from S. Lost remove messages are tolerated by occasionally sending the entire outlist to the target site in an *update message*. When the target site receives the update message, it replaces its inlist with the outlist received. The message protocol maintains the following *inter-site invariant*:

References from S to $T \subseteq$ outlist of S to $T \subseteq$ inlist of T from S

This invariant is needed for the following purposes:

- References from S to $T \subseteq$ inlist of T from S:
  So that local traces at T are safe.
- Outlist of S to $T \subseteq$ inlist of T from S:
  So that S need not send an add message for a reference in its outlist.
- References from S to $T \subseteq$ outlist of S to T:
  So that S may send its outlist to update the inlist at T.

In the example in Figure 5.1, when S does its next local trace, it collects d, removes e from its outlist, and sends a remove message to T. T removes e from its inlist from S and collects e the next time it does a local trace. Thus, garbage objects that are not reachable from other sites are collected locally. (Since objects are clustered such that inter-site references are rare compared to local references, the bulk of the garbage is local.) Further, the collection of an inter-site chain of garbage objects involves only the sites containing the chain. We have formalized this feature as the *locality property*:

Collecting a garbage object involves only the sites from which it is reachable.

This property minimizes inter-site dependence: if a garbage object is reachable from some object at a site S, S must play a role in collecting the object anyway. The locality property results in fault tolerant and timely garbage collection. Local tracing does not, however, collect inter-site garbage cycles such as objects g and h; collection of such garbage is discussed in Chapter 6.

5.1.1 Inter-Site Reference Listing vs. Reference Counting

Some distributed systems use alternative forms of storing the inset, although most of them store outlists as described above. In systems using inter-site reference counting, the target site records a *count* of the source sites that hold a reference to a given local object—without recording the identities of the individual source sites [Bev87, Piq91]. When the count of a reference reduces to zero, it is removed from the inset. On the other hand, our scheme effectively records the identities of the source sites for each reference in the inset, a technique known as *reference listing* [Mah93a, BEN^{+}93]. Reference listing keeps more information than reference counting, but we use it for the following reasons:

1. Reference listing tolerates permanent site failures. If a client or a server X fails permanently, other servers need simply remove their inlists from X. With reference counting, other servers cannot figure out which counts to decrement without resorting to a global mechanism.
2. Reference listing tolerates message loss, which might happen when a site crashes (see Section 2.6). This is because add and remove messages are idempotent and can be safely sent again. Further, occasional update messages compensate for lost remove messages and conservatively sent add messages. On the other hand, increment and decrement messages for reference counting are not idempotent. Add/remove messages do require ordered delivery between a pair of sites, as do increment/decrement messages.
3. The space used by reference listing is less than twice of that used by reference counting. This is because sites must maintain outlists even with reference counting in order to detect when to send increment or decrement messages. Since inlists reflect outlists, the space used by reference inlists is roughly the same as that of outlists.

5.2 Client-to-Server References

A server must include references from client sites in its root set. This is because a persistent object might become unreachable from persistent roots while it is still reachable from handles at clients. For example, in Figure 5.2, the client has cached persistent objects a and b. It removes the reference from a to b and stores it into the volatile object d. When it commits the transaction, b becomes unreachable from persistent roots. If the server does a garbage collection at this point without considering incoming references from the client, it would collect b and c. This is in error because the client might try to access c and find it deleted, or it might make b reachable from a again. In the latter case, the client could send a copy of b to the server, but it would not be able to send c, which is lost for good.

A client may fetch a large amount of data from a server, potentially containing millions of references. Recording all such references in a translist will use a lot of space and time, and delay fetch requests [ML94]. Another problem with treating all references held by clients as roots is that it might prevent the collection of garbage objects: If the pages cached by a client happen to contain a bunch of garbage objects, the server will not collect that garbage. If one or other client caches the garbage in this manner all the time, the server will never collect it.

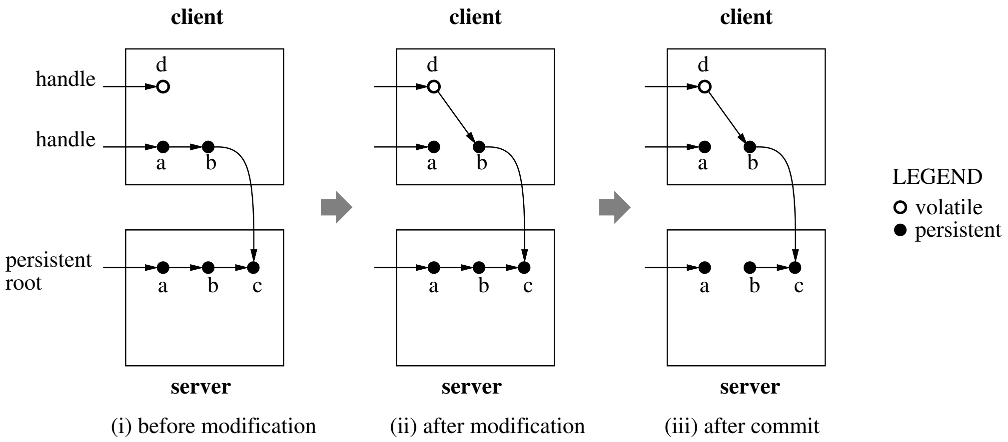

Figure 5.2: Need for including client references in the root set.

5.2.1 The Essential Translist

It is not essential to treat all client-held references as roots. If a client can reach a reference *c* only by reading a local copy of another persistent object *b*, the server need not include *c* in its root set because *c* is protected from collection by the reference from *b*. In effect, the server needs to account for only handles and the references stored in the volatile heap at the client. More specifically, the server need record only the following:

> The *essential translist* from a client to a server is the set of references to the server's objects that were reachable from the client's handles via zero or more volatile objects when the current transaction started at the client.

In Figure 5.2, before the client commits the transaction, the essential translist includes only a, and after the commit, it includes a and b. We reason below that the essential translist is the necessary and sufficient set that a server must include as roots.

A server must include the references in the essential translist from a client as roots—except references that are persistent roots. Consider any reference b in the essential translist from a client X. If b is not a persistent root, another client Y could commit a transaction that makes b unreachable from persistent roots. If the server does not include b in its root set, it would collect b. However, client X can access the reference to b without reading any persistent object, since this reference is reachable from a handle via only volatile objects. It can therefore commit a transaction that copies this reference into another persistent object, and the transaction will commit because it does not conflict with the transaction by client Y. However, committing the transaction would result in a dangling reference to b.

Now we show that it is sufficient to include the references in the essential translist as roots. If client X uses any other persistent reference c, it must have found c by reading a path of persistent objects, $b_0 \rightarrow \ldots \rightarrow b_n \rightarrow c$, all during the current transaction, such that b_0 is in the essential translist. There are two possibilities regarding the collection of c at the server. First, the copies of $b_0, \ldots, b_n$ read by X remain un-modified at the server until the transaction at X commits. In this case, c is protected from collection because b_0 is treated as a root and there is a path from b_0 to c. This condition is always true in systems using lock-based concurrency control since $b_0, \ldots, b_n$, are read-locked by client X.

The second possibility arises due to the optimistic concurrency control used in Thor. Another client Y might have committed a transaction that deleted the path $b_0 \rightarrow \ldots \rightarrow b_n \rightarrow c$ by modifying some object b_i ($0 \leq i \leq n$). The server might then collect c although client X may reach c. However, in this case, the transaction at client X will abort later—even if there were no garbage collection—because it read an old copy of b_i. In particular, if client X stored c in another object or created a handle to it, these effects will be undone when the transaction aborts. Further, if client X attempts to fetch object c from the server before committing the transaction, the server will raise a "not found" exception. (Assume for the moment that the name c has not been reassigned to a new object.) The exception indicates that the transaction is going to abort later, so the client aborts the transaction immediately. This has the benevolent effect of expediting impending aborts.

5.2.2 Name Reuse Problems

The second possibility discussed above raises additional problems if the names of collected objects may be reassigned to new objects. These problems arise because of incarnation mismatch between a reference and the copy of an object cached at a client. There are two such problems: the client has an old reference and a new copy of an object, or the client has a new reference and an old copy. Even though a transaction using mismatched incarnations will abort later, the danger is that the client might suffer a runtime type error because of the mismatch, which might result in unspecified behavior. One solution is to type-check an object when it is fetched into the client cache or when a reference is unswizzled [Mos92], and to abort the transaction on a mismatch. Since fetch-time type-checking is expensive in a system with subtyping, we provide alternative solutions to the two mismatch problems based on invalidation messages.

Old Reference and New Copy

The old-reference-new-copy problem might occur when a client X has reached a reference c by traversing $b_0 \to \ldots \to b_n \to c$, and another client Y disconnects c from persistent roots by modifying an object b_i in the path. The server may then collect c and reassign its name to a new object. Now client X might fetch the new incarnation of c while the reference $b_n \to c$ expects the old incarnation. If b_i and c belong to the same server, invalidation messages prevent the problem as follows. When b_i is modified by client Y, the server buffers an invalidation message for client X. Since messages from the server are delivered to the client in the order they were sent, client X will receive the invalidation for b_i before the response to the fetch request for object c. It will therefore abort the transaction before invoking any operation on the new copy of c. The invalidation message will also cause b_i to be evicted, so future transactions at client X will not access c along the old path.

However, if b_i and c reside in different servers, say S and T, then the invalidation from S might not reach the client before the client fetches c from T. This is illustrated in Figure 5.3

We present a conservative solution based on the fact that S must send a remove message to T before T may collect c. Until then, c is protected from collection by the inlist of T from S. Therefore, we employ a *remove barrier*:

> A server waits before sending a remove message until all invalidations sent for previously committed transactions have been acknowledged.

This barrier is implemented as follows. S assigns an increasing *commit id* to each transaction that commits at S. When S obtains a snapshot for a local partition trace, it records the highest commit id so far. After the trace, S does not remove unreachable references in its outlists immediately; instead, it stamps them with the recorded commit id. S removes such a reference only after it has received acknowledgements for all invalidations with commit ids less than or equal to the associated stamp. Clients that do not acknowledge an invalidation for a long time may be shut down as described in Section 5.4.

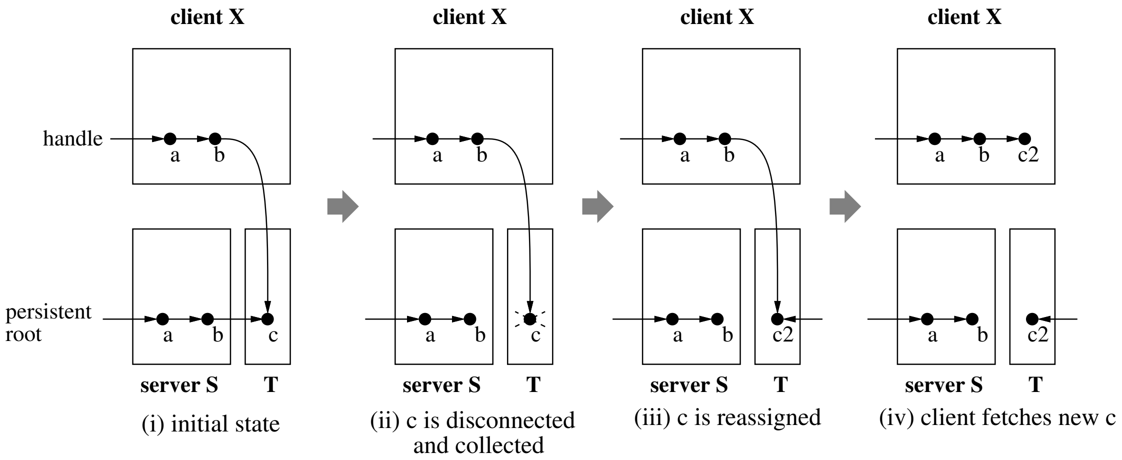

Figure 5.3: Problem of old reference and new copy in client cache.

New Reference and Old Copy

The new-reference-old-copy problem is illustrated in Figure 5.4. Here, client X has cached the old copy of *c*. Meanwhile, a modification to *b* by another client Y results in the collection of *c*. A further modification creates a new object with the same name, *c*, and stores a reference to it into object *d*. Now, client X fetches object *d*. The response to the fetch is preceded by an invalidation message for *b*. The invalidation causes X to evict *b* but does not abort the transaction because X did not read *b*. Further, X finds that *d* points to *c* and uses the cached copy of *c*.

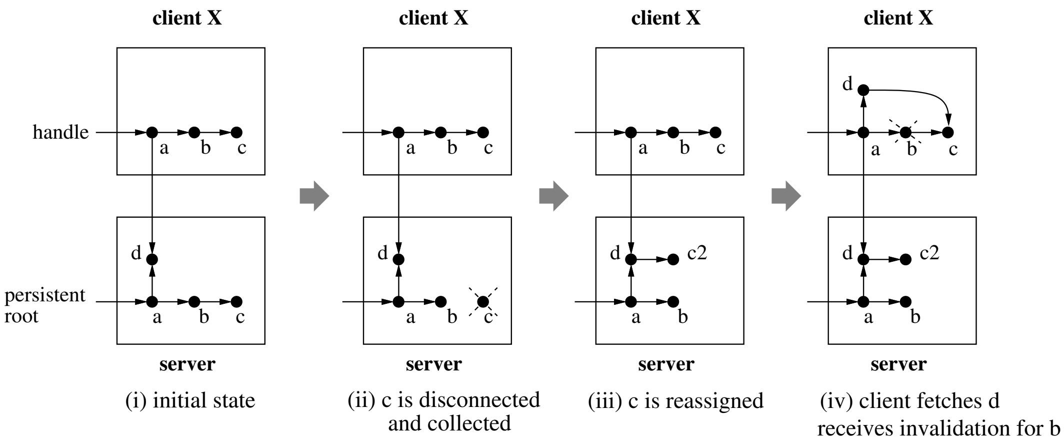

Figure 5.4: Problem of new reference and old copy in client cache.

This problem is solved by treating the deletion of an object *c* as a modification to *c*. Specifically, the server enters *c* in the invalid set of the clients that might have cached *c* and sends invalidation messages to them in the background. If *c* and *d* are on the same server then client X will receive an invalidation for *c* before it fetches the new copy of *d*. Client X will then evict the old copy of *c*.

However, if *c* is on server S and *d* on T, then the client might not receive the invalidation for *c* from S before it fetches *d* from T. A conservative solution here is to employ a *name-reuse barrier*:

> A server waits before reusing the name of a garbage object until all invalidations sent for that object have been acknowledged.

Aborts Due to Garbage Collection

Treating deletion as modification might cause extra aborts, though this is extremely unlikely. Suppose a persistent root a at server S refers to object b at server T. Client X commits a transaction τ_1 that has read a and b. Suppose, τ_1 is coordinated by another server U, as illustrated in Figure 5.5. Soon after server S prepares τ_1, another client Y commits a transaction τ_2 that removes the reference from a to b. Now, S might send a remove message for b to T, which might reach T before the prepare message for τ_1. Note that the remove barrier might not apply here because client X might have evicted a (see Section 2.3). On receiving the remove message for b, T would collect b and add b to the invalid set for client X. Therefore, when the prepare message for τ_1 reaches T, T would abort τ_1.

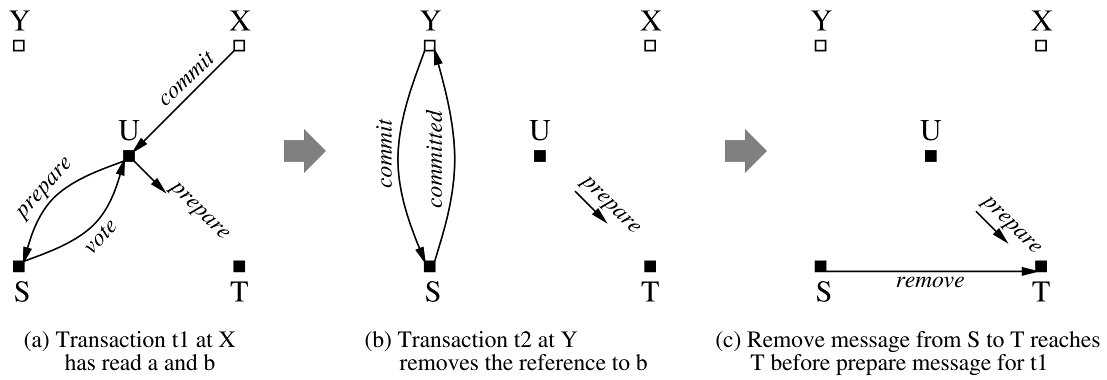

Figure 5.5: Possibility of abort due to garbage collection.

