Fault-Tolerant Distributed Garbage Collection in a Client-Server Object-Oriented Database

Umesh Maheshwari Barbara H. Liskov

M.I.T. Laboratory for Computer Science
Cambridge, MA 02139

Abstract

We present a scalable garbage collection scheme for systems that store objects at multiple servers while clients run transactions on locally cached copies of objects. It is the first scheme that provides fault tolerance for such a system: Servers recover from failures and retrieve information needed for safe garbage collection; clients do not recover from failures, yet the scheme is able to reclaim objects referenced only from failed clients. The scheme is optimized to reduce overhead on common client operations, and it provides fault tolerance by doing work in the background and during client operations that are infrequent.

1 Introduction

This paper presents a scheme for distributed garbage collection in a client-server object-oriented database system. The algorithm is scalable and fault-tolerant, and minimizes the overhead on common client operations.

The collector works in a system in which persistent objects are stored at geographically distributed servers, while applications run on client machines and interact with the system by invoking object methods. The method calls actually run at the client machines using cached copies of objects, which are fetched from the servers as needed. The method calls are grouped into transactions; when a client commits a transaction, modified copies of objects are sent back to the servers.

Persistence of objects is governed by reachability from a designated root object; the collector must reclaim the storage of objects that cease to be reachable. The scheme collects all unreachable objects except those linked by cycles of references that span multiple servers. It needs to be augmented to collect cyclic distributed garbage; we ignore the issue in this paper.

Authors' email addresses: {umesh,liskov}@lcs.mit.edu.
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.

The algorithm avoids centralized mechanisms so that it can scale to a large number of servers. Like other such schemes, it employs a variant of distributed reference counting, but it differs in two important ways:

1. It is integrated with client caching and distributed transactions. It is optimized to reduce overhead on common client operations, especially object fetches, by avoiding extra messages, processing, and stable storage accesses.

2. It can tolerate both server and client failures. Servers recover from crashes without loss of information; this is achieved without updating stable information when clients fetch objects. Clients do not survive failures, yet the system is able to reclaim persistent objects referenced only from failed clients. Communication failures such as network partitions may cause some servers to view a live client as failed, but the scheme prevents dangling references at such clients from corrupting persistent objects.

The remainder of the paper is organized as follows. Section 2 describes our system and states the requirements for the collector. Section 3 describes our scheme for tracking inter-node references. (We use the generic term *node* for a client or a server.) Then we describe how the scheme works in the absence of failures: Section 4 describes fetching objects into client caches, and Section 5 describes transaction commits. Section 6 describes how server and client failures are handled. Section 7 summarizes the space and time overheads of the scheme. We discuss related work in Section 8 and conclude in Section 9.

2 The environment

Our algorithm is designed for use in the Thor object-oriented database system [LDS92]. Thor provides a universe of persistent objects stored at geographically distributed servers. The server where an object resides is referred to as its *owner*. Objects contain references to other

objects, which may reside at any server. Persistence is determined by reachability from the persistent root, which is implemented as a collection of persistent root objects, one for each server.

The objects at a server are grouped into *segments*, which are the units of transfer between disk and primary memory. Objects are clustered so that objects that refer to one another are likely to be in the same segment. A reference is a triple ⟨owner-id, segment-id, index⟩, which allows objects to be located efficiently [DLMM93]. References are recycled for pragmatic reasons: after an object is reclaimed, a new object may be given the same reference.

User applications run on client machines and interact with Thor by invoking object methods. Method calls are grouped into transactions; the application specifies when to commit or abort the current transaction. Applications interact with Thor through a piece of Thor code that runs on the client machine. This code, which we will refer to as the *client*, executes the methods invoked by the application using a cache of objects fetched from the servers. For example, in Figure 1, the client has fetched objects x and z. Method calls read and modify the cached copies of objects. Note that object references may point from one server to another or from a client to a server, but never from a server to a client or from a client to another client.

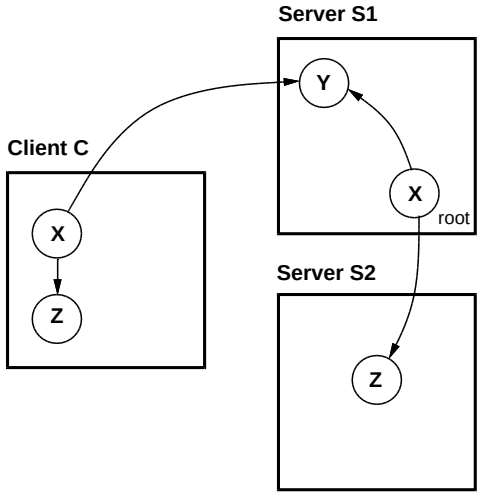

Figure 1: The client-server model.