This scenario is extremely unlikely because it involves τ_1 and τ_2 to commit in close proximity and requires the prepare message to be delayed by a long period. Further, τ_1 would have aborted anyway if τ_2 reached S before τ_1; applications cannot distinguish between such aborts and those due to garbage collection.

5.2.3 Maintaining the Essential Translist

A client maintains an essential outlist for a server by adding references to it at commit and removing references from it when the volatile heap is garbage collected.

Adding References

When the client receives a commit request from the application, it computes the set of references that should be added to its essential outlists, called the *add set*:

1. Persistent objects that were assigned handles during the transaction.
2. References to persistent objects that were stored in volatile objects during the transaction.

Here, persistent objects include the objects that become persistent as a result of the transaction. To compute the second part of this set, the client scans the volatile objects modified by the transaction. The client need not scan volatile objects that become persistent due to the transaction.

The client removes the references in the add set that are already in its outlists and sends the remaining ones in the commit request; this effectively piggybacks add messages on the commit message. First, consider a single-server transaction. If the transaction is validated and committed, the server adds the references to its inlist from that client. When the client finds that the transaction has committed, it adds the references to its outlist, thus preserving the inter-site invariant.

The add protocol for a multi-server transaction is tricky because participants prepare the transaction at different times. Safe addition of references into inlists from clients is closely related to the addition of references into inlists from servers. A general solution that achieves both is discussed in Section 5.3.

Note that when clients fetch objects, no extra work is needed for garbage collection. This is desirable because fetching objects is expected to be the most common operation between clients and servers.

Removing References

The client trims its outlists when the volatile heap is garbage collected. The roots for tracing the volatile heap consist of the handles given to the application. When this trace reaches a reference to a persistent object, the reference is put in a new version of the appropriate outlist. After the trace, the client replaces the outlists with their new versions. Then it sends remove messages containing the references dropped from its outlists.

Tracing the Volatile Heap

A trace of the volatile heap may run concurrently with other operations at the client. The trace obtains a safe view of the heap using the old-view approach described in Section 3.4.1: the trace scans an object at most once—the old state of the object, and it need not scan new objects because they are created marked. It is particularly convenient to run a trace that sees the volatile heap as it existed when the current transaction started. The roots for this trace are the handles that existed at the start of the transaction. When the trace reaches a modified volatile object, it scans the base copy of the object maintained for transactional rollback. Besides providing a consistent view, scanning base copies ensures that there are no dangling references even if the current transaction aborts.

At the end of the trace, all untraced volatile objects that were created before the current transaction are collected, and old outlists are replaced with new ones. Note that new outlists preserve the inter-site invariant: They include all persistent references in the committed copies of live volatile objects. Further, since the trace does not scan new copies of objects modified by the transaction, a new outlist is a subset of the old version. Therefore, the new outlist is a subset of the corresponding inlist.

If a transaction commits while the volatile trace is in progress, the following steps are taken after the commit is confirmed:

1. All objects created during the transaction are marked scanned. This ensures that these objects will not be collected when the trace ends.
2. The add set for the transaction is added to the new outlists under preparation. This is needed for the inter-site invariant.
3. The base copy of each object modified during the transaction is scanned (unless the object is already scanned), because base copies are deleted after the transaction.

5.3 Server-to-Server References

A server records references from another server S in its inlist from S. Here we describe how inter-server references are added to or removed from inlists without sending extra foreground messages. We also describe how inter-site inlists and outlists interact with partition tracing within a server.

5.3.1 Adding References

When a client copies references between objects and commits the transaction, it may create new inter-server references. Figure 5.6 illustrates the most generic scenario: a reference from object a at the source server S to an object b at the target server T is copied into an object c at the recipient server R. At some point, R will send an add message to T, and T will add b in its inlist from R. Until this happens, b must be protected from collection through other means. This section presents an add protocol that is safe and does not add foreground messages to commit processing.

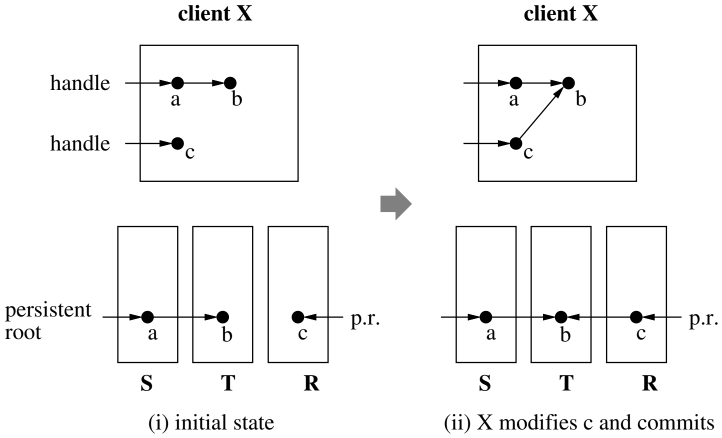

Figure 5.6: Creation of a new inter-server reference.

The add protocol must be robust against participants preparing the transaction at different times. For example, suppose that transaction τ_1 at client X copies the reference $a \rightarrow b$ into c as described above. Soon after server S prepares τ_1, another client commits a transaction τ_2 that removes the reference from a to b. Now, S might complete a local trace and send a remove message to T. If T receives the remove message from S before it receives the add message from R, it might delete b, causing c to contain a dangling reference. A similar situation might arise if S and T are the same server: the local trace at S might delete b before receiving the add message. In general, T might not even be a participant in the commit since client X does not need to read b in order to copy a reference to b.

Another desirable property for the add protocol is that the add message to server T should not delay the commit of τ_1. It is desirable not to add T as a participant since T might be a far-away server requiring long message delays, or T might be temporarily slow or unavailable. Below we present an add protocol that is safe and does not include extra participants or extra foreground messages.

Overview of the Add Protocol

The key insight behind the add protocol is the following. If a client X commits a new inter-server reference to b, at least one site involved in the commit must be already protecting b from collection. This site, called the *guardian* of b, can be made to extend the protection until the target T is known to have received the add message. The guardian may be the client itself or some participant server depending on the following cases:

1. Client X read reference b from a handle or a volatile object, so b must be in the essential outlist of the client. Here, X is the guardian of b.

2. Client X read reference b from a persistent object a at some server S, so either S contains object b ($S = T$) or b must be in the outlist of S. Here, S must be a participant in the transaction and it is the guardian of b.

The add protocol has two parts. The first is conducted during the commit protocol. It informs the guardian, G, to continue protecting b. The guardian stores b in its *guardlist*. A remote reference stored in a guardlist is effectively in the outlist, while a local reference in a guardlist is effectively in the inlist[1]. On the other hand, the recipient R stores the reference b and the identity of the guardian G in its *guardianlist*.

The second part of the add protocol is conducted lazily. R sends an add message to T, and then the guardian is informed to remove b from the guardlist. The add protocol weakens the inter-site invariant as follows:

References from R to $T \subseteq$ outlist of R to T, and
($b \in$ outlist of R to T) $\Rightarrow$ ($b \in$ inlist of T from R) or
($b \in$ guardlist at G and $b \in$ outlist of G to T) or
($b \in$ guardlist at T)

The Commit Part of the Add Protocol

The add protocol is similar to "indirect protection" [Piq91, SDP92]. The problem in implementing it in a client-caching system is that the client does not always know who the guardian is. If the client is not a guardian itself, any participant could be a guardian. Therefore, it must broadcast a request to guard b to all participants. Such a protocol is given below.

1. Before sending the commit request, the client scans each modified object for inter-site references. If the base copy of the object is available, it considers only those references not present in the base copy. For such a reference b in an object at R, the client checks if b is in the client's outlist. If so, it piggybacks a message *guarded*[b, R] on the commit request. Otherwise, it piggybacks the message *guard*[b, R].
2. When the coordinator receives *guard*[b, R] with the commit request, it propagates *guard*[b, R] with the prepare message to each participant. If it receives *guarded*[b, R], it sends *guard*[b, R] only to R.
3. When a participant P receives *guard*[b, R] with a prepare message, it works as follows:
   (a) If $P = R$, and b is already in its outlist, it responds *outlisted*[b, R] with its vote.
   (b) If $P = T$, it enters [b, R] in its guardlist and responds *guarded*[b, R].
   (c) Otherwise, if b is in the outlist of P, it enters [b, R] in its guardlist and responds *guarded*[b, R].
4. The coordinator collects the set of sites (participants or the client) that sent a *guarded* message, called the *potential guardians*. If the coordinator receives *outlisted* or if the transaction aborts, it sends *unguard*[b, R] with the phase-2 message to all potential guardians. Otherwise, it selects a guardian G, sends *unguard*[b, R] to all potential guardians other than G, and sends *guardian*[b, G] to R.
5. When a participant receives *unguard*[b, R] in a phase-2 message, it removes the entry from its guardlist. When R receives *guardian*[b, G], it enters [b, G] in its guardianlist and enters b in its outlist; these steps preserve the inter-site invariant.

[1]The reason why a site might need to store a local reference in its guardlist instead of its inlist is described later.

Note that the guardian the client read the reference from may be different from the one selected by the coordinator. We refer to the former as the *previous* guardian; in the example above, S is the previous guardian. The add protocol transfers protection of the reference from the previous guardian to the selected guardian without any gap in between. The previous guardian either protects the reference until it prepares the transaction or it must abort the transaction. Further, since the previous guardian must be a potential guardian, its guardlist protects the reference until phase 2 of the commit protocol. By then, the selected guardian would have protected the reference in its guardlist.

Note that if the target T is a participant, it will surely be a potential guardian. Nonetheless, the coordinator must send *guard* messages to other participants during phase 1. This is because T might not be the previous guardian, and the previous guardian must protect the reference until T has added it to its guardlist. Further, when T receives *guard*[b, R], it adds b to its guardlist instead of its inlist. If T added b to its inlist at this time, the entry could be deleted by an old remove message that was sent by R before R received the prepare message for the transaction. A safe rule is that references in T's inlist from R are added and removed only upon receiving messages from R.

The extended commit protocol does not send extra foreground messages and it does not require contacting servers that are not participants. It does need to send phase-2 messages to potential guardians other than the selected guardian. In the basic commit protocol, phase-2 messages need not be sent to participants where no object was modified. Therefore, the add protocol sends extra phase-2 messages to potential guardians that were not selected and where no objects were modified. These messages can be buffered and piggybacked on other messages.

The Lazy Part of the Add Protocol

Servers exchange messages in the background to remove entries from guardlist and guardianlist as follows:

1. Server R reads an entry, [b, G], from its guardianlist and sends *add*[b, G] to the target T.
2. T adds b in its inlist from R and sends *unguard*[b, R] to guardian G.
3. G removes [b, R] from its guardlist and sends *unguarded*[b, G] to R.
4. R removes [b, G] from its guardianlist.

It is possible for multiple transactions that commit at R in quick succession to guard the same reference b at other servers. This is because R does not enter b in its outlist until such a transaction commits. Therefore, entries in guardlist and guardianlist and the messages sent to remove them must be distinguished by the id of the transaction that created them.

A desirable property of the protocol described above is that the add message is sent directly from the recipient to the target. Direct add messages ensure that both add and remove messages for updating a given inlist are sent on the same channel (see Section 2.6 for message delivery). Therefore, add and remove messages pertinent to an inlist are received in the order they were sent.

Special Cases

The add protocol described so far accounted for the most generic case. Various optimizations are possible under specific cases.

Target $T \in$ Participants. The lazy part of the add protocol saves one message when T is chosen as the guardian. Therefore, if T is a potential guardian, the coordinator selects it over other potential guardians.

Coordinator U = Recipient R. In this case, the coordinator first checks whether b is in its outlist. If so, it does not send guard messages to other participants. If b is not in its outlist and T is a participant, the coordinator can piggyback an add message on the prepare message to T. In this case, T enters b in its inlist instead of its guardlist, and R does not enter $[b, T]$ in its guardianlist. The coordinator still needs to send guard messages to other participants, however, so that b is protected until T receives the add message.

Coordinator U = Target T. In this case, the coordinator selects itself as the guardian and need not send guard messages to other participants. This is safe because T is effectively the first to receive the prepare message and therefore need not depend on other potential guardians to protect b. Further, the recipient R may piggyback an add message on its vote to the coordinator and bypass entering $[b, T]$ in its guardianlist.

Recipient R = Client X. The case when the client needs to add a reference b to its essential outlist is handled just like when a new inter-site reference is created with the client as the recipient. Thus, the client sends *guard*$[b, X]$ with the commit message; the coordinator selects a guardian and informs the client. An important optimization is possible if T is the only participant in the commit, or if T is the coordinator of the commit. In this case, the client can piggyback an add message to T on the commit request.

5.3.2 Storing Inlists and Outlists

Here we describe how inlists and outlists are organized to support partitioned collection within a server. As mentioned, a site treats another site as a single partition. However, within a site, inlists and outlists must be shared between various partitions such that it is efficient to use inlists as roots and update outlists.

The outlist of server S to T holds references to T from various partitions of S. It is therefore implemented as a collection of partition-level translists: each from some partition in S to T. For example, in Figure 5.7, the outlist of S to T contains pointers to the translists from $S.P$ to T and $S.Q$ to T. Note that these translists are also referenced from the outsets of the source partitions, $S.P$ and $S.Q$.

Similarly, the inlist of T from S holds references from S to various partitions of T. It is also implemented as a collection of partition-level translists: each from S to some partition in T. In the example, the inlist of T from S contains the translist from S to $T.M$ and the translist from S to $T.N$. These translists are also referenced in the insets of the target partitions, $T.M$ and $T.N$.

As described in Section 3.3, references are added to translists between partitions on the same site as modified objects are inter-scanned lazily. This is not the case for translists between different sites because the add protocol must be used to ensure the inter-site invariant. (Therefore, inter-scanning ignores inter-site references.) An inter-site reference from S to T is added to the outlist of S when the transaction creating it commits. The reference is added to the inlist of T when T receives an add message from S. Below, we describe how the partition-level translists composing the outlist of S and the inlist of T are updated.

Translists to Other Sites

For efficient determination of when to send an add or remove message, references in the various translists to other servers are consolidated in a redundant table indexed by the outgoing reference. Each entry in the table, called an *outref*, contains the list of source partitions that contain the given reference. The source list may also contain entries due to the guardlist, if any (see Section 5.3). The structure of the outrefs at S is illustrated below.

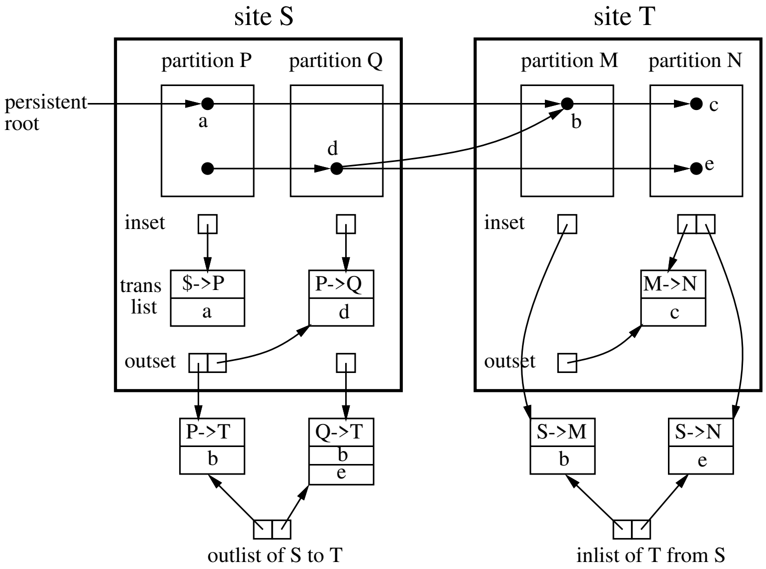

Figure 5.7: References between partitions at different servers.

ref.	source list
b	source: P, Q; guardlist: X
e	source: Q

Whenever a reference to another server is added to a translist (delta list) or the guardlist, the change is reflected in the corresponding outref. When a transaction stores a reference b to server T into an object in partition P of server S, S does the following:

If S has an outref for b when preparing the transaction
  If the source list of b does not contain P
    Add P to the source list of b and add b to the delta list from P to T.
  If the collector is tracing P,
    add b to the new version of the translist from P to T.
  Send an outlisted message to the coordinator.
Otherwise, wait until the transaction commits and then
  Add an outref for b.
  Add P to the source list of b and add b to the delta list from P to T.

Whenever a reference to another server is removed from a translist (after the source partition is traced) or the guardlist (after receiving an unguard message), the change is reflected in the corresponding outref. When the source list of an outref becomes empty, the outref is removed and a remove message for the reference is buffered. In Figure 5.7, when partition P is traced, it would remove d from the translist from P to Q; when partition Q is traced next, it would remove b and e from the translist from Q to server T. Thus, Q would be removed from the source list of outrefs b and e. Since, the source list of outref e is now empty, a remove message is sent for e.

Translists from Other Sites

When server T receives a remove or add message from S, it makes the change in the appropriate translist in its inlist from S. Unlike updates to other translists, updates to translists from other sites are not buffered in delta lists because these updates are not recoverable from the local log.

5.4 Site Crashes

Here we discuss how server and client crashes are handled. We also discuss how network partitions between servers and clients are handled, since they are often indistinguishable from crashes.

5.4.1 Server Crash

When a server recovers from a crash, it must retrieve its inlists before doing a local trace. Further, it must retrieve its outlists so that it can determine when to send add and remove messages. Therefore, the server stores the translists composing its inlists and outlists as persistent objects—like translists between its partitions. The inlist and outlist objects containing pointers to these translists are also stored persistently. The server updates these objects in response to add and remove messages, which are processed in the background.

Further, the server stores guardlists and guardianlists as persistent objects. Updates to the guardlist are logged when preparing the transaction, and updates to the guardianlist are logged when committing the transaction.

5.4.2 Client Crash

Clients may terminate normally or they may crash and never recover. A server must be able to discard its inlists from clients that appear to have failed. Otherwise, it would lose the storage used by the inlists; worse, it would be unable to collect objects that were reachable from the failed clients.

However, there are two problems with discarding information for such clients. First, the client might be a guardian for some reference b stored in another server R. Therefore, removing the inlist from the client might lead to collecting b while R holds a reference to b. Second, a client that appears to have failed may not have crashed; instead it might just be unable to communicate with the server because of a network partition. Since it is impossible to distinguish a client crash from a network partition, it is unavoidable that objects reachable from such a client might be collected. However, servers must guard against the possibility that such a client might later commit a transaction that stores dangling references into objects at the servers. This problem arises because a client might not know that some server has assumed it to have failed, so the client might continue working. In particular, the client might commit transactions at other servers.

For example, consider a client that has a handle to object b at server T. Suppose that T suspects the client has failed, removes the inlist from the client, and collects b. Now, the client might store a reference to b in object c at R and commit the transaction, which creates a dangling reference. Note that dangling references might be created even if add messages were sent synchronously as part of the commit protocol. A dangling reference is not detectable at commit time because it might point to a newer incarnation of the object.

We present a safe protocol that avoids both of the above problems. The basic idea is that before any server discards information about a client, it must inform other servers to *ban* the client, *i.e.*, not service fetch and commit requests from it.

The Ban Protocol

When the client first starts up, it sends a message to some preferred server that will act as a reliable *proxy* for the client. The proxy will always know whether the client has been banned; it stores the id of the client in a stable *clientlist*.

When a server S receives a request from a client (or on behalf of the client, as in a distributed commit), the server checks whether it has a *license* to serve the client. If not, it sends a request to the client's proxy. The identity of the proxy is included in the all client requests. This is the only instance when a foreground message is necessary for the purpose of garbage collection; such messages are acceptable because they are sent infrequently. When the proxy receives a request for a license, it checks whether the client is in its clientlist. If not, it denies the license and, in turn, S signals the client to stop. Otherwise, the proxy enters S in a stable list of licensees for the client. S may give up its license and inform the proxy provided it does not have inlist, guardlist, or guardianlist entries for the client. However, S may cache the license longer to avoid requesting a license frequently.

A server might suspect a client to have failed if the client has not communicated with it for a long time and does not respond to repeated query messages. The server sends a request to the proxy to ban the client. (Also, when a client terminates normally, it sends such a message to its proxy.) The proxy conducts a two-phase ban protocol (illustrated in Figure 5.8):

1. The proxy sends a *ban* message to all licensees.
2. When a server S receives a *ban* message, it decides not to service any further request from the client. Further, if S has an entry in its guardianlist where the client is the guardian:
   (a) S sends an add messages to the target server.
   (b) When the target server acknowledges the add message, S removes the guardianlist entry.

   Then S sends an acknowledgement to the proxy.
3. When the proxy has received acks from all licensees, it enters the second phase and sends *discard* messages to them.
4. Upon receiving this message, a server discards all information about the client. Also, the proxy removes the client from the clientlist.

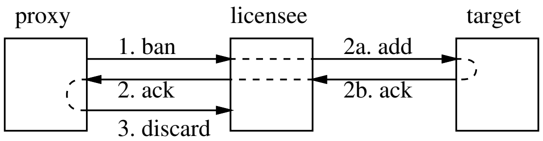

Figure 5.8: The 2-phase ban protocol.

The 2-phase ban protocol provides the following guarantee:

> *No object that is guarded by the client or that may be used by the client in a committed transaction is collected until all servers have banned the client and all servers have removed dependencies on the client for guarding their references.*

After the ban, even if the client approaches a server it never communicated with before, the server will not service the client because the server will fail to get a license from the proxy. Therefore, any dangling references held by the client are harmless.

5.5 Summary

A server records incoming references from clients and other servers in inlists, which are treated as roots when tracing locally. Since clients may cache a large amount of data, recording all references from clients would be a large overhead and might even prevent the collection of garbage. Therefore, a server records only the essential inlist for a client, which consist of references reachable from client handles through zero or more volatile objects. It is necessary and sufficient to treat the essential inlist as roots in local tracing. The client adds references to the essential inlist upon commit and removes references after it traces the volatile heap; no extra work is needed at a server when a client fetches objects. Optimistic concurrency control and the reuse of object names introduce incarnation-mismatch problems. These problems are solved by requiring the server to wait for acknowledgements for invalidation messages before reusing names and before sending remove messages.

Inter-server references may be created when a client commits a transaction. The server receiving the reference sends an add message to the target of the reference. The add message is sent in the background such that it does not delay the commit; if the target server is not a participant, it is not involved in the commit protocol. Thus, no extra messages are sent in the foreground, and no extra participants are added to the commit protocol. A new inter-site reference is guarded by a guardian site until the target is known to have received the add message from the recipient. The guardian may be the client performing the transaction or a participant. Since the guardian might not be known, the coordinator broadcasts a guard request to all participants.

A server stores information such as inlists on stable storage so that the information can be recovered after a crash. When a client crashes or appears to have crashed, the server discards its inlist from the client. Their are two problems in discarding the inlist: First, the client might be guarding a reference for another server. Second, the client might actually be live and might commit dangling references into objects at other servers. These problems are avoided by executing a two-phase protocol. The first phase bans service to the client at all servers and removes dependencies on the client for guarding references. Only after this is complete, are the inlists from the client removed. Thus, clients that are live but banned are harmless.

Chapter 6

Inter-Site Garbage Cycles

This chapter describes fault-tolerant and scalable techniques for collecting inter-site garbage cycles. Treating inter-site references as roots for site-marking retains inter-site garbage cycles and objects reachable from them. Inter-site cycles are relatively uncommon, but they do occur in practice. For example, hypertext documents often form complex cycles spread over many sites.

In Thor, volatile objects at a client cannot be part of inter-site garbage cycles because servers or other clients cannot store references to them. Therefore, inter-site garbage cycles consist only of persistent objects stored at the servers. Thus, a *global marking* scheme spanning all servers could collect inter-site garbage cycles. The roots for global marking would be persistent roots and references in the inlists from clients; we refer to these roots as the *global roots*. However, while marking within a site is acceptable, global marking is a problem because sites may crash independently, become disconnected, or become overloaded and unresponsive. Since complete propagation of global marks involves all sites, global marking might never finish in a system with thousands of sites. Until global marking finishes, no site can collect any inter-site garbage cycle. Therefore, we do not use global marking.

Minimizing inter-site dependence requires the locality property:

Collecting a garbage cycle should involve only the sites containing it.

Locality has proven surprisingly difficult to preserve in collecting inter-site cycles; most previous schemes do not preserve locality. For example, some conduct periodic global marking in addition to site-marking [Ali85, JJ92]. The drawbacks of global marking can be alleviated by marking within groups of selected sites [LQP92, MKI+95, RJ96], but inter-group cycles might never be collected. Few schemes for collecting inter-site cycles have the locality property. A prominent technique is to *migrate* an inter-site garbage cycle to a single site, where it is collected by site-marking [Bis77, SGP90]. However, previous schemes based on migration are prone to migrating live objects besides garbage cycles, which is undesirable because migration is expensive.

We present a practical scheme with locality to collect inter-site cycles. It has two parts. The first identifies objects that are highly likely to be cyclic garbage—the *suspects*. This part may use an unsafe technique that might suspect live objects, although a performance requirement is that few suspects be live. We find suspects by estimating *distances* of objects from persistent roots in terms of inter-site references. This technique preserves locality, has very little overhead, and guarantees that all inter-site garbage cycles are eventually detected.

The second part checks if the suspects are in fact garbage. This part may use techniques that would be too costly if applied to all objects but are acceptable if applied only to suspects. We present two alternatives for checking suspects. The first migrates the suspects such that an inter-site cycle converges on a single site. Unlike previous migration-based proposals, it avoids migration as much as possible. However, some systems do not support migration. The second technique traces *back* from the suspects to check if they are reachable from any global root. Unlike forward global marking, this approach preserves locality and scalability. Back tracing was proposed earlier by Fuchs [Fuc95]. However, that proposal assumed that inverse information was available to trace references backwards, and it ignored problems due to concurrent mutations and forward local traces.

The organization of the full scheme is illustrated in Figure 6.1. Section 6.1 describes the distance heuristic for finding suspects. Section 6.2 describes checking suspects by limited migration, and Section 6.3 describes checking suspects by back tracing. Section 6.4 describes previous proposals for collecting distributed garbage cycles.

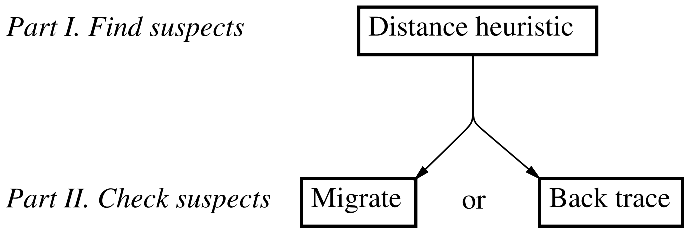

Figure 6.1: Two-part scheme for collecting inter-site garbage cycles.

6.1 Distance Heuristic

The insight behind the distance heuristic is that the set of objects reachable from global roots through a bounded number of inter-site references gets closer to the set of all live objects as the bound is increased. Therefore, objects not in the first set may be suspected to be garbage. The distance of an object is defined as follows:

> The *distance* of an object is the minimum number of inter-site references in any path from a global root to that object. The distance of an object unreachable from the global roots is infinity.

Figure 6.2 illustrates the notion of distance. Object a is a global root; therefore, the distance of a and objects locally reachable from it is zero. (An object is *locally reachable* from another if there is a path of local references from the first to the second.) Object c is reachable from a through two paths: one with two inter-site references and another with one; therefore, its distance is one.

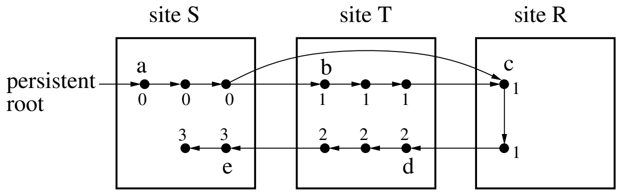

Figure 6.2: Distances of object.

Suspects are found by *estimating* distances. The estimated distances of objects tend towards their real values: those of live objects converge to finite values, while those of uncollected cyclic garbage keep increasing with time.

6.1.1 Estimating Distances

Object distances are estimated efficiently as follows. Rather than associating a distance field with each object, a distance field is associated with each entry in inlists and outlists. The distance of global roots (references in inlists from clients and persistent roots) is zero. Distances are propagated from inlist entries to outlist entries during local tracing, and from outlist entries to inlist entries on target sites through update messages. Update messages are sent in the background; in particular, they may be piggybacked on remove messages carrying changes in the outlists.

For efficiency, references in the various inlists are consolidated into a redundant table indexed by references. Each entry in the table, called an *inref*, records the list of source sites containing the reference. Each source site in the list is associated with a distance; the distance of the inref as a whole is the minimum such distance. For example, in Figure 6.3, site R has an inref for *c*, with a distance one from S and two from T; its distance is one. The inrefs table is similar to the outrefs table introduced in Section 5.3.2, which stores a consolidated view of the various outlists at a site. The source list of an outref records the partitions that contain the outgoing reference, but we omit these source lists in the figures for clarity.

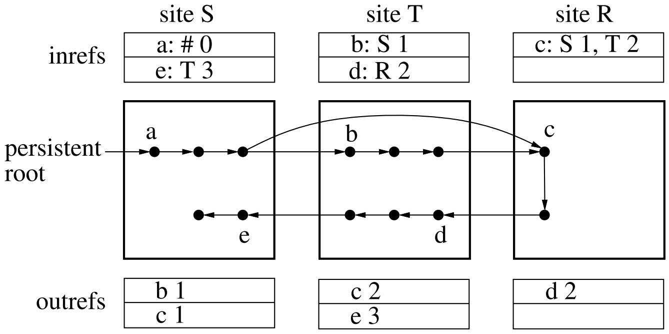

Figure 6.3: Distances associated with inrefs and outrefs

Local tracing is responsible for setting the distance of each outref to one plus the minimum distance of any inref it is locally reachable from. First we describe how distances would be propagated if a site were traced as a unit. In order to propagate distances correctly, inrefs are traced in the order of increasing distances, and the trace from one inref is completed before tracing from the next. When the trace reaches an outref for the first time, the distance of the outref is set to one plus that of the inref being traced from. For example, in Figure 6.3, site S will trace from inref a before it traces from inref e. Therefore, it sets the distance of outref b to one plus that of a. Changes in the distances of outrefs are propagated to the target sites in update messages. When a target site receives such a message, it reflects the changes in its inrefs.

Estimating Distances Through Partition Traces

Since a server traces its partitions independently, each partition trace must propagate distances. A distance field is stored with each reference in translists. The references in the inset of a partition P are traced in the order of increasing distances, and the trace from one inset reference is completed (within P) before tracing from the next. The distance of a reference o in the outset of P is set to that of the inset reference being traced if o is a local inter-partition reference, and to one plus that if o is an inter-site reference.[1] Finally, the source list of an outref records the distance from each source partition; the distance of the outref as a whole is the minimum such distance.

Since distances of references in delta lists are not known, they are conservatively assumed to be zero. This is suitable because such a reference must be live—at least until the modification that caused it to be entered in the delta list. Tracing the source partition later computes the appropriate distances of these references.

If there is a path of references from an inref i to an outref o through partitions $P_1 \ldots P_m$, changes in the distance of i are propagated to o when the sequence of partitions traced, $\overline{T}$, includes $P_1 \ldots P_m$ as a subsequence. Suppose there are n total partitions and they are traced in round-robin order. Then, propagating the change might take 1 round in the best case (when a round traces the partitions in the desired order, $P_1 \ldots P_m$), and m rounds in the worst case (when each round traces the partitions in the opposite order, $P_m \ldots P_1$). Propagation of distance changes can be accelerated by increasing the trace-priority of any partition that has unpropagated distance changes in its inset. This approach is similar to the reduced-inset heuristic described in Section 3.2.

Overheads and Constraints

The space overhead of distance propagation is a distance field per translist entry, and a distance field per source in inrefs and outrefs. A distance field requires only a few bits: a one-byte field can account for chains with 255 inter-site references. The message overhead consists of information about changes in distances piggybacked on remove messages. The processing overhead is negligible because distance propagation within a site is piggybacked on local tracing. Distance propagation does constrain the order in which the roots must be traced in each partition, and it requires completing the trace from one root before the next root is traced. However, this does not constrain the trace to use a depth-first traversal since objects reachable from one root can be traced in any order before the next root is traced.