To *commit* the current transaction, the client sends the request to a server selected from the owners of objects used in the transaction. Copies of objects modified during the transaction are sent along with the request. A 2-phase commit is needed for transactions that use objects at multiple servers [Gra78, Ady94]. We outline the commit protocol since it must be augmented to make garbage collection run properly. The selected server acts as the *coordinator*; it sends *prepare* messages to the owners of other used objects, called the *participants*. Each participant determines whether the transaction can commit, and if so it responds OK after possibly logging a prepare record on stable storage. If all participants agree to commit, the coordinator logs a commit record and notifies the client of a successful commit. If some participants refuse to commit, or don't respond, the coordinator aborts the transaction. The second phase of the commit protocol happens in the background: the coordinator notifies the participants, whereupon they *commit* the modified versions of objects. Future fetches read the new versions from the log until they are *installed*, overwriting the old versions.

When a new object is created, it appears in the client cache as a volatile object. When a transaction commits, any volatile objects that have become reachable from the persistent root are sent to the coordinator; the objects then become persistent at some preferred server.

Transactions are serialized using *optimistic concurrency control* [KR81, Ady94]. Objects are not locked when fetched by a client, so that other clients are free to fetch and modify them. Modifications committed by one client may cause objects cached by another to become invalid. Servers attempt to keep client caches up to date; they track which objects are stored at clients and, if these objects are modified, send *invalidation* messages to the affected clients. Invalidation is an optimization to prevent transaction aborts that would be caused by reading stale data.

Object references never leave the Thor system: When an application calls a method that returns an object, the client code returns a *handle*, which can be used by the application to refer to the object in subsequent calls. (Applications begin their interaction with Thor by obtaining a handle to a persistent root.) A handle is local to a client; for example, an application cannot store it in a file and use it later to interact with another client. This constraint is essential for garbage collection since it guarantees that the only persistent objects that matter are those that are reachable from the persistent roots or from active clients.

Clients and servers have different fault-tolerance characteristics. Servers are persistent and eventually recover from crashes; they are replicated for high availability and reliability [LGGJSW91]. Clients are temporary: they may terminate either normally or due to a crash. Further, servers cannot differentiate between client crashes and long term communication failures like network partitions.

Our garbage collection requirements are as follows:

safety: Objects reachable from the persistent roots or from active clients must not be reclaimed.

liveness: The non-reachable objects should be reclaimed eventually.

It is worth noting that the safety requirement about active clients is actually needed. For example, in Figure 1, client C

has a reference to persistent object y. At the time it obtained this reference, y was reachable from the persistent root, but while the reference resides in C's cache, a transaction committed by some other client may make y unreachable. Nevertheless, we cannot discard y because C may use it later. In fact, C may make it reachable from the persistent root again.

3 The basic scheme: reference listing

Scalable distributed garbage collection schemes use variants of reference counting between separately traced regions as pioneered by [Bis77]. Each node does *local* garbage collections independently of other nodes. To avoid reclaiming objects that are unreachable from the local roots but are reachable from the global set of roots via other nodes, each node keeps some form of *reference information* for local objects that are referenced from other nodes. Local collection uses the reference information as a root in addition to its local roots. (In our system, the local roots are the handles at clients and the persistent roots at the servers.) *Distributed* garbage collection is responsible for updating the reference information as objects are modified: when a node acquires or drops a remote reference, the reference information at that object's owner is updated appropriately.

We use a variant of distributed reference counting that we refer to as *reference listing*, in which each node tracks the identities of the nodes that refer to its objects. A node N_1 keeps, for every other node N_2, a list of objects in N_1 that N_2 may refer to. We call the list the *inlist* for N_2 at N_1, denoted as $\text{IN}(N_2)@N_1$. In our system, clients do not have inlists, while servers keep inlists for clients and other servers. Similar schemes have been used before for non-client-server systems [SDP92, BENOW93]; we have adapted them to handle fetches and commits and to provide the fault tolerance needed in our environment.

The scheme will never reclaim reachable objects provided it satisfies the *safety invariant*:

> If node N_2 refers to an object x at N_1, then x is in $\text{IN}(N_2)@N_1$.

To preserve the invariant, whenever a node acquires a new remote reference, the owner must record it in the appropriate inlist. Consider Figure 2, which shows node N_3 sending N_2 a reference to object z at N_1. In this context, we call N_3 the *sender* and N_2 the *receiver*. N_2 or N_3 must send an *insert* message to N_1 so that N_1 can add z to $\text{IN}(N_2)@N_1$; furthermore, this insertion must be done *before* the reference can be used at N_2.