Distance propagation also relies on the use of source-listing of inter-site references rather than reference counting (Section 5.1). The source list of an inref stores a distance for each source site separately. Thus, in the example shown in Figure 6.3, if T discards the reference to c, R can update the distance of inref c to the new minimum, two. With reference counts, it would not be possible for R to find the new distance.

6.1.2 Deviation in Distance Estimates

When a new inter-site reference is created and the target site receives an add message, the distance associated with the reference is conservatively set to one for want of better information. Later, local tracing at the source site computes the appropriate distance of the reference. The change is then propagated to the target site. Similarly, distances of all references in the guardlist are conservatively assumed to be one. This is suitable because these references were live when they were stored and because they are stored in the guardlist only temporarily.

Estimated distances may also be *higher* than real distances temporarily. In the example in Figure 6.4, creation of a local reference in T reduces the real distance of e to two. However, the estimated distance of e is not updated until local tracing at T finds the shorter path and T sends an update message to S.

Thus, changes in estimated distances may lag behind changes in real distances. Temporary deviations in estimated distances are acceptable since distances are used merely as a heuristic for

[1] A more general distance heuristic could account for both inter-partition and inter-site references by assigning a non-zero weight to each inter-partition reference and a higher weight to each inter-site reference.

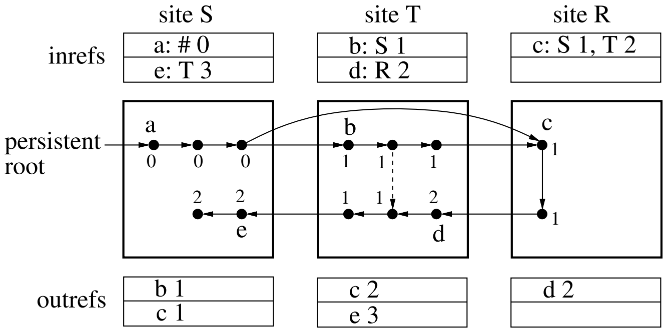

Figure 6.4: Distances drop on creation of references (dotted arrow).

suspecting cyclic garbage. A negative deviation in the distance of a live object is benevolent because the object will not be suspected. A high positive deviation from the real distance might cause the object to be suspected, but that is acceptable provided such occurrences are not common.

In the rest of this chapter, we will use "distance" to mean "estimated distance."

6.1.3 Distance of Cyclic Garbage

Distances of cyclic garbage increase without bound as they are propagated through local tracing and update messages. Intuitively, this happens because a garbage cycle acts as a positive feedback loop: each local trace increments the distances when propagating them from inrefs to outrefs. When a cycle is connected from a global root, the path from the global root holds down the distances; when the cycle is disconnected, there is no such constraint. Below, we quantify the rate of increase of distances of cyclic garbage.

The delay in propagation of distance from an inref to an outref and then to the target inref depends on factors such as the frequency of tracing at the site and the number of partitions in the path from the inref to the outref. For simplicity of analysis, assume that this delay for inrefs and outrefs in the cycle of interest is bounded by a certain period of time, called a *step*.

First consider the simple cycle in Figure 6.5. As long as the cycle is connected from a, the distance of c is 1, d is 2, and e is 3. Within a step after the cycle is disconnected, S recomputes the distance of outref c as 4 and propagates it to R. In the next step, R sets the distance of d to 5; then T sets the distance of e to 6; so on. The progress of distances is plotted in the figure; the zero on the horizontal axis marks the time when the cycle is disconnected.

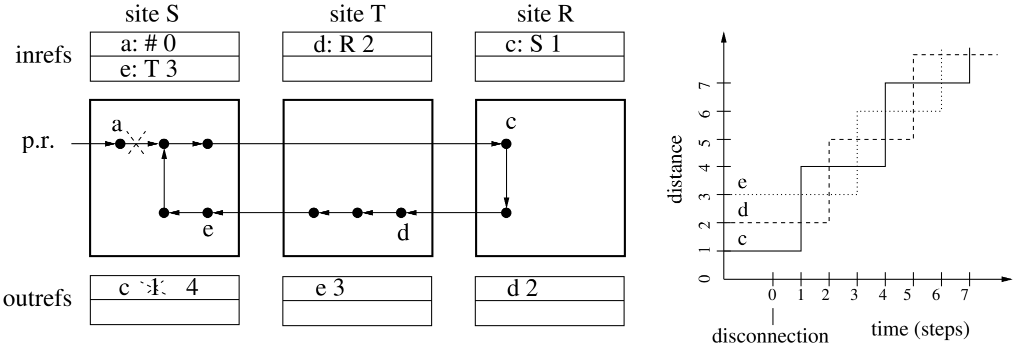

Figure 6.5: Distances of a disconnected cycle (dotted cross).

In fact, the following theorem holds for arbitrary graphs of garbage objects, including multiple cycles connected to form a compound cycle.

Garbage Distance Theorem

J steps after an object became garbage, the estimated distance of the object will be at least J if the object is not collected by then.

Proof (by induction)

Consider an object c that became garbage at some point in time. At that time, the set of objects that c is still reachable from, $\mathcal{G}$, must be garbage as well. Further, since the mutator does not create new references to garbage objects, this set cannot grow with time.

The theorem holds trivially when J is zero. Suppose the theorem holds when J is K. Thus, K steps after c became garbage, the distance of any inref for an object in $\mathcal{G}$ must be at least K. Now consider any outref to an object in $\mathcal{G}$. This outref can be reachable only from inrefs for objects in $\mathcal{G}$ (by the definition of $\mathcal{G}$). Therefore, within the next step, either the outref will be collected or the distance of the outref will be set to at least $K + 1$. Since the distances of outrefs are reflected in the corresponding inrefs, the distance of any inref for an object in $\mathcal{G}$ will be at least $K + 1$.

□

The propagation of distances in the simple example in Figure 6.5 illustrates the theorem. Note that the theorem is not limited to inter-site cyclic garbage. However, garbage other than inter-site cycles will be collected; therefore, it is only the inter-site garbage cycles (and objects reachable from them) whose distances keep increasing with time.

6.1.4 The Threshold Distance

Since estimated distances of live objects converge on their actual distances, while those of garbage cycles keep increasing, objects with distances above a *suspicion threshold* are suspected to be cyclic garbage. Inrefs with distances less than or equal to the threshold —and objects and outrefs traced from them—are said to be *clean*. The remaining are said to be *suspected*.

Setting the threshold involves a tradeoff. The higher the threshold, the fewer the live objects suspected, but the longer the time to identify suspects. Thus, the accuracy of suspicion can be controlled by setting the threshold appropriately. Fortunately, the penalty on setting the threshold a little high or a little low is not severe. In particular, neither safety nor completeness is compromised. If live objects are suspected, the scheme that checks suspects will ensure that they are not collected. On the other hand, cyclic garbage is sure to be detected eventually; the Garbage Distance Theorem ensures that the distances of all cyclic garbage will cross the threshold, regardless of how high the threshold is.

The choice of the threshold in a system depends on two factors:

- The rate of generation of inter-site cyclic garbage: The higher this rate, the higher the need to collect the garbage quickly, so the lower the threshold ought to be.
- The distribution of distances of the objects: If there are many live objects with large distances, the threshold should be high enough to avoid suspecting many live objects.

The threshold can be adapted dynamically by receiving feedback from the scheme that checks whether suspects are in fact garbage. For example, if too many suspects are found to be live, the threshold should be decreased. The design of such a mechanism requires further work.

6.1.5 Summary of Distance Heuristic

Distance propagation preserves the locality property and fault tolerance of local tracing. Identifying a suspect involves only the sites it is reachable from. No global mechanism is required. Thus, if a site S is down, disconnected, or otherwise slow in tracing and sending update messages, it will only delay the identification of the garbage reachable from S. Further, it will not cause more live objects to be suspected.

The distance heuristic identifies all inter-site cyclic garbage, since their distances will cross any threshold eventually. Its accuracy in suspecting only inter-site cyclic garbage can be controlled by setting the threshold suitably. Distance propagation is piggybacked on local tracing with little overhead. Its message overhead consists of distance information in update messages and its space overhead consists of distance fields in translist entries.

6.2 Migration

The basic idea behind migration is to converge objects on a potential garbage cycle on one site. If the objects are in fact garbage, they will be collected by local tracing; otherwise, they will survive, although displaced from their original sites.

We assume an underlying mechanism for migrating objects that is robust against concurrent mutations and that updates references to migrated objects [DLMM94, Ady94]. In Thor, a migrated object leaves a *surrogate* object at the old location that contains a forwarding reference to the new location of the object. Old references to a surrogate are updated in the background. As with normal objects, a surrogate is collected when it is not reachable from applications.

Migration has been proposed earlier to collect distributed cycles [Bis77, SGP90]. However, previous proposals used weak heuristics to decide which objects to migrate. A popular heuristic was to move to all objects not locally reachable from a global root. For example, in the state shown in Figure 6.6, such a heuristic would move the live objects such as *b* and *c* to site S. Another heuristic is to migrate such objects only if they have not been accessed for some period. In a persistent store, however, live objects may not be accessed for long periods.

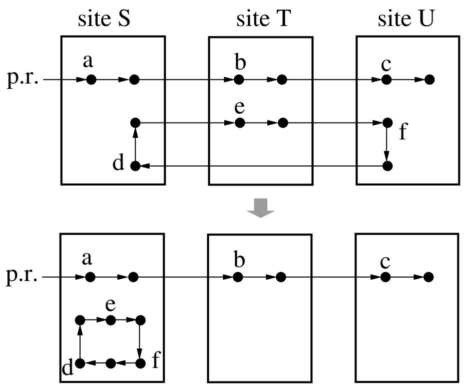

Figure 6.6: Desired migration of an inter-site cycle to a single site.

Migration of live objects wastes processor and network resources, and it might spoil load balance and object placement. For example, a database might be partitioned between two sites such that objects are positioned close to their frequent users. Yet the database may have a single logical root, say, on the first site. Objects on the second site are locally unreachable from a global root, but they should not be moved. In general, object placement should not be bound to garbage collection; instead, it should admit an independent policy such as hints from applications.

The distance heuristic provides a technique for limiting migration of live objects to an arbitrary level. In fact, the previous heuristic is a degenerate case of distance heuristic when the threshold is zero. Only objects traced from suspected inrefs are migrated. Specifically, all objects traced from a suspected inref are migrated as a batch. Note that the set of objects *traced* from an inref may be smaller than the set of objects locally *reachable* from it. For example, in site T in Figure 6.7, inref b is trace before inref e; therefore, object b_2 is traced from b and e_2 is traced from e. If the suspicion threshold distance is 10, e_2 will be batched for migration with e, while b_2 will not be migrated. This is desirable because b_2 is likely to be live.

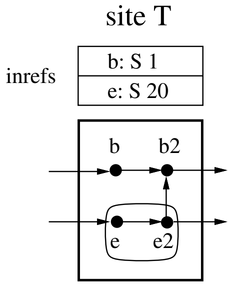

Figure 6.7: Objects traced from a suspected inref are batched for migration.

6.2.1 Where to Migrate

Some previous schemes migrate objects to a fixed dump site [GF93], but this can be a performance or fault-tolerance bottleneck in a large system. Other schemes migrate objects to sites that refer to them. To ensure that objects in a cycle converge on the same site instead of following each other in circles, sites are totally ordered and migration is allowed in one direction only, say, from higher-id sites to lower-id sites [SGP90]. We refer to this rule as the *one-way* rule.

The one-way rule ensures convergence, but objects on a multi-site cycle might be migrated multiple times before they converge on a site. For example, in Figure 6.6, object f and others in its batch might be migrated first to site T and then to site S. In general, for a simple cycle that spans C sites, $O(C^2)$ object migrations might be performed: the object closest to the final destination site is migrated once, while the object farthest from it might be migrated up to $C - 1$ times.

Since migration is an expensive operation, we limit the number of migrations by migrating objects directly to the final destination as much as possible. The *destination* of a suspected object is defined as the lowest-id of any site from where it is reachable.

Destinations of suspected objects are estimated just like distances. Suspected inrefs and outrefs have destination fields, and local tracing propagates destinations from suspected inrefs to outrefs. In order to propagate destinations of suspected inrefs correctly, inrefs with distances above the suspicion threshold are traced in the order of increasing destination. It is acceptable to not trace them in distance order because they are already suspected to be cyclic garbage. If suspected inrefs were traced in distance order, the lowest site-id might not propagate fully in a compound cycle. (For instance, if an outref o is reachable from two suspected inrefs i_1 and i_2, such that i_1 had a lower distance and i_2 had a lower destination, o would be traced from i_1, blocking destination information from i_2.) When the trace reaches a suspected outref for the first time, the destination of the outref is set to the minimum of the local site-id and the destination of the inref being traced. Finally, changes in the destination of outrefs are propagated to target sites in update messages.

Migration is not necessary for objects that are reachable from an inter-site garbage cycle but are not part of the cycle. If these objects did not migrate, they would still be collected, though after the cycle has been collected. The scheme presented here does not prevent the migration of such objects, but it does avoid migrating them multiple times. It may migrate them to different sites, however. For instance, consider Figure 6.8, where the site with the lowest id in the cycle is S, while a garbage chain passes through site R with an even lower id. In this case, object b will be migrated to S, while object d will be migrated to R.

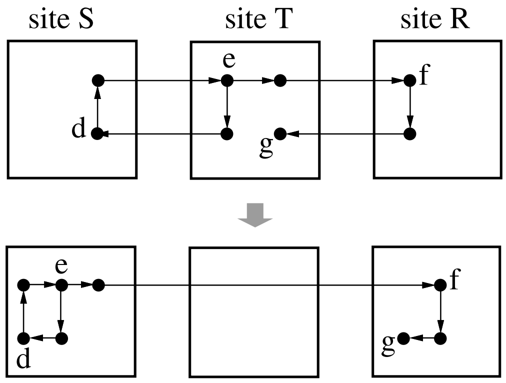

Figure 6.8: A chain reachable from a cycle may migrate to a different site.

Unlike some leader election algorithms [LeL77], estimating the destination does not incorporate termination detection, so sites must guess when destination propagation has completed; we discuss how to make this guess in the next section. The advantage of our scheme is that it is simple and effective even in compound cycles.

6.2.2 When to Migrate: the Second Threshold

A site migrates a batch of suspected objects only after it is likely to have received the final destination for the batch. This is achieved by waiting until the distance of a suspected inref crosses a *migration threshold* that is higher than the suspicion threshold.

Setting the migration threshold, t_m, involves a tradeoff similar to that for the suspicion threshold, t_s. It should be high enough that by the time the distance of an inref increases to t_m, its destination field is likely to be set correctly. But it should be low so that cyclic garbage is migrated quickly. The penalty on setting the threshold a little low or high is not severe because the destination is used only as a heuristic. If the destination is not set correctly when objects are migrated, the objects might be migrated again, but the one-way rule still ensures eventual convergence. Further, even if the destination is not set correctly, the objects are likely to be migrated fewer times than if the heuristic were not used at all.

Below we provide an estimate for t_m based on a simple garbage cycle with C inter-site references. The distance of any inref in such a cycle jumps up by C after every C steps, as shown in Figure 6.5. Consider the inref i on the lowest-id site in the cycle. At some point, the distance of i will cross t_s. At that time, the distances of other inrefs on the cycle must be less than $t_s + C$. After the distance of i crosses t_s, the site-id of i will be propagated around the cycle as the destination. In particular, when the distance of any other inref crosses $t_s + C$, it will have received the destination information from i. Thus, a suitable value for t_m is $t_s + C$, where C is a conservatively estimated (large) cycle length in the system. Possible future work in this area is to adapt t_m dynamically.

6.2.3 Summary of Migration

Migration is a safe technique for checking suspects because suspects are collected only if local tracing finds them unreachable. Further, migration is robust against concurrent modifications; therefore, the complexity of handling modifications is limited to the migration module. Migration preserves the locality property: the collection of a cycle involves only the sites containing the cycle. Migration collects all inter-site cycles except those that are too big to fit in a single site.

Migration has potentially high processor and network overhead. The heuristics presented here help to limit migration: the distance heuristic limits the number of objects migrated and the destination heuristic limits the times they are migrated. However, migration of large garbage objects can still be expensive. Another drawback of migration is that it might conflict with autonomy or security concerns. Some systems cannot migrate objects across heterogeneous architectures, and some do not support migration because of the complexity of updating references to migrated objects. Finally, the one-way rule makes migration unfavorable to sites with lower ids.

6.3 Back Tracing

The key insight behind back tracing is that whether an object is reachable from a global root is equivalent to whether a global root is reachable from the object if all references are reversed. Thus, the idea is to trace backwards from a suspect: if the trace encounters a global root, the suspect is live, otherwise it is garbage.

The virtue of back tracing is that it has the locality property. For example, a back trace started in a two-site cycle will involve only those two sites. This is in contrast to a forward trace from the global roots, which is a global task. Indeed, a forward global trace would identify *all* garbage in the system, while a back trace checks only whether a particular object is live. Thus, back tracing is not suitable as the primary means of collecting garbage. Most garbage is expected to be collected by forward local tracing (partition tracing). Back tracing is a complementary technique to detect uncollected garbage, and is used for objects suspected to be on inter-site garbage cycles. We present practical techniques for conducting back tracing.

6.3.1 The Basic Scheme

This section describes the basic scheme for back tracing; Section 6.3.2 describes the computation of information required for back tracing, and Section 6.3.3 describes solutions to problems due to concurrent mutations and forward traces. Section 6.3.4 describes the consequences of *partitioned* local tracing for back tracing.

Back Steps

Tracing back over individual references would be prohibitively expensive—both in the time required to conduct it and in the space to store the required information. Therefore, a back trace leaps between outrefs and inrefs. For brevity, we refer to inrefs and outrefs collectively as *iorefs*. A back trace takes two kinds of steps between iorefs:

Remote steps that go from an inref to the corresponding outrefs on the source sites.
Local steps that go from an outref to the inrefs it is locally reachable from.

The information needed to take remote steps from an inref is already present in its source list, but extra information is required to take local steps. We define the *inreach* of a reference as the set of inrefs from which it is locally reachable. For example, in Figure 6.9, the inreach of outref c in site T is $\{a, b\}$. We compute and store the inreaches of outrefs so that back traces may use them when required. Section 6.3.2 describes techniques for computing inreaches efficiently during forward local traces.

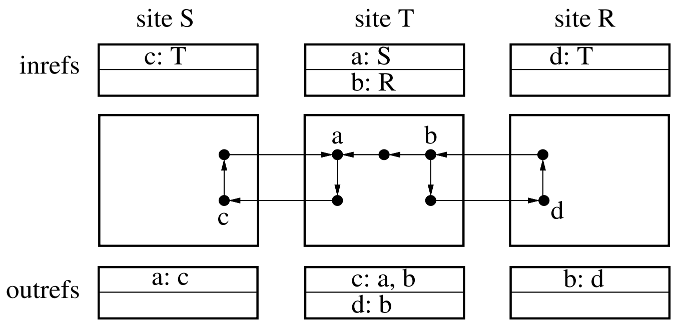

Figure 6.9: Inreaches of suspected outrefs.

A back trace consists of taking local and remote steps alternately. For example, a back trace at outref c in T will take local steps to inrefs a and b. From there, it will take remote steps to outrefs a and b and so on.

In general, a back trace may be started from any suspected ioref. However, a back trace started from an inref a will not find paths to object a from other inrefs on the same site. For example, in Figure 6.9, a back trace started from inref a will miss the path from inref b to object a. On the other hand, a back trace started from an outref c that is locally reachable from a will find all paths to a because the set of inrefs that c is locally reachable from must include all inrefs that a is locally reachable from. Thus, if a back trace started from an outref does not encounter a global root, all inrefs visited by the trace must be garbage objects. Therefore, we start a back trace from an outref rather than an inref.

How Far Back to Go

A practical requirement on a back trace is to limit its spread to *suspected* iorefs. Thus, rather than going all the way back in the search for a global root, a back trace returns "Live" as soon as it reaches a clean ioref. This rule limits the cost of back tracing to the suspected parts of the object graph in two ways:

- A back trace from a live suspect does not spread to the clean parts of the object graph.
- We need to compute inreaches for suspected outrefs only.

The rule has the disadvantage that back tracing will fail to identify a garbage cycle until all objects on it are suspected. This is not a serious problem, however, because the distance heuristic ensures that all cyclic garbage objects are eventually suspected. The next section describes a suitable time to start a back trace.

When to Start

A site starts a back trace from a suspected ioref based on its distance. There is a tradeoff here. A back trace started soon after an ioref crosses the suspicion threshold, t_s, might run into a garbage ioref that is clean because its distance has not yet crossed t_s and return Live unnecessarily. On the other hand, a back trace started too late delays collection of inter-site garbage cycles.

Here, we estimate a suitable *back threshold* t_b to trigger a back trace. (This threshold is similar to the one for migrating suspected objects.) Ideally, by the time the distance of any ioref on a cycle is above t_b, those of other iorefs on the cycle should be above t_s. If the distance of an ioref y is t_b and the distance from x to y is C, then the distance of x should be at least $t_b - C$. Therefore, an appropriate value for t_b is $t_s + C$, where C is a conservatively estimated (large) cycle length.

If a back trace is started in a garbage cycle too soon, the trace might return Live unnecessarily, but a future trace would confirm the garbage. To determine when to start another back trace, each suspected ioref has a back-threshold field initially set to t_b. When a back trace visits an ioref x, the back threshold of x is incremented by, say, C. Thus, the next back trace from x, if any, is triggered only when its distance crosses the increased threshold. This has the following desirable properties:

1. Live suspects stop generating back traces once their back thresholds are above their distances.
2. Garbage objects generate periodic back traces until they are collected.

Back Tracing Algorithm

Back tracing can be formulated as two mutually recursive procedures: BackStepRemote, which takes remote steps, and BackStepLocal, which takes local steps. Both are similar to a standard graph search algorithm.

```
BackStepRemote (site S, reference i)
    if i is not in Inrefs return Garbage
    if Inrefs[i] is clean return Live
    if Inrefs[i] is visited by this trace return Garbage
    mark Inrefs[i] visited by this trace
    for each site T in Inrefs[i].Sources do
        if BackStepLocal(T, i) is Live return Live
    return Garbage
end

BackStepLocal (site S, reference o)
    if o is not in Outrefs return Garbage
    if Outrefs[o] is clean return Live
    if Outrefs[o] is visited by this trace return Garbage
    mark Outrefs[o] visited by this trace
    for each reference i in Outrefs[o].Inreach do
        if BackStepRemote(S, i) is Live return Live
    return Garbage
end
```

If the reference being traced is not found among the recorded iorefs, its ioref must have been deleted by the garbage collector, so the call returns Garbage. To avoid revisiting iorefs and to avoid looping, an ioref remembers that a trace has visited it until the trace completes; if the trace makes another call on the ioref, the call returns Garbage immediately. Note that the calls within both for-loops can be made in parallel. For example, in Figure 6.10, a call at inref c in R will fork off two branches to S and T. One branch will visit inref a first and go further back, while the other will return Garbage.

An activation frame is created for each call. A frame contains the identity of the frame to return to (including the caller site), the ioref it is active on, a count of pending inner calls to BackStep, and a result value to return when the count becomes zero. Thus, a call returns only when all inner calls have returned. We say that a trace is *active* at an ioref if it has a call pending there. We say that a trace has *visited* an ioref if the ioref is marked visited by the trace—even though the trace may have returned from there.

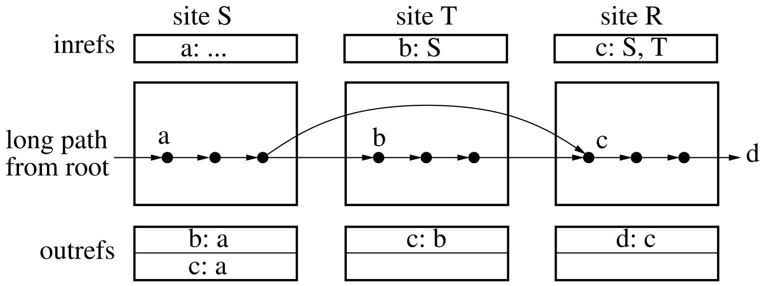

Figure 6.10: A back trace may branch.

Collecting Garbage

If the outer-most call to BackStep returns Garbage, all inrefs visited by the trace are garbage. However, when an *intermediate* call returns Garbage, it cannot be inferred that the corresponding ioref is garbage, because that call would not have visited iorefs that have already been visited by other branches in the same trace. For example, in Figure 6.9, the call at inref b may return Garbage because inref a has been visited, although b is not garbage. Therefore, no inref is removed until the outer-most call returns.

When the outer-most call returns, the site that initiated the back trace reports the outcome to all sites reached during the trace, called the *participants*. For the initiator to know the set of participants, each participant appends its id to the response of a call. If the outcome is Garbage, each participant flags the inrefs visited by the trace as garbage. If the outcome is Live, each participant clears the visited marks for that trace. Note that the outcome is likely to be Garbage since the suspected objects are highly likely to be garbage.

An inref flagged as garbage is not used as a root in the local trace. Such an inref is not removed immediately in order to maintain the inter-site invariant between outrefs and inrefs. Flagging the visited inrefs causes the cycle to be deleted the next time the containing sites do a local trace. The flagged inrefs are then removed through regular update messages.

Message Complexity

Back tracing involves two messages for each inter-site reference it traverses—one for the call and another for its response. Also, reporting the outcome involves a message to each participant. Thus, if a cycle resides on C sites and has E inter-site references, $2E + C$ messages are sent. These messages are small and can be piggybacked on other messages.

Loss of messages can be handled by using timeouts as follows. A site S waiting for the response to a call to site T may send a query message to T. If the call is still pending at T when T receives the query message, it returns a $wait$ message. If T has no information about the call, it conservatively returns Live. If S does not receive a response to repeated query messages, it times out and assumes that the call returned Live. Similarly, a site waiting for the final outcome may time out on the site that initiated the trace after repeated query messages, and assume that the outcome is Live. The use of wait messages makes the timeout mechanism independent of the size of the cycle; in particular, bigger cycles are not more susceptible to timeouts. Instead, timeouts depend on the communication channels between pairs of sites.

Multiple Back Traces

Several back traces may be triggered concurrently at a site or at different sites. In fact, multiple traces might be active at an ioref or at iorefs in the same cycle. Traces are distinguished by a unique id assigned by the starting site. Thus, the visited field of an ioref stores a set of trace ids.

Multiple back traces at iorefs in the same cycle are wasteful. However, such traces are not likely for the following reason. The distances of various iorefs in a cycle are likely to be different such that one of them will cross the threshold t_b first. Even if several iorefs have the same distance, there will be differences in the real time at which they cross t_b due to randomness in when local traces happen. The time between successive local traces at a site is long—on the order of minutes or more—compared to the small amount of processing and communication involved in a back trace—on the order of milliseconds at each site (or tenths of a second if messages are deferred and piggybacked). Therefore, the first back trace started in a cycle is likely to visit many other iorefs in the cycle before they cross t_b.

There is no problem if one trace confirms garbage and results in the deletion of an ioref when another trace has visited it. The second trace can ignore the deletion, even if it is active at the ioref, because activation frames provide the necessary return information.

6.3.2 Computing Back Information

Back information comprises the source sites of inrefs (for remote steps) and the inreaches of outrefs (for local steps). The source sites are maintained by the add protocol described in Section 5.3. This section describes the computation of the inreaches of suspected outrefs. Inreaches are computed during forward local tracing at a site and are stored such that they are ready for use by back traces when needed. This section assumes that a site is traced as a unit; Section 6.3.4 describes the implications of partitioned local tracing.

Inreaches of outrefs are computed by first computing their inverse, namely, the *outreaches* of suspected inrefs. The outreach of a reference is the set of suspected outrefs locally reachable from it. Outreaches of inrefs and inreaches of outrefs are merely two different representations of the reachability information between inrefs and outrefs.

It is desirable to compute outreaches during the forward local trace from suspected inrefs. However, a trace does not provide full reachability from inrefs to outrefs. This happens because a trace scans a reachable object only once (or sometimes twice for site-marking), which is crucial for linear performance. For example, in Figure 6.11, if a is traced before b, then the trace from a will visit z first. Therefore, the trace from b will stop at z and will not discover the outref c. Below we describe two techniques: the first is straightforward but may retrace objects, while the second traces each object only once.

Independent Tracing from Inrefs

The straightforward technique is to trace from each suspected inref ignoring the traces from other suspected inrefs. Conceptually, each trace from a suspected inref uses a different color to mark visited objects. Thus, objects visited by such a trace may be revisited by another trace. However, objects traced from clean inrefs are never revisited. They may be considered marked with a special color, black, that is never revisited. As mentioned before, clean inrefs are traced before suspected inrefs.

Separate tracing from each suspected inref reaches the complete set of suspected outrefs locally reachable from it. The problem with this technique is that objects may be traced multiple times.

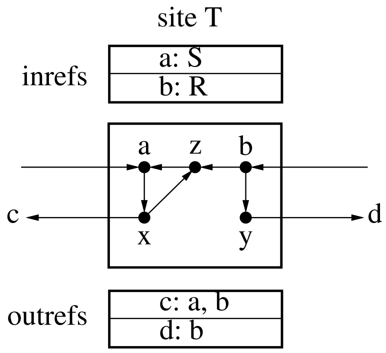

Figure 6.11: Tracing does not compute reachability.

If there are n_i suspected inrefs, n suspected objects, and e references in the suspected objects, the complexity of this scheme is $O(n_i \times (n + e))$ instead of the usual $O(n + e)$.

Bottom-up Computation

Outreaches of suspected inrefs may be found by computing the outreaches of suspected objects bottom up during the forward trace. The outreaches are remembered in an *outreach table*, which maps a suspected object to its outreach. Once the outreach of a suspect z is computed, it is available when tracing from various inrefs without having to retrace z. The following is a first cut at the solution, using a depth-first traversal of suspected objects:

```
ScanSuspected(reference x)
    Outreach[x] := null
    for each reference z in x do
        if z is clean continue loop
        if z is remote add z to Outreach[x] and continue loop
        if z is not marked
            mark z
            ScanSuspected(z)
        Outreach[x] := Outreach[x] ∪ Outreach[z]
    endfor
end
```

The above solution does not work because it does not account for backward edges in the depth-first tree correctly. For example, in Figure 6.11, if inref a is traced first, the outreach of z would be set to null erroneously instead of $\{c\}$ because, when z is scanned, the outreach of a is null. Since the outreach of inref b uses that of z, it would miss c as well. In general, a backward edge introduces a strongly connected component, and the outreaches of objects in a strongly connected component should all be equal.

Fortunately, strongly connected components can be computed efficiently during a depth first traversal with linear performance [Tar72]. For each object, the algorithm finds the first object visited in its component, called its *leader*. The algorithm uses a counter to mark objects in the order they are visited. It also uses an auxiliary stack to find the objects in a component. The following code combines tracing, finding strongly connected components, and computing outreaches in a single depth first trace. The algorithm sets the outreach of each object to that of its leader.

```
ScanSuspected(reference x)
    push x on Stack
    Outreach[x] := null
    Leader[x] := Mark[x]
    for each reference z in x do
        if z is clean continue loop
        if z is remote add z to Outreach[x] and continue loop
        if z is not marked
            Mark[z] := Counter
            Counter := Counter+1
            ScanSuspected(z)
        Outreach[x] := Outreach[x] ∪ Outreach[z]
        Leader[x] := min(Leader[x], Leader[z])
    endfor
    if Leader[x] = Mark[x] % x is a leader
        repeat
            z := pop from Stack % z is in the component of x
            Outreach[z] := Outreach[x]
            Leader[z] := infinity % so that later objects ignore z
        until z = x
end
```

This algorithm traces each object only once. In fact, it uses $O(n + e)$ time and $O(n)$ space except for the union of outreaches. In the worst case, if there are n_o suspected outrefs, the union of outreaches may take $O(n_o \times (n + e))$ time and $O(n_o \times n)$ space. Below we describe efficient methods for storing outreaches and for implementing the union operation; these methods provide near-linear performance in the expected case.

First, suspected objects that have the same outreach share storage for the outreach. The outreach table maps a suspect to an *outreach id* and the outreach itself is stored separately in a canonical form. If objects are well clustered, there will be many fewer distinct outreaches than suspected objects. This is because there are expected to be many fewer outrefs than objects; further, objects arranged in a chain or a strongly connected component have the same outreach.