The liveness requirement is that when a node no longer refers to a remote object, that object must eventually be removed from the inlist for that node at the object's owner. A node will discover that it has no more references to a remote

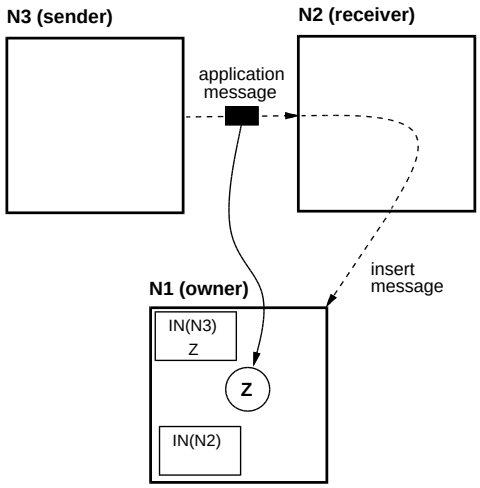

Figure 2: A node acquires a new remote reference.

object as a result of a local collection. Unlike inserting entries into the inlists, deletion can be done lazily. Synchronous insertion and lazy deletion guarantee the safety invariant.

There are two ways of removing entries from inlists. N_2 may send a *delete* message to N_1 for z [BENOW93] or N_2 may periodically send the complete list of references that it holds for objects in N_1 [SDP92]. We call the list the *outlist* for N_1 at N_2, denoted as $\text{OUT}(N_1)@N_2$. Upon receiving the outlist, N_1 uses it to replace $\text{IN}(N_2)@N_1$.

We use outlists because if an outlist message is lost, the next one (sent after the next local garbage collection) will automatically compensate for the loss. Thus, nodes can exchange outlists in the background using unreliable delivery. By contrast, failed delete messages must be remembered and retried. Further, delete messages require that nodes maintain the set of outgoing remote references stably so that the removal of such a reference can be detected; outlists need not be recorded on stable storage.

Insert and outlist messages must still be delivered in order: a late outlist message that is reordered behind a more recent insert message must be rejected. This is achieved by simply numbering the messages.

Reference listing has two important advantages over other variants. First, it tolerates message failures: Insert and delete/outlist messages are idempotent, in contrast to the increment or decrement messages in reference counting, and therefore can be retried on failure. Second, it tolerates client crashes. A server creates an inlist for a client when the client first fetches an object from it. We refer to this as the opening of a *session* between the client and the server. The client can close a session whenever it has an empty outlist for the server or when it terminates. When a client crashes, the server simply discards the inlist for it. With

reference counts, a server cannot figure out which counts to decrement locally; it needs global information to detect garbage objects referenced by failed clients.

Reference listing requires more space than other variants but has an efficient implementation in our system, assuming clustering of objects in segments (see Section 2) corresponds well to application behavior. If a list contains a large number of references from a segment, we replace those references with a bit vector for the entire segment; if the n'th bit in a vector is set, it indicates that the object with index n in the segment is referenced in the list. Because of the clustering assumption, a list will consist primarily of a few bit vectors, one for each segment that is frequently referenced. Further, the bit vectors are small because object indices are small numbers that are reused when objects are deleted.

The interaction between local and distributed collections is shown in Figure 3. Local collection starts tracing from the local root and the inlists and generates new outlists, while the distributed collection uses the outlists to replace the corresponding inlists. In the remainder of this paper, we ignore local collection and focus on distributed collection.

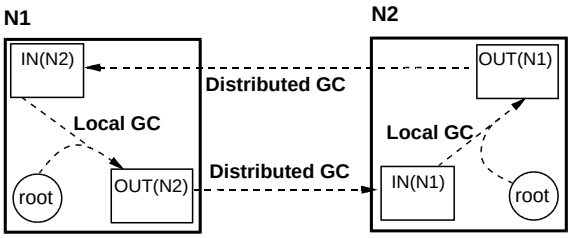

Figure 3: How local and distributed collections interact.

4 The fetch operation

This section describes what happens when clients fetch copies of persistent objects. It ignores failures, which are discussed in Section 6. We begin by describing a simple scheme, and then discuss two important optimizations.

When a client fetches objects from a server, the server records each fetched object in its inlist for the client. It then scans the fetched objects to record all local references contained in them as well. When the client receives the objects, it checks for inter-server references contained in them; if the client did not already have such a reference, it sends an insert message to the owner. The owner records the reference in its inlist for the client and sends an acknowledgment. This is illustrated in Figure 4, where the client fetched an object, x, which contains a local reference, y, and a remote reference, z.

After a client completes local collection, it sends its outlists to the servers, which use them to replace their inlists for the client. This is prone to a race condition: the outlist might not include recently fetched objects that had not yet arrived at the client when it computed the outlists. The problem is solved by requiring the client to delay sending its outlists until all outstanding fetches are complete, and then adding all new references to the outlists before sending them.