Second, the results of uniting outreaches are memoized. A hash table maps pairs of outreach ids to the outreach id for their union. Thus, redoing memoized unions takes constant time. Only if a pair is not found in the table, is the union of the two outreaches computed. Another table maps existing outreaches (in canonical form) to their ids. If the computed outreach is found there, we use the existing id.

The various data structures used while tracing from suspected inrefs are deleted after the trace. Only the outreaches of the suspected inrefs are retained. As mentioned, these outreaches are equivalent to keeping the inreaches of the suspected outrefs, since one may be computed from the other. The space occupied by inreaches or outreaches is $O(n_i \times n_o)$, where n_i and n_o are the number of suspected inrefs and suspected outrefs. The space overhead is less than other non-migration-based schemes with locality [Sch89, LC97]. These other schemes require complete local-reachability information between inrefs and outrefs—not just suspected ones, and they store additional path information for each inref.

6.3.3 Concurrency

The description so far assumed that back traces used the information computed during previous local traces and there were no intervening modifications. In practice, clients may modify the object graph such that the computed back information is no longer accurate. Further, modifications, forward local tracing, and back tracing may execute concurrently. This section presents techniques to preserve safety and completeness in the presence of concurrency. We divide the problem into several parts for simplicity:

1. Keeping back information up to date assuming that forward local traces and and back traces execute atomically, that is, happen instantaneously without overlap.
2. Accounting for a non-atomic forward local trace: While the forward trace is computing back information, a client may change the object graph, or a back trace may visit the site.
3. Accounting for a non-atomic back trace: Even if back information is kept up to date at each site, a back trace might see an inconsistent distributed view. This is because a back trace reaches different sites at different times, and because the participants of a multi-server transaction may prepare it at different times.

We address each of these parts below.

6.3.3.1 Updating Back Information

Back information may change due to reference creation and deletion. We ignore deletions since doing so does not violate safety; also, ignoring deletions does not violate completeness because deletions are reflected in the back information computed during the next local trace. On the other hand, reference creations must be handled such that a back trace does not miss a new path to a suspect. For example, Figure 6.12 shows the creation of a reference to z followed by the deletion of a reference in the old path to z. If site T does not update its back information to reflect the new reference, but site U does a local trace to reflect the deletion, a subsequent back trace from g might return Garbage.

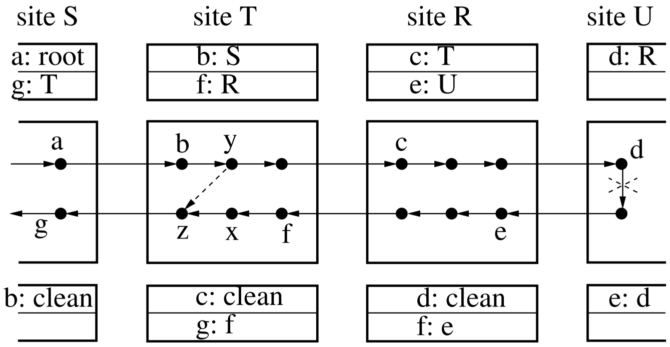

Figure 6.12: Reference modifications (dotted lines).

Reference creations are handled as follows. In general, creating a reference involves copying some reference z contained in some object x into some object y.[2] Suppose x, y, and z are in sites X, Y, and Z respectively, some or all of which may be the same. If X is the same as Y, we say that the reference copy is local; otherwise, we say it is remote. We discuss these situations separately below.

[2]The creation of a reference to a *new* object z may be modeled as copying a reference to z from a special persistent root at the site.

Local Reference Copy

A local reference copy is tricky to account for because it may change the reachability from inrefs to outrefs although none of the objects involved might be associated with inrefs or outrefs. In Figure 6.12, creating a reference to z makes outref g reachable from inref b. The insight behind the solution is that in order to create a reference to a suspect z, the client must have read a path of references from some global root to z; this path must include some suspected inref on X. In the example shown, the client must have read f. Therefore, the following *suspect barrier* is used:

> If a transaction has read an object for which there is a suspected inref, the inref and its outreach is flagged clean.

Iorefs so cleaned remain clean until the site does the next local trace. (Also, these iorefs are not removed until then.) If a back trace visits such an ioref before the next local trace, it will return Live. The back information computed in the next local trace will reflect the paths created due to the new reference. Thus, if a back trace visits the site after its next local trace, it will find these paths.

The suspect barrier is implemented by examining the set of objects accessed by a transaction at prepare time. (For a description of the suspect barrier in an RPC-based system, where references are transferred between sites by direct messages, see [ML97a].) Since suspected objects are highly likely to be garbage, the suspect barrier is rarely invoked.

The suspect barrier maintains the *local safety invariant*:

> For any suspected outref o, $o.inreach$ includes all inrefs o is locally reachable from.

We show below that the suspect barrier preserves the local safety invariant. We use an auxiliary invariant: For any suspected outref o, $o.inreach$ does not include any clean inref. This invariant holds right after a local trace, and the suspect barrier preserves it because whenever it cleans any inref i, it cleans all outrefs in $i.outreach$. (Inreaches and outreaches are consistent since they are different views of the same information.)

Proof of Local Safety Suppose a reference from x to z is copied into y. This affects only the outrefs reachable from z: any such outref o may now be reachable from more inrefs than listed in $o.inreach$. This is illustrated in Figure 6.13. (If z is an outref itself, then o is identical to z.) We show that all outrefs such as o must be clean after the mutation.

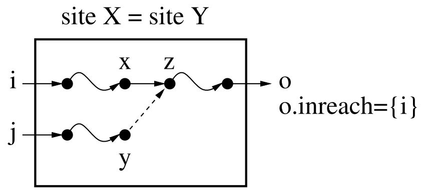

Figure 6.13: Local copy.

Suppose object x was reachable from some inref i before the modification. Since x pointed to z, outref o was reachable from inref i as well. Therefore, if the local safety invariant held before the modification, either o was clean or $o.inreach$ included i. Suppose $o.inreach$ included i. If x was reachable from a *clean* inref i, the auxiliary invariant implies that o must be clean. Otherwise, the suspect barrier must have been applied to some suspected inref i that x was reachable from. The barrier would have cleaned all outrefs in $i.outreach$, which includes o since $o.inreach$ includes i. In either case, o must be clean after the modification.
□

It is important not to clean inrefs and outrefs unnecessarily in order that cyclic garbage is collected eventually. We ensure the following invariant for completeness:

> An inref is clean only if its estimated distance is less than the suspicion threshold or if it was live at the last local trace.
> An outref is clean only if it is reachable from a clean inref or if it was live at the last local trace at the site.

Proof of Completeness The invariant is valid right after a local trace. Thereafter, we clean inrefs and outrefs only due to the suspect barrier. Suppose applying the barrier cleans inref i and outref o in the outreach of i. Then i must be live when the barrier was applied. Since o was reachable from i when the last local trace was performed, o must be live at that time as well.
□

Remote Reference Copy

If a reference to z is copied from x at site X into y at another site Y $(X \neq Y)$, we handle it in one of the following ways. First, consider the case when object z is in site Y $(Z = Y)$, as shown in Figure 6.14. Since X has a reference to z, Y must have an inref for z. Y preserves the local safety invariant by cleaning z and $z.outreach$. Thus the *suspect barrier* is extended as follows: If an object modified by a transaction contains a reference to some object for which there is a suspected inref, the inref and its outreach is cleaned.

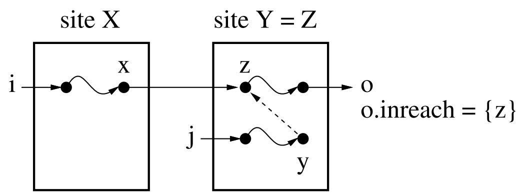

Figure 6.14: First case of a remote copy.

Now consider the case when object z is not in site Y, as shown in Figure 6.15. In this case, if Y has a clean outref for z, no change is necessary. If Y has a suspected outref for z, Y preserves local safety by cleaning the outref for z. Thus the *suspect barrier* is extended as follows: If an object modified by a transaction has a reference to some object for which there is a suspected outref, the outref is cleaned.

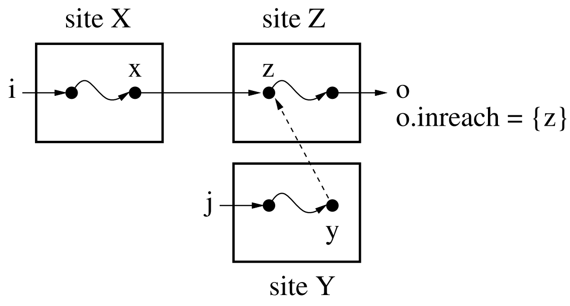

Figure 6.15: Second case of a remote copy.

Finally, if Y has no outref for z, Y creates a clean outref for z and sends an add message to Z, as described in the add protocol in Section 5.3. This is only case that results in the creation of a new inter-site reference. Here, a potential problem is that back information is not up to date until the add message adds Y to the source list of z. However, until this happens, some site must contain z in its guardlist. Therefore, iorefs with guardlist entries are treated as clean. If a back trace visits an inref z before the add message reaches there, it will return Live from the clean outref. If the back trace happens later, it will find all sites containing z. Thus, the following *remote safety invariant* is maintained:

> For any suspected inref i, either
> $i.sources$ includes all sites containing i, or
> $i.sources$ includes at least one site with a clean outref for i.

The site Y to which the reference z is transferred might be a client that has stored z in a handle or a volatile object. The remote safety invariant applies to this case. Further, when the add message from the client arrives, the client is added to the source list of the inref, making the inref a global root and therefore keeping the inref clean.

6.3.3.2 Non-Atomic Forward Local Tracing

Modifications may change the object graph while the local trace is computing back information. The computed information must account for these modifications safely. Further, a back trace may visit a site while it is computing back information.

During a local trace, we keep two versions of the back information: the old version retains the information from the previous local trace while the new version is being prepared by the current trace. Back traces that visit a site during a local trace use the old version. When the local trace completes, the new version replaces the old atomically.

The new version of back information must be fixed to account for modifications made during the local trace. The old-view approach for local tracing provides a consistent, albeit old, snapshot of the object graph (Section 3.4). The back information computed by such a trace can be fixed by cleaning the iorefs that were cleaned in the old version during the local trace. We also clean the outreach of any inref so cleaned, should the outreach contain additional outrefs not present in the old version. This maintains the local safety invariant.

If the new-view approach were used for local tracing, another mechanism would be necessary to fix the computed back information. Specifically, if a new reference is created to a suspected object z during the trace, the outreach of z computed by the trace must be cleaned.

6.3.3.3 Non-Atomic Back Tracing

So far we assumed that a back trace happens instantaneously such that it sees consistent back information at various sites. In practice, a back trace may overlap with modifications, making it unsafe even if back information is kept up to date at each site. For example, in Figure 6.16, a back trace from g may visit site T *before* the new reference from y to z is created, so that it does not see the updated back information at T. Yet the trace may visit U *after* the indicated reference is deleted and after U has done a local trace to reflect that in the back information.
We use the following *clean rule* to ensure a safe view:

> If an ioref is cleaned while a back trace is active there, the return value of the trace is set to Live.

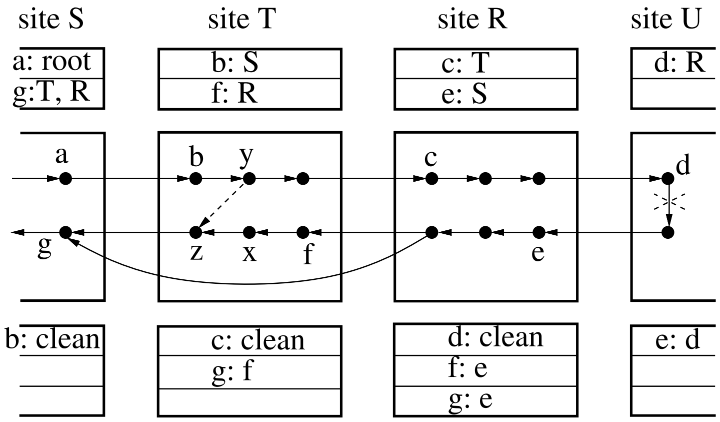

Figure 6.16: A non-atomic back trace may see an inconsistent view.

The clean rule ensures that if there is any overlap in the periods when an ioref is clean and when a trace is active there, the trace will return Live. For a proof of safety of the clean rule in an RPC-based system, see [ML97a]. Below we discuss safety in a transactional system.

An additional problem arises in transactional systems because servers may prepare a transaction at different times. The problem is illustrated in the context of Figure 6.16. Here, a transaction, τ_1, creates the reference from y to z and a later transaction, τ_2, removes the old path to z by deleting a reference in U. The following scenario shows that it is possible for a back trace to miss both the old and the new paths to z. Since τ_1 read the old path to z, both R and T are among the participants of τ_1. If R prepares τ_1 before T, a back trace from g may miss the modification made by τ_1, yet see the modification made by τ_2 as follows:

1. R receives the prepare message for τ_1. It applies the suspect barrier and cleans inref e and $e.outreach$. However, an immediate local trace at R reverts them to their suspected status.
2. The back trace from inref g forks two parallel branches: branch 1 to site T and branch 2 to site R. Branch 2 visits inref e first, so when branch 1 reaches inref e, it returns Garbage.
3. T receives the prepare message for τ_1. It applies the suspect barrier and cleans inref f and outref z. However, branch 1 has already returned from site T.
4. Site U prepares and commits τ_2. It then does a local trace that removes outref e.
5. Branch 2 of the back trace reaches site U, finds no outref for e, and returns Garbage.

The solution to this problem is to extend the suspect barrier such that iorefs cleaned due to a transaction are not reverted or removed until the transaction is known to have committed or aborted. Therefore, iorefs cleaned due to the suspect barrier are recorded in a *cleanlist* along with the id of the transaction which invoked the suspect barrier. An entry is removed from the cleanlist only after the transaction is known to have committed or aborted; the corresponding ioref remains clean until the site conducts a local trace after the entry is removed. The extended suspect barrier weakens the completeness invariant given in Section 6.3.3.1, but it does not compromise completeness assuming transactions are eventually known to be committed or aborted. Finding out that the transaction committed or aborted requires that the site be sent a phase-2 message about the transaction even if it were a read-only participant in the transaction (such as R in τ_1). However, such occurrences are limited to the cases when the suspect barrier is invoked, which happens infrequently.

Below we prove that the extended suspect barrier and the clean rule provide a safe view of a local copy to a non-atomic back trace. The case of a remote copy is discussed later.

Proof of Safety for Local Copy and Non-Atomic Back Traces

Suppose a transaction τ copies a reference to a suspected object z into object y. To do this, the transaction must have read a chain of objects such as that shown in Figure 6.17. Here, the chain passes through sites $S_0, S_1, \ldots, S_{k-1}, S_k, \ldots, S_n$. In this chain, i_k and o_k are the inref and the outref on site S_k (outref o_{k-1} corresponds to the inref i_k). The chain shown in the figure is only a suffix of the chain from a global root such that iorefs i_0 and o_0 are clean while the rest are suspected.

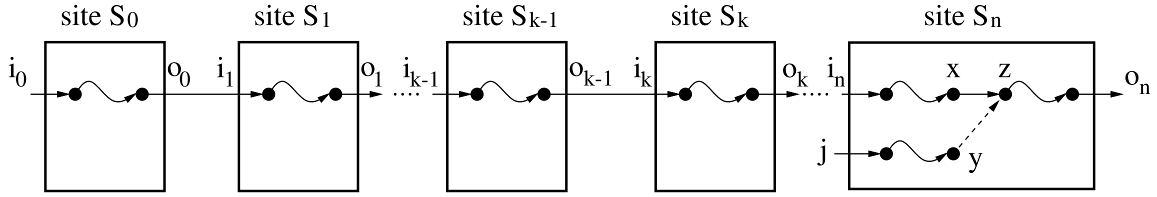

Figure 6.17: Non-atomic back trace and a local copy.

The new reference to z may affect back traces that visit an outref o_n reachable from z. Consider the following cases:

- If the back trace visits o_n after τ has committed and S_n has done a local trace, it will find the new path created by the modification.
- If the back trace visits o_n after S_n has prepared τ but before τ has committed and S_n has done a local trace, o_n will be clean due to the extended suspect barrier. So the back trace will return Live.
- If the back visits o_n before S_n prepares τ, by induction on Lemma 1 below, it will return Live or visit o_0 before S_0 prepares τ. Since o_0 is clean, the back trace will return Live. □

Lemma 1 If a back trace visits o_k before S_k prepares τ, it will return Live or visit o_{k-1} before S_{k-1} prepares τ.