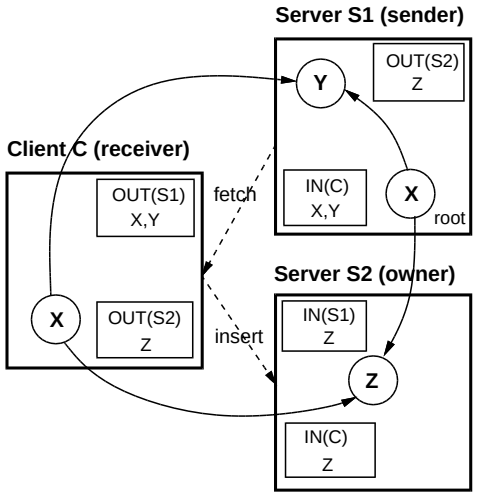

Figure 4: Fetching an object that contains references.

This scheme is correct since it preserves the safety invariant. However, sending insert messages and processing references contained in the fetched objects delay the fetch. The following sections describe two optimizations to avoid this delay.

4.1 Indirect protection

A client can avoid sending an insert message to the owner of an inter-server reference contained in a fetched object, because the reference is already secured by the owner's inlist for the sender server. For example, in the scenario shown in Figure 4, z is already secured by $\text{IN}(S_1)@S_2$, because S_1 has object x that refers to z. But we need to ensure that $\text{OUT}(S_2)@S_1$ contains z at least as long as the client holds a reference to it — even if x is modified in the meantime to drop the reference. We achieve this by recording z in $\text{IN}(C)@S_1$ at the time of the fetch, even though S_1 does not own z. That is, the server records all references contained in fetched objects, local as well as remote. The situation is shown in Figure 5.

In effect, the sender secures the reference to the object at the owner on behalf of the receiver (the client). We refer to this scheme as *indirect protection*. The schemes used in [Piq91, SDP92] are similar, but implemented differently. Indirect protection is safe because it maintains a

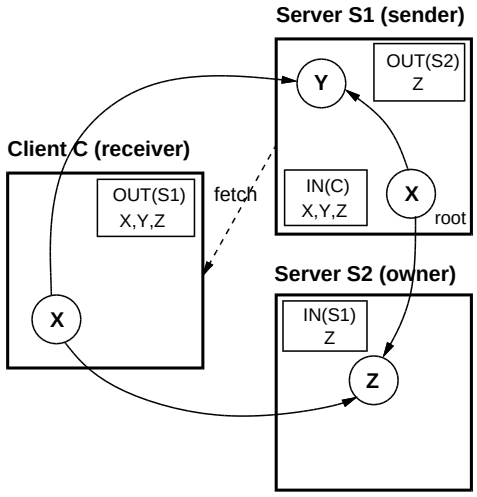

Figure 5: Indirect protection to avoid the insert message.

more general form of the safety invariant:

> If a client C refers to an object z at server S_2, either z is in $\text{IN}(C)@S_2$, or there exists a server S_1 such that z is in both $\text{IN}(C)@S_1$ and $\text{IN}(S_1)@S_2$.

For this scheme to work, the client must remember that it got a reference to z from S_1 and include z in its outlist for S_1, so that S_1 does not remove z from $\text{IN}(C)@S_1$ on receiving the outlist. If the client has received a reference from several servers, it needs to record the reference in only one of the outlists.

4.2 Lazy scan

Recording references contained in the fetched objects delays the fetch. We can avoid recording the contained references because the fetched objects are secured against local collection at the server, and they in turn secure the references contained in them. However, this holds only until a fetched object is modified: when a reference is removed from a fetched object, we must ensure that it is recorded in the inlist.

The *lazy scan* scheme avoids scanning objects at fetch time. When the client does the next garbage collection and the server replaces its inlist with the outlist sent by the client, all contained references actually in use at the client are automatically added to the inlist. However, if a fetched object is modified before this point, the server must explicitly record the references contained in the old version in the inlist. This is accomplished by splitting each inlist into a *scanned* and an *unscanned* part. At fetch time, the server records the fetched objects in the unscanned inlist. When the server receives an outlist from the client, it clears the unscanned inlist and uses the outlist to replace the scanned inlist.

When an object is modified as the result of committing a transaction, its old version secures any contained references until the new version is installed. (The local collector must trace from both the old and the new versions until then.) Before installing the new version of an object, if the object is present in the unscanned inlist for any client, the server records the references contained in the old version in the scanned inlist for that client; the server also moves the entry for the object itself from the unscanned inlist to the scanned inlist.

The probability that a new version will be installed for a recently fetched object (in the unscanned inlist) is likely to be small, assuming that the clients do garbage collection and send outlists fairly frequently. Therefore, lazy scanning reduces the server load in addition to reducing fetch latency. Scanning old versions may delay the installation of the new versions, but installs are infrequent and are done in the background.

To see why lazy scanning is safe, we define the *closure* of a client inlist as the union of the sets of references in the scanned and unscanned inlists and the references contained in the currently installed versions of objects in the unscanned inlist. Lazy scan maintains the following safety invariant:

> If a client C refers to an object z at server S_2, then either z is in the closure of $\text{IN}(C)@S_2$, or there exists a server S_1 such that z is in the closure of $\text{IN}(C)@S_1$ and in $\text{IN}(S_1)@S_2$.

5 The commit operation

The commit operation may create new persistent objects. A server assigns references to its new persistent objects and records those references in its scanned inlist for the client.

The commit operation may also create new inter-server references. Consider Figure 5, where the client fetched x from S_1, thus obtaining a reference to z at S_2. Figure 6 shows a possible later state of the system: the client fetched w from S_3 and copied a reference to z into w. The client then committed the modification, changing w at S_3. In effect, the client has sent a new remote reference to S_3.

The basic reference listing scheme requires that the server receiving a new remote reference, S_3, send an insert message to the owner of the object, S_2. Note that the client already has an entry for z in its outlist for the original sender, S_1, which in turn has an entry in its outlist for the owner. We could use the indirect protection scheme to suppress the insert message: the client and the original sender could secure the reference on behalf of the receiver, S_3. Unfortunately, this does not tolerate the temporary nature

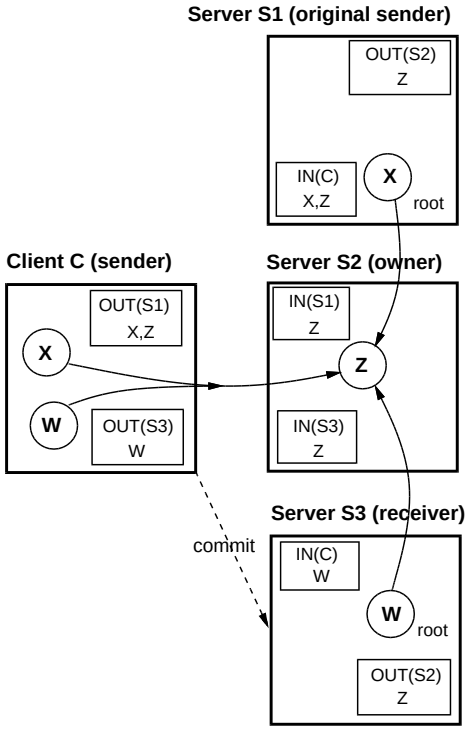

Figure 6: Commit transfers a remote reference.

of clients: When the client terminates, the indirect protection disappears. In fact, the client could crash immediately after sending the commit request.

Thus, an insert message needs to be sent to a server that can secure the reference, such as the original sender, S_1, or the owner, S_3. Our scheme sends an insert message to the owner. The insert message must be sent during phase 1 of the commit protocol, so that the transaction can be aborted if the message fails. An abort annuls the modifications made by the transaction, which avoids the danger of creating a dangling reference in the persistent store.

A straightforward way to incorporate insert messages in phase 1 is described below and illustrated in Figure 7:

1. The coordinator sends prepare messages containing modified copies of objects to the participants.
2. A participant searches for newly acquired remote references in the modified objects and sends insert messages to their owners. (A participant can tell whether a remote reference is new by looking in its outlist for the owner.)
3. An owner records the references in its inlist for the participant and sends an acknowledgment to the coordinator.
4. The participant informs the coordinator (in its acknowledgment message) that the owners are new participants in the transaction.
5. The coordinator waits for acknowledgments from the original and the new participants. (Actually, each acknowledgment carries information about the work completed by the participant, and the coordinator waits for each participant to complete all work expected of it.)

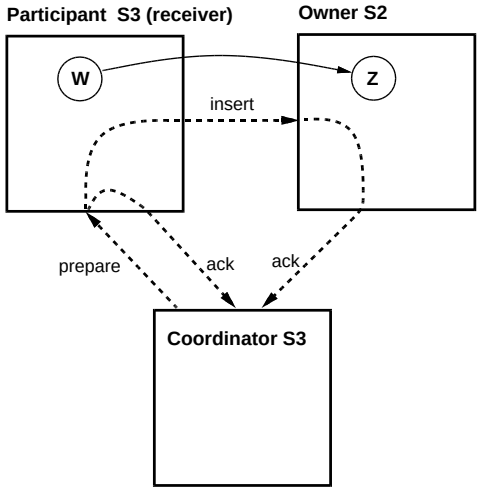

Figure 7: Insert message as part of the commit protocol.