Proof of Lemma 1

If the back trace visits o_k before S_k prepares τ, there are the following possibilities:

1. The back trace is still active at o_k when S_k prepares τ. The clean rule will cause the trace to return Live.
2. The back trace visits all inrefs in the inreach of o_k before S_k prepares τ. Further, i_k must be among these inrefs because the path from i_k to o_k could not have been deleted if S_k is to validate τ later.

If the back trace visits i_k before S_k prepares τ, there are the following possibilities:

1. The back trace is still active at i_k when S_k prepares τ. The clean rule will cause the trace to return Live.
2. The back trace visits all outrefs on the source sites of i_k before S_k prepares τ. Since S_{k-1} could not have committed τ before S_k prepares τ, there are two possibilities:
   (a) S_{k-1} has not prepared τ. In this case S_{k-1} must contain the outref o_{k-1} because the reference could not have been deleted if S_{k-1} is to validate τ later. Therefore, from the remote safety invariant, the back trace would either visit o_{k-1} or return Live from a clean outref on some other site.
   (b) S_{k-1} has prepared τ but not committed it. In this case S_{k-1} must have a clean outref for o_{k-1} because of the extended suspected rule, so the back trace will return Live. □

Safety for Remote Copy and Non-Atomic Back Traces

The case for a remote reference copy is shown in Figure 6.18. Here, a transaction τ copies a reference to z at site Z into object y at site Y. If a back trace visits o_n before S_n commits τ and conducts a local trace, it would return Live as in the case of a local copy. However, if the back trace visits o_n later, it might return erroneously. The error might occur because of an ill-informed selection of the guardian site during the add protocol (see Section 5.3). Suppose that prior to τ, sites S_n and G contained outrefs for z. Further, suppose G is a participant in τ and the coordinator of τ selects G as the guardian for reference z stored in Y. Now, a back trace from i_{n+1} will branch to sites S_n and G. The branch to G might reach there before G prepares τ and return Garbage. The branch to S_n might visit o_n after S_n has committed τ and return Garbage too (because the old path to z might be deleted).

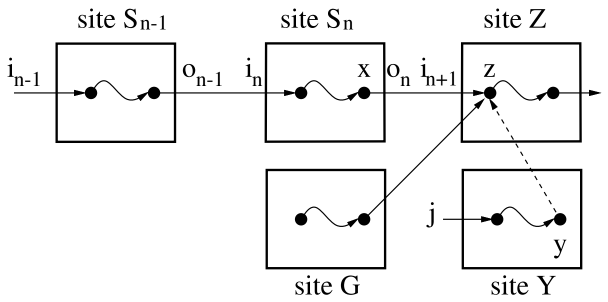

Figure 6.18: Non-atomic back trace and a remote copy.

The solution is to extend the clean rule as follows: If an ioref is cleaned while a back trace has visited it (but is not active there anymore), the ioref and the id of the back trace are recorded in a *guard-cleanlist*. The entry is removed from the guard-cleanlist when the final outcome of the back trace is known. In the above example, when G adds z to its guardlist, it will clean outref z and add it to its guard-cleanlist since a back trace has visited outref z. (Even if outref z were cleaned due to some other reason, it would be added to the guard-cleanlist.) Further, when a site adds an outref to its guardlist, it checks whether the reference is in its guard-cleanlist. If so, it must send a *clean* message to the target site to clean the corresponding inref and receive an acknowledgement before responding to the coordinator. The clean message to the target is sent in the foreground, but that is acceptable since such cases are highly unlikely. In the above example, G will send a clean message to Z; when Z receives this message it cleans inref z. If the back trace is still active at inref z, the trace will return Live. Otherwise, the back trace must have visited o_n while τ is not committed, so then it also will return Live.

6.3.4 Back Tracing Through Partitions

The description so far assumed that forward local traces compute back information at the site level. However, since a server traces its partitions independently, each partition trace must compute back information for the suspected objects in the partition. In particular, for each suspected reference o in the outset of the partition, the trace must compute the references in the inset of the partition from which o is reachable through intra-partition references.

There are two possibilities for using partition-level back information. The first is a straightforward adaptation of using site-level back information: a back trace would step from one partition to another rather than between sites. The second possibility is to consolidate partition-level information into site-level information, so a back trace would go from one site to another directly as assumed in previous sections. However, this possibility requires recomputing the site-level information whenever the partition-level information changes. We have not explored the implications of these possibilities. They must be studied further to adopt back tracing in a partitioned system.

Since back information must be maintained at partition level, back tracing could be used to collect inter-partition cycles within a site. Collecting inter-partition cycles this way requires that such cycles are identified by the suspect-finding heuristic. The distance heuristic can be changed to identify inter-partition cycles by associating a positive weight with each inter-partition reference. However, this approach would result in many more suspected objects, since there is likely to be much more inter-partition cyclic garbage than inter-site cyclic garbage. Similarly, the time and space overhead of computing and maintaining back information and running back traces would increase substantially. Therefore, we chose to limit the use of distance heuristic and back tracing for collecting inter-site cycles. This choice was also encouraged by the fact that site-marking is an acceptable solution for collecting cycles within a site since there is no need to tolerate independent failures. In general, site-marking could be extended to a cluster of tightly-coupled computers and locality-preserving techniques could be used to collect garbage between clusters.

6.3.5 Summary of Back Tracing and Comparison with Migration

Unlike forward tracing from the global roots, back tracing from a suspected object has locality because the trace visits only the sites the suspect is reachable from. A back trace is started from an outref when its distance increases above a threshold. The thresholds are updated such that live suspects stop generating back traces eventually, while garbage suspects keep generating back traces until they are collected. Thus, back tracing is collects all inter-site garbage cycles.

The message overhead of back tracing is less than that of any other scheme for collecting inter-site garbage cycles that has locality. If a cycle resides on C sites and has E inter-site references, $2E + C$ messages are sent; further, these messages are only a few words. The message overhead is much less than that of migration, which must transmit batches of suspected objects and update references to migrated objects. The processing overhead of back tracing is low because back tracing leaps between inrefs and outrefs rather than trace individual references. Further, the computation of back information is folded into the forward local trace without rescanning objects. The back information comprises inreaches of outrefs, which takes $O(n_i \times n_o)$ space, where n_i and n_o are the number of suspected inrefs and suspected outrefs. The space overhead is more than migration, which consists of destination information in suspected inrefs and outrefs ($O(n_i + n_o)$), and storage of surrogates for migrated objects. However, the space overhead of back tracing is less than other schemes with locality [Sch89, LC97]. These other schemes require complete local-reachability information between inrefs and outrefs—not just suspected ones, and they store additional path information for each inref.

Back traces are conducted concurrently with modifications and forward local traces. We use two barriers to keep back information conservatively safe, yet we ensure that the barriers do not prohibit the collection of garbage cycles. These barriers are applied when a transaction reads an object with a suspected inref or stores a reference to such an object. Keeping the back information up to date at each site is not sufficient because a back trace may see an inconsistent distributed view, but we ensure that back traces sees a safe distributed view.

Computation of back information and handling concurrent modifications make back tracing more complex than migration. The complexity of migration is hidden in the implementation of the underlying layer that migrates objects and updates references to migrated objects. However, back tracing is likely to be more suitable for most systems because its network overhead is much less and it is not limited by autonomy concerns against migrating objects.

6.4 Related Work

Previous schemes for collecting inter-site garbage cycles may be categorized as follows.

Global Tracing

A complementary global trace is conducted periodically to collect cyclic garbage, while other garbage is collected more quickly by local tracing [Ali85, JJ92]. Termination of the global trace is detected using the model for diffusive computation [Aug87, DS89]. The drawback of global tracing is that it may not complete in a system with a large number of faulty sites.

Hughes's algorithm propagates *timestamps* from inrefs to outrefs and collects objects timestamped below a certain global threshold [Hug85]. The persistent roots always have the current time, and a global algorithm is used to compute the threshold. The advantage of using timestamps over mark bits is that, in effect, multiple marking phases can proceed concurrently. However, a single site can hold down the global threshold, prohibiting garbage collection in the entire system.

Central Service

Beckerle and Ekanadham proposed that each site send inref-outref reachability information to a fixed site, which uses the information to detect inter-site garbage cycles [BE86]. However, the fixed site becomes a performance and fault tolerance bottleneck.

Ladin and Liskov proposed a logically central but physically replicated service that tracks inter-site references and uses Hughes's algorithm to collect cycles [LL92]. The central service avoids the need for a distributed algorithm to compute the global threshold. However, cycle collection still depends on timely correspondence between the service and *all* sites in the system.

Subgraph Tracing

The drawbacks of global tracing can be alleviated by first delineating a *subgraph* of all objects reachable from an object suspected to be cyclic garbage. Another distributed trace is then conducted within this subgraph; this trace treats all objects referenced from outside the subgraph as roots. All subgraph objects not visited by this trace are collected. Note that a garbage cycle might point to live objects, and the associated subgraph would include all such objects. Thus, the scheme does not possess the locality property.

Lins *et al.* proposed such a scheme as "cyclic reference counting" in a system that used reference counting for local collection instead of local tracing [LJ93]. This scheme requires two distributed traces over objects in a subgraph. Jones and Lins improved the scheme such that multiple sites could conduct traces in parallel, but it required global synchronization between sites [JL92].

Group Tracing

Another method to alleviate the drawbacks of global tracing is to trace within a *group* of selected sites, thus collecting garbage cycles within the group. A group trace treats all references from outside the group as roots.

The problem with group tracing is configuring groups in order to collect all inter-site cycles. Lang *et al.*. proposed using a tree-like hierarchy of embedded groups [LQP92]. This ensures that each cycle is covered by some group, but the smallest group covering, say, a two-site cycle may contain many more sites. Further, the policy for forming and disbanding groups dynamically is unclear.

Maeda *et al.* proposed forming groups using subgraph tracing [MKI^{+}95]. A group consists of sites reached transitively from some objects suspected to be cyclic garbage. This work was done in the context of local tracing and inter-site weighted reference counting. Rodrigues and Jones proposed an improved scheme in the context of inter-site reference listing [RJ96]. One drawback of this approach is that multiple sites on the same cycle may initiate separate groups simultaneously, which would fail to collect the cycle. Conversely, a group may include more sites than necessary because a garbage cycle may point to chains of garbage or live objects. Another problem is that group-wide tracing might never collect all cycles: Since a group-wide trace is a relatively long operation involving multiple local traces, it is not feasible to reconfigure groups to cover all garbage cycles in a system with many sites.

Schemes with Locality

Few schemes for collecting cyclic garbage have the locality property. The oldest among these is migrating a suspected distributed garbage cycle to a single site [Bis77]. Since migration is expensive, it is crucial to use a good heuristic for finding suspects; we proposed the distance heuristic in this context earlier [ML95]. However, some systems do not support migration due to security or autonomy constraints or due to heterogeneous architecture. Those that do must patch references to migrated objects. Shapiro *et al.*. suggested *virtual* migration [SGP90]. Here, an object changes its logical space without migrating physically. However, a logical space may span a number of sites, so a local trace must involve inter-site tracing messages.

Schelvis proposed forwarding local-reachability information along outgoing inter-site references [Sch89]. Objects are identified by unique timestamps to detect concurrent mutator activity. This algorithm is intricate and difficult to understand; however, some of its problems are apparent. The algorithm requires full reachability information between all inrefs and outrefs (not just suspected ones). An inref i records a set of *paths* instead of source sites; each path indicates a sequence of inrefs leading to i. Collecting a cycle located on N sites might take $O(N^3)$ messages. Recently, Louboutin presented an improved scheme that sends only $O(N)$ messages [LC97]. However, it too requires full inref-outref reachability information, and its space overhead is even larger: each inref i stores a set of vector timestamps; each vector corresponds to a path i is reachable from.

Back tracing was proposed earlier by Fuchs [Fuc95]. However, this proposal assumed that inverse information was available for references, and it ignored problems due to concurrent mutations and forward local traces. A discussion on back tracing, conducted independently of our work, is found in the archives of the mailing list gclist@iecc.com at ftp://iecc.com/pub/gclist/gclist-0596.

6.5 Summary

This chapter presented the first practical scheme that preserves the locality property in collecting inter-site garbage cycles. Thus, the scheme is suitable for use in a system with many sites. Such a scheme became possible by using a combination of two parts—either of which are incorrect or impractical to use by itself.

The first part identifies suspected cyclic garbage, but it is not safe in that it might suspect live objects. Nonetheless, it can be made arbitrarily accurate in suspecting only cyclic garbage by trading quickness with accuracy. This part is based on estimating the minimum number of inter-site references from global roots to objects. It has very low overhead since it is piggybacked on underlying partition traces.

The second part checks if the suspects are in fact garbage. It is responsible for restoring safety. Since it needs to check only the suspects identified by the first part, it can use techniques that would be too costly if applied to all objects but are acceptable if applied only to suspects. We presented two such techniques: one based on migrating a suspected cycle to a single site, and the other based on tracing backwards from suspects in search of global roots. The first is simpler, but the second has lower overheads and applies to systems that do not migrate objects.

Chapter 7

Conclusions

This thesis has presented a comprehensive design for garbage collection in a large, distributed object store. Below we give a summary of the overall design and discuss its significance. We also give guidelines for how the design may be evaluated and we indicate some directions for future research. Although the design is presented as a whole, most of the techniques underlying it are applicable separately.

7.1 The Overall Design

The thesis applies to a large object store that provides reliable access to shared and persistent data. Such an object store might span thousands of sites, and each site might store billions of objects on disk. The thesis provides scalable techniques for complete, timely, efficient, and fault tolerant garbage collection in such a system. Scalability is achieved by partitioning garbage collection at two levels:

1. Each site traces its objects independently of the others. This provides the locality property: the collection of a chain of garbage objects involves only the sites on which it resides.
2. The disk space at each site is divided into partitions that are traced one at a time in main memory. This avoids disk thrashing.

To trace a site independently of other sites, and a partition independently of other partitions, incoming references from other sites and other partitions are treated as roots. This introduces two problems:

Efficiency Maintaining up-to-date information about inter-site and inter-partition references can delay applications and increase utilization of memory, disk, and network.

Completeness Treating inter-site and inter-partition references as roots fails to collect cycles of garbage objects spanning multiple sites or multiple partitions.

Solutions to these problems have been proposed earlier in single-site or distributed systems, but they do not scale to many partitions or many sites. For example, they are not efficient in managing large numbers of inter-partition references, and they do not preserve the locality property in collecting inter-site garbage cycles. This thesis provides scalable and fault tolerant techniques for solving these problems.

7.1.1 Recording Inter-partition and Inter-site References

Information about both inter-partition and inter-site references is maintained persistently. However, we use different techniques to record the two kinds of references to match efficiency and fault-tolerance requirements.

The main issue in recording inter-partition references is to keep the disk usage low. Two new techniques are used to manage a potentially large number of such references. First, these references are recorded in translists based on their source and target partitions. Translists are shared so that dropping unnecessary references from the outset of a source partition removes them from the insets of the target partitions. Second, new inter-partition references are found by scanning the log of modified objects lazily, and these references are recorded in in-memory delta lists. Delta lists defer and batch disk accesses to update translists. Our experiments show that delta lists result in substantial reduction in disk-time usage when not enough memory is available to cache all translists in use, and that they result in negligible degradation when enough memory is available.

The main issue in recording inter-site references is to avoid extra messages—especially foreground messages that might delay applications. Copies of an inter-site translist are maintained on both the source and the target sites: The outlist at the source site avoids unnecessary add and remove messages, and the inlist at the target site provides local access to the roots. One site treats another as a single partition so that each may configure its disk partitions without affecting others.

Inter-site references in a system that caches objects at client sites include both inter-server references and client-to-server references. New inter-server references are processed when transactions commit. They cannot be processed lazily like inter-partition references because different sites conduct traces asynchronously, while different partitions at a site are traced synchronously. The add protocol for new inter-server references piggybacks some messages on the commit protocol and sends others in the background. Thus it does not increase the commit latency with extra foreground messages. While inter-server references are expected to be relatively infrequent due to clustering of objects, a client might cache millions of references to server objects. Therefore, we use a new technique to record only a minimal subset of references in the client cache. This technique exploits the cache coherence protocol.