If an insert message to an owner fails, the coordinator will not receive an acknowledgment from it, and the transaction will abort as desired.

Commits that involve insert messages thus have an added latency of one additional message. [Mah93] describes a scheme that hides this latency by having the coordinator send the insert messages on behalf of the participants in parallel with the prepare messages. However, this technique has drawbacks of its own — it complicates the in-order delivery of insert and outlist messages, and may send unnecessary insert messages since the coordinator does not know which remote references already exist at the participants. We expect that commits in which participants acquire new remote references are relatively rare so that adding a one-message delay to such commits does not degrade the performance enough to merit a more complicated scheme.

6 Failures

As mentioned, servers are persistent and recover from crashes, while clients are temporary and may terminate normally or crash. Below we discuss how server and client failures are handled.

6.1 Server crash recovery

When a server recovers from a crash, it must retrieve all its inlists before doing a local collection. The server keeps its inlists for other servers on stable storage; it updates the stable information in response to insert messages from other servers during phase 1 of the commit protocol. As discussed in Section 5, inserting into a server inlist is a relatively rare occurrence; further, the stable update of the inlist happens in in parallel with the logging of the prepare record at other participants.

If the server were to store inlists for clients on stable storage, each fetch request would be delayed by a write to stable storage — a significant price to pay. Therefore, servers retrieve client inlists by communicating with their clients instead. When a server opens a session for a client, it records the client's id and net-address in its *client-list* on stable storage. This makes opening sessions relatively expensive, but such events are expected to be rare.

Thus, when a server recovers from a crash, it knows who its clients are. It sends query messages to them asking them to send it their outlists, and does a local garbage collection only after it has restored the inlists of all its clients. This raises a question: What happens if one of its clients does not respond to query messages? We address this issue in the next section.

To close a session, the client sends a message to the server, which then removes the client from its client-list. The client does not wait to ensure the delivery of a close-session message: The message may fail to arrive either due to network failure or because the server had crashed at the time. The failure of close-session messages is equivalent to the case when the client crashes without notifying the servers, which is discussed below.

6.2 Client failures

Servers need to discard inlists for clients that appear to have failed, since they would otherwise be unable to collect garbage objects that were referenced by such clients. If a client has not communicated for a long time and does not respond to repeated query messages, servers assume it has failed. However, 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. In this case it may hold references to deallocated objects, and these references must be prevented from corrupting persistent objects at other servers.

One solution is for servers to never reuse references assigned to deleted objects, as in [BENOW93]. This would allow the owner of a reference to detect whether the reference is invalid by checking if the referenced object exists. Owners need to perform this check on receiving insert messages to prevent dangling references from entering server objects. We rejected this scheme because of its expense: in addition to the cost of the check, it precludes the use of small and efficient references [DLMM93]. Reuse of references allows us to use small references that contain information to locate objects efficiently, and to avoid maintaining information about deleted objects; note that in a long-lived system like ours, there can be a large number of deleted objects.

Note that the problem can be solved trivially in a single-server system: Since a client must open a session with the server before using it, a server rejects any request from a client for which it has no session information. However, when there are multiple servers, one server might assume a client to have failed while another server does not. When the first server discards its inlist for the client, it might cause the second server to reclaim an object that it had indirectly secured on behalf of the client. (The problem arises because of our use of indirect protection to suppress the insert message.) Now, if the client tries to fetch that object from the second server, the server would not know whether the reference is stale, *i.e.*, refers to a deleted object whose reference may have been reused. Also, the client could commit a transaction that inserts stale references into objects at the second server. Therefore, we employ an *atomic shutdown* protocol, which ensures that all servers get a consistent view of a client's status.

When the client first starts up, it sends a *startup* message to some preferred server, such as the one containing the persistent root used by its application. The server assigns the client a globally unique id, records it in the stable client-list, creates an inlist for the client, and returns the id to the client. This server acts as the reliable *proxy* for the client; it will always know whether the client has been shut down or not. It also tracks the servers that have open sessions with the client: it stores a stable *server-list* for each client for which it acts as a proxy. All requests sent by the client include the identity of the proxy server. When the client terminates normally, it notifies the proxy, which discards the related information.

Before a server (say, S_1) opens a session for a client, it sends a query message to the client's proxy to determine the status of the client, as shown in Figure 8. If the proxy believes the client is live, it adds S_1 to the stable server-list for the client and responds OK. Otherwise, it tells S_1 that the client has been shut down. If the proxy validates the client, S_1 opens a session as described before except that it also records the id of the proxy in the client-list.

If later S_1 is unable to communicate with the client, it asks the client's proxy to initiate a shut down. If the proxy does not have any information about the client, it tells S_1 that the client has terminated (this could happen, for instance, if the close-session message sent from the client to S_1 failed to arrive). Otherwise, the proxy carries

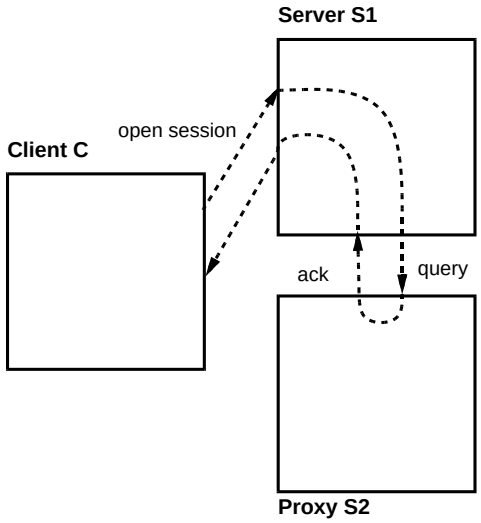

Figure 8: Servers contact proxy before opening sessions.

out a 2-phase shutdown protocol:

1. In phase 1, the proxy notifies all servers in the server-list of the client to shut it down.
2. The servers flag the client as dead (on stable storage) and send an acknowledgment. They do not discard the client's inlist yet, but merely agree not to honor any further requests from the client.
3. When all servers have acknowledged, the proxy sends out phase 2 messages.
4. The servers remove the client from their client-list and discard the client's inlist. The proxy also discards all information about the client.

If any server is temporarily unavailable during phase 1 of the protocol, the proxy simply waits, which is acceptable because servers are expected to recover from failures and survive partitions. If a server doesn't receive the phase 2 message, it can ask the proxy to do the shut down again; this allows the protocol to be robust even if the proxy fails during phase 1.

The 2-phase protocol guarantees the *shutdown invariant*:

> No object referred to by a live client is reclaimed until all servers with sessions for that client know that it has been shut down.

No server will honor requests from the client after it is shut down: servers it had sessions with know about the shut down by then (or will find out when they recover from a failure), and other servers will be unable to start up sessions for the client. Therefore, the stale references held by the client are harmless. In particular, transaction commits will not transfer stale references into persistent objects: Since the coordinator must have a session with the client, it will be notified of the shut down before any object referred to by the client is reclaimed. The coordinator aborts a transaction if it is notified of the shut down at any time before it commits the transaction. Thus, if a transaction commits, it is guaranteed that no object referred to by the client was reclaimed by then.

We now address the question left unanswered at the end of Section 6.1. If a server recovering from a crash is unable to recover the inlist of a client in its client-list, it asks the proxy to initiate a shut down. The recovering server must wait until phase 2 of the shutdown protocol before doing a local collection; this is needed to preserve the shutdown invariant.

7 Performance

Table 1 summarizes the data structures required at the servers. (Note that a portion of the client inlist is already needed for other purposes such as cache invalidation.) Table 2 summarizes the overhead on the latency of client operations in terms of the number of extra stable storage writes and extra messages due to distributed garbage collection. The extra message for commit is needed only in the case where a server acquires a new remote reference. Note that the numbers represent the latency of the operation and therefore discount stable writes and messages that happen in parallel. For example, in the commit operation, the insert messages are sent in parallel and account for the latency of one extra message; the stable storage writes at the owners happens in parallel with the logging of the prepare records at the participants, and the acknowledgments from the owners overlap with those from the participants.

data structure at server	storage
inlist for each client	volatile
inlist for each server	stable
outlist for each server	volatile
client-list	stable
server-list for each "proxied" client	stable

Table 1: Space overhead at server.

client operation	#stable writes	#messages
startup at proxy	1	0
open-session	2	2
fetch	0	0
commit with insert	0	1

Table 2: Overhead on client operations.

8 Related Work

Distributed garbage collection techniques fall into two categories, global marking and distributed reference counting. Global-marking traverses the entire object graph from the roots, sending *marking messages* to span remote references [HK82]. Such schemes do not tolerate node crashes; further a global sweep over large numbers of nodes each with large storage does not allow timely collection of garbage.

Most scalable systems therefore use some variant of distributed reference counting [Bis77]. The variants differ in the information kept for incoming remote references. Some schemes only record a flag for remotely referenced local objects [Ali84, JJ92]. Although this approach minimizes the reference information, it cannot detect locally when an object ceases to be referenced remotely.

Other schemes record a count of how many external nodes have references to an object [Ves87]. These schemes can detect when an object is no longer remotely referenced, since the count is incremented or decremented as nodes acquire or drop the reference. The increment and decrement messages must be sent reliably — without duplication, loss, or reordering. Most of the research in the area has focused on how to avoid the extra messages [Bev87, Piq91]; these schemes do not address node failures. [MS91] uses a combination of a reference count and a bit per client for remotely referenced objects to handle crashes.