Garbage collection tolerates server and client crashes. Servers store translists on disk so that inter-partition and inter-site information is recovered quickly after crashes. A client might crash unrecoverably or might appear to have crashed due to communication problems. Therefore, the system uses the ban protocol that allows servers to discard information about such clients safely.

7.1.2 Collecting Inter-partition and Inter-site Garbage Cycles

We also use different techniques to collect inter-partition and inter-site garbage cycles to match fault-tolerance and efficiency requirements. In particular, a complementary marking scheme is used to collect garbage cycles within a site, but a global marking scheme to collect cycles between sites would not be fault tolerant.

Inter-partition garbage cycles on the same site are collected using a site-wide marking scheme. These marks are propagated incrementally through partition traces such that it adds little overhead. This is the first such scheme that does not delay the collection of non-cyclic garbage and terminates correctly in the presence of modification. A phase of site-wide marking might involve many partition traces, but that is acceptable provided inter-partition cyclic garbage is generated slowly compared to other garbage.

Inter-site garbage cycles are collected using a scheme that preserves locality so that collecting a cycle involves only the sites it resides on. A two-part method is used to enable such a scheme. The first finds objects that are highly likely to be cyclic garbage but might be live. This part uses a new technique—the distance heuristic—to find suspects with an arbitrarily high accuracy. The distance heuristic has very little overhead since it is folded into local tracing. The second part checks if the suspects are in fact garbage. This part may either migrate suspected cycles to single sites or check if they are live by tracing back from them in search of roots. Back tracing is likely to be more suitable than migration in most systems because its network overhead is much less and it is not limited by autonomy concerns against migrating objects. We provide practical techniques to compute the information required for back tracing and to handle concurrent modifications.

7.2 Guidelines for Performance Evaluation

This thesis does not provide a comprehensive performance evaluation of the proposed design. In this section, we provide guidelines for such an evaluation in future. The evaluation has two goals: evaluate the effect of garbage collection as a whole, and evaluate the effects of the techniques proposed in this thesis.

The net gain from garbage collection is that applications speed up due to two factors:

1. Higher spatial density of useful (live) objects, resulting in better utilization of memory caches and of disk and network bandwidth.
2. Reduction of time spent waiting for free space to allocate new objects. In some systems, an application fails if no free space is available; this is equivalent to an infinite wait.

Evaluating the net gain as a single number would require a long-term study of the system and would depend on many different characteristics of the workload. While such a study is useful, a more insightful study is to evaluate the various costs and benefits that compose the net gain under various workload and system parameters.

7.2.1 Metrics

The costs can be divided into background costs, which delay applications through increased utilization of resources, and foreground costs, which stall applications directly. The following metrics are useful in measuring the costs of our garbage collector:

Background Costs

- Disk-time usage for fetching pages of partitions and flushing compacted pages.
  This depends on the frequency of partition traces, the size of partitions, and the occurrence of pages with no garbage.
- Disk-time usage for accessing and updating inter-partition information.
  This depends on the frequency of creation and removal of inter-partition references. (In Section 3.5, we measured the usage for adding references, but not for removing references from translists.)
- Memory usage for tracing partitions and storing partition information.
  This is mostly controlled by the size of partitions.
- Processor usage in tracing partitions and processing the log.
  This depends on the frequency of partition traces, the size of partitions, the density of contained objects and references, and the frequency of modifications.
- Network usage for add, remove, guard, distance-update, and other messages.

Foreground Costs

- Average time for which the collector halts applying transactions before tracing a partition (as described in Section 3.4).

- Average time for a server to obtain a license from a client's proxy when the client first contacts the server (as described in Section 5.4).

The benefits can be evaluated by measuring the average rate at which space is reclaimed. The space reclaimed can be categorized as either reclaimed by partition traces or by site-marking. It is difficult to compute the space reclaimed by collecting inter-site garbage cycles, since this part does not reclaim objects itself and relies on partition traces to delete objects. However, an estimate of the space reclaimed in this way can be found by adding up the space of all suspected objects, since such objects are highly likely to be cyclic garbage. Thus, the rate at which inter-site cyclic garbage is reclaimed can be estimated as the rate of deletion of suspected objects.

7.2.2 Workload Parameters

Ideally, the performance should be evaluated for a large number of representative real applications. Nonetheless, evaluation for synthetic workloads is insightful because these workloads can provide independent control over different parameters. Some parameters of interest in our garbage collector are the following:

- Spatial distribution of references:
  - Inter-server references and cycles.
  - Inter-partition references and cycles.
  - Client handles.
  - References to persistent objects in volatile objects at clients.
- Temporal distribution of modifications:
  - Creation of new objects.
  - Creation of new inter-partition and inter-site references.
  - Generation of local garbage.
  - Generation of inter-partition garbage chains and cycles.
  - Generation of inter-server garbage chains and cycles.
- Number of clients and the average number of servers each client uses.

7.2.3 System Parameters

Finally, the performance of the garbage collector depends on various internal system parameters:

- Scheduling priority of the collector thread relative to other service threads.
- The amount of memory space allocated to the collector.
- Average size of partitions.
- Rate at which partitions are traced.
- Policy for selecting partitions.
- Rate at which partitions are reconfigured to match reference distribution.
- The setting of distance threshold for suspecting inter-site garbage cycles.
- The setting of threshold to initiate migration or back tracing for checking suspects.

7.3 Directions for Future Work

This thesis does not cover all issues in partitioned garbage collection and it covers some of them only partially. Below we list some areas that require further work. We also discuss the applicability of some techniques used in this thesis to fields outside of garbage collection.

Partition Configuration

Partitions must be configured such that there are few inter-partition references. This is important for timely garbage collection as well as reducing the amount of inter-partition information. Configuring partitions involves grouping the disk pages into disjoint sets such that each set can fit in a small fraction of the main memory. Finding the configuration with the least number of inter-partition references is likely to require work that is exponential in the number of pages; therefore, effective heuristics are needed instead. Ideally, the garbage collector should reconfigure partitions dynamically to adapt to the current inter-page references. Previous research on clustering objects dynamically is relevant to this work.

Partition Selection for Tracing

Garbage collection can be more efficient by carefully selecting partitions for tracing. It is important to use heuristics to find partitions that contain a lot of garbage or whose trace will lead to the collection of a lot of garbage in other partitions. In Section 3.2, we suggested the reduced-inset heuristic for selecting partitions. Cook *et al.* have suggested some other heuristics for selecting partitions [CWZ94]. However, more work is needed to devise sophisticated heuristics for prompt collection of both partition-local and inter-partition garbage.

Delta Lists

We used delta lists to defer and batch additions to translists. This technique can be generalized for efficient implementation of sets that are augmented more frequently than they are searched or enumerated. Elements added to such a set can be logged and stored in an in-memory delta list without fetching the set from the disk. Previous research on type-specific concurrency control is relevant to this work.

Client Crashes

We presented the ban protocol to handle client crashes that are indistinguishable from communication problems. This protocol uses a proxy server per client that must be contacted whenever the client approaches a new server. This approach is not suitable if clients browse over many different servers; a light-weight protocol is desirable in that case. We have studied one such protocol based on leases [Mah93b]. Here, a client must periodically renew leases with servers that have stored information for it.

A ban protocol may be applied to fields outside of garbage collection where it is necessary for all servers to ban a client before discarding state about the client, *e.g.*, in on-line financial services. However, the ban protocol does not tolerate malicious clients; more work is required to tolerate such clients.

Dynamic Tuning of Thresholds

The heuristics for collecting inter-site garbage cycles involve thresholds for suspecting garbage and initiating migration or back tracing. While the scheme does not depend on these thresholds for safety or completeness, the thresholds determine how quickly garbage cycles are collected and how much work is wasted due to poor guesses. Ideally, these thresholds should be set dynamically to achieve the desirable mix. Further, the thresholds should be customized to sub-parts of the system—possible to individual inrefs and outrefs.

Back Tracing Through Partitions

A back trace in a partitioned server may proceed by stepping from one partition to another using partition-level information. An important optimization would be to consolidate partition-level back information into site-level information such that a back trace may advance at the site-level. However, further techniques are needed maintain such information and to enforce the various safety barriers.

Distance Heuristic

The distance heuristic finds objects that are likely to be cyclic garbage with an arbitrarily high accuracy. The estimated distances of garbage cycles keep increasing without bounds because such cycles act as positive-feedback loops without any path from a global root to hold down the increase. We expect that a similar heuristic can be used in other problems based on cyclic references in distributed systems, such as distributed deadlocks.

Bibliography

[ABC$^+$83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[Ady94] A. Adya. Transaction management for mobile objects using optimistic concurrency control. Master's thesis, Massachusetts Institute of Technology, 1994.

[AFFS95] L. Amsaleg, P. Ferreira, M. Franklin, and M. Shapiro. Evaluating garbage collection for large persistent stores. In *Addendum to Proc. OOPSLA Workshop on Object Database Behavior*. ACM Press, 1995.

[AGF95] L. Amsaleg, O. Gruber, and M. Franklin. Efficient incremental garbage collection for workstation–server database systems. In *Proc. 21st VLDB*. ACM Press, 1995.

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proc. SIGMOD*, pages 23–34. ACM Press, 1995.

[Ali85] K. A. M. Ali. Garbage collection schemes for distributed storage systems. In *Proc. Workshop on Implementation of Functional Languages*, pages 422–428, 1985.

[Aug87] L. Augusteijn. Garbage collection in a distributed environment. In J. W. de Bakker, L. Nijman, and P. C. Treleaven, editors, *PARLE'87 Parallel Architectures and Languages Europe*, volume 258/259 of *Lecture Notes in Computer Science*, pages 75–93. Springer-Verlag, 1987.

[Bak78] H. G. Baker. List processing in real-time on a serial computer. *CACM*, 21(4):280–94, 1978.

[Bak93] H. G. Baker. 'Infant mortality' and generational garbage collection. *ACM SIGPLAN Notices*, 28(4), 1993.

[BE86] M. J. Beckerle and K. Ekanadham. Distributed garbage collection with no global synchronisation. Research Report RC 11667 (#52377), IBM, 1986.

[BEN$^+$93] A. Birrell, D. Evers, G. Nelson, S. Owicki, and E. Wobber. Distributed garbage collection for network objects. Technical Report 116, Digital Systems Research Center, 1993.

[Bev87] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187. Springer-Verlag, 1987.

[Bis77] P. B. Bishop. Computer systems with a very large address space and garbage collection. Technical Report MIT/LCS/TR–178, MIT, 1977.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), 1984.

[CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proc. SIGMOD*, pages 12–21. ACM Press, 1993.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-graned sharing in a page server OODBMS. In *Proc. SIGMOD*, Minneapolis, MN, 1994. ACM Press.

[CKWZ96] J. E. Cook, A. W. Klauser, A. L. Wolf, and B. G. Zorn. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *Proc. SIGMOD*. ACM Press, 1996.

[CWZ94] J. E. Cook, A. L. Wolf, and B. G. Zorn. Partition selection policies in object databases garbage collection. In *Proc. SIGMOD*. ACM Press, 1994.

[DLMM94] M. Day, B. Liskov, U. Maheshwari, and A. Myers. References to remote mobile objects in Thor. *Letters on Programming Languages and Systems*, 2(1–4):115–126, 1994.

[DS89] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11, 1989.

[EH84] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transaction on Database Systems*, 9(4):560–595, 1984.

[FS96] P. Ferreira and M. Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proc. 16th ICDCS*, 1996.

[Fuc95] M. Fuchs. Garbage collection on an open network. In H. Baker, editor, *Proc. IWMM*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[GF93] A. Gupta and W. K. Fuchs. Garbage collection in a distributed object-oriented system. *IEEE Transactions on Knowledge and Data Engineering*, 5(2), 1993.

[Ghe95] S. Ghemawat. The modified object buffer: A storage management technique for object-oriented databases. Technical Report MIT/LCS/TR-666, MIT Laboratory for Computer Science, 1995.

[Gra78] J. N. Gray. Notes on database operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.

[HM92] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[Hug85] R. J. M. Hughes. A distributed garbage collection algorithm. In *Proc. 1985 FPCA*, volume 201 of *Lecture Notes in Computer Science*, pages 256–272. Springer-Verlag, 1985.

[JJ92] N.-C. Juul and E. Jul. Comprehensive and robust garbage collection in a distributed system. In *Proc. IWMM*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[JL92] R. E. Jones and R. D. Lins. Cyclic weighted reference counting without delay. Technical Report 28–92, Computing Laboratory, The University of Kent at Canterbury, 1992.

[KW93] E. K. Kolodner and W. E. Weihl. Atomic incremental garbage collection and recovery for large stable heap. In *Proc. 1993 SIGMOD*, pages 177–186, 1993.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. SIGMOD*, pages 318–329. ACM Press, 1996.

[LC97] S. Louboutin and V. Cahill. Comprehensive distributed garbage collection by tracking the causal dependencies of relevant mutator events. In *Proc. ICDCS*. IEEE Press, 1997.

[LeL77] G. LeLann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

[LGG+91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. SOSP*, pages 226–238. ACM Press, 1991.

[LJ93] R. D. Lins and R. E. Jones. Cyclic weighted reference counting. In K. Boyanov, editor, *Proc. Workshop on Parallel and Distributed Processing*. North Holland, 1993.

[LL92] R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *Proc. ICDCS*. IEEE Press, 1992.

[LQP92] B. Lang, C. Queinniec, and J. Piquer. Garbage collecting the world. In *Proc. POPL '92*, pages 39–50. ACM Press, 1992.

[LSW91] B. Liskov, L. Shrira, and J. Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2):125–142, 1991.

[Mah93a] U. Maheshwari. Distributed garbage collection in a client–server persistent object system. In E. Moss, P. R. Wilson, and B. Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, 1993.

[Mah93b] U. Maheshwari. Distributed garbage collection in a client–server, transactional, persistent object system. Technical Report MIT/LCS/TR–574, MIT Press, 1993.

[Mah97] U. Maheshwari. Hula: An efficient protocol for reliable delivery of messages. Technical Report Technical Report MIT/LCS/TR–720, MIT Laboratory for Computer Science, 1997.

[MKI+95] M. Maeda, H. Konaka, Y. Ishikawa, T. T. iyo, A. Hori, and J. Nolte. On-the-fly global garbage collection based on partly mark-sweep. In H. Baker, editor, *Proc. IWMM*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[ML94] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. In *Proc. PDIS*. IEEE Press, 1994.

[ML95] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proc. PODC*, pages 57–63, 1995.

[ML96] U. Maheshwari and B. Liskov. Partitioned garbage collection of a large object store. Technical Report MIT/LCS/TR–699, MIT LCS, 1996.

[ML97a] U. Maheshwari and B. Liskov. Collecting distributed garbage cycles by back tracing. In *Proc. PODC*, 1997.

[ML97b] U. Maheshwari and B. Liskov. Partitioned garbage collection in a large object store. In *Proc. SIGMOD*. ACM Press, 1997.

[MMH96] J. E. B. Moss, D. S. Munro, and R. L. Hudson. Pmos: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proc. 7th Workshop on Persistent Object Systems*, 1996.

[Mos92] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(3), 1992.

[ONG93] J. W. O'Toole, S. M. Nettles, and D. Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proc. 14th SOSP*, pages 161–174, 1993.

[Piq91] J. M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In Aarts et al., editors, *PARLE'91 Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Pos81] J. Postel. DoD standard transmition control protocol. DARPA-Internet RFC-793, 1981.

[RJ96] H. Rodrigues and R. Jones. A cyclic distributed garbage collector for network objects. In *Proc. 10th Workshop on Distributed Algorithms*, 1996.

[Sch89] M. Schelvis. Incremental distribution of timestamp packets — a new approach to distributed garbage collection. *ACM SIGPLAN Notices*, 24(10):37–48, 1989.

[SDP92] M. Shapiro, P. Dickman, and D. Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proc. PODC*, 1992.

[SGP90] M. Shapiro, O. Gruber, and D. Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapports de Recherche 1320, INRIA-Rocquencourt, 1990.

[Sob88] P. Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT, AI Lab, 1988.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2), 1972.

[Ung84] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

[WLM92] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 32–42. ACM Press, 1992.

[YNY94] V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in client–server persistent object stores. In *Proc. Data Engineering*, pages 120–133. IEEE Press, 1994.