[SDP92] uses reference listing and outlist messages. Nodes use a form of indirect protection to suppress insert messages. When a node terminates abnormally, other nodes cannot discard their inlists for it because that might cause indirectly protected objects reachable from live nodes to be reclaimed. The fixes suggested involve either a global mechanism or indefinite retention of garbage. The model does not consider nodes that recover from crashes.

[BENOW93] uses reference listing for a model similar to that of [SDP92], but takes the opposite approach. It sends synchronous insert messages rather than have temporary nodes provide indirect protection. When a node crashes, its inlists at other nodes are discarded. As in our scheme, the inlist for a live but uncommunicative node might be discarded; the use of stale references by such nodes is detected by not reusing object references. The model does not consider persistent nodes.

[LL92] uses a logically centralized service that tracks all inter-node references. Nodes inform the service of their outgoing references and references in transit to other nodes. They also query the service about the reachability of their remotely referenced objects. One drawback of this scheme is that the service may become a bottleneck in a scalable system.

The only other scheme we know of that caters to client-server database systems is [YNY94]. The model involves multiple clients and a single server, and the paper focuses on the various alternatives for local collection at the server.

Variants of distributed reference counting do not collect distributed cyclic garbage. There are two approaches to handling this problem. One is to use complementary global marking [JJ92]. [Hug85] propagates timestamps instead of marks so that multiple rounds of marking and collection proceed simultaneously. [LQP92] uses marking within groups of nodes so that a cycle of garbage can be collected by a group that includes the nodes on which the cycle resides. The second approach is to migrate objects unreachable from local roots to the node from which they are referenced. The scheme meshes well with reference listing because that provides information about which remote nodes reference a local object. This approach was originally proposed by [Bis77] and is used in [SGP90].

9 Conclusion

This paper has described a distributed garbage collection scheme for a client-server object-oriented database. Like other scalable schemes, our scheme is a variant of reference counting, with the difference that it is integrated with client caching and distributed transactions. The scheme is optimized to reduce the overhead on object fetches by clients. As shown in Table 2, it trades off delay in fetches, which happen frequently, for delay in startup and opening sessions, which happen rarely. The work involved in lazy scanning at install time, close-session, and shutdown, happens in the background and does not delay the client.

In addition, the scheme tolerates server and client failures. When a server recovers from a crash, it retrieves its client inlists by contacting the clients rather than by storing the inlists stably, which would delay the fetches. When a client appears to have failed, the servers execute an atomic shutdown protocol that provides a consistent view of the client's status. If the client had not actually failed, but was merely unable to communicate, the protocol guarantees that it will be unable to use its references once it has been shut down, and therefore it will be unable to corrupt persistent objects.

Like all schemes based on distributed reference counting, ours is unable to collect objects on inaccessible, distributed cycles. We are currently investigating extensions to handle this problem.

Acknowledgments

The authors are grateful to S. Ghemawat, R. Gruber, F. Kaashoek, A. Myers, L. Shrira, J. O'Toole, and the referees for their comments.

References

[Ady94] A. Adya. *A Distributed Commit Protocol for Optimistic Concurrency Control.* Master's thesis, Massachusetts Institute of Technology, February 1994.

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

[BENOW93] A. Birrell, D. Evers, G. Nelson, S. Owicki, and E. Wobber. *Distributed Garbage Collection for Network Objects.* Systems Research Center Technical Report 116, Digital, December 1993.

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

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

[DLMM93] M. Day, B. Liskov, U. Maheshwari, and A. Myers. References to Remote Mobile Objects in Thor. *ACM Letters on Programming Languages and Systems*, 1994.

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

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

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

[JJ92] N. C. Juul, E. Jul. Comprehensive and Robust Garbage Collection in a Distributed System. *1992 International Workshop on Memory Management, (Lecture Notes in Computer Science 637)*, Springer-Verlag, 1992.

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

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

[LGGJSW91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 226-238, 1991.

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

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

[Mah93] U. Maheshwari. *Distributed Garbage Collection in a Client-Server, Transactional, Persistent Object System.* Technical Report MIT/LCS/TR-574, Massachusetts Institute of Technology, July 1993.

[MS91] L. Manchini, and S. K. Shrivastava. Fault-Tolerant Reference Counting for Garbage Collection in Distributed Systems. *The Computer Journal*, 34(6), pages 503–513, December 1991.

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

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

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

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

[YNY94] V. Yong, J. F. Naughton, and J. Yu. Storage Reclamation and Reorganization in Client Server Persistent Object Stores. *Data Engineering*, 1994.