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

Umesh Maheshwari*

1 Introduction

We have designed a distributed garbage collector for a new object-oriented database system called *Thor*. The client-server model of Thor makes for a roundabout transfer of references between processes. Clients fetch copies of objects from multiple servers into their caches, modify the cached copies, and send the modifications back at commit points — effectively transferring references from one server to another.

Our design avoids global mechanisms in order to be scalable. It tolerates abnormal termination of client processes, and allows the servers to restart collection after recovery from a crash. It handles the unreliability of a wide-area network, including partitions. While these goals have been met previously in other settings, our design handles (and minimizes the overhead to) the client-induced operations that transfer references: fetching objects from servers, and committing modifications. The basic scheme is a variant of reference counting, and as such, does not collect cyclic garbage spanning multiple servers.

This is a brief report; the details can be found in [Mah93]. Section 2 presents an abstracted and simplified version of the client-server model of Thor. Section 3 lays down the design goals. Section 4 describes the basic scheme that records incoming and outgoing references. Concerns regarding fetching objects are dealt in Section 5, while those regarding committing modifications are in Section 6. Section 7 gives a summary.

2 The Setting

Servers are responsible for persistent storage of objects. Objects contain references to other objects, which may be at the same or another server. Each server may have some objects marked as the top-level *directories*, whose names are publicly known.

A client *fetches* objects from different servers into its local cache. It begins by fetching a directory object and thereby gets hold of references to other objects, which it can then fetch, if required. The client may modify the copies of objects it has cached. In Figure 1, the client has fetched objects X and Y, copied the reference to Z from Y into X, and deleted the original references contained in X and Y.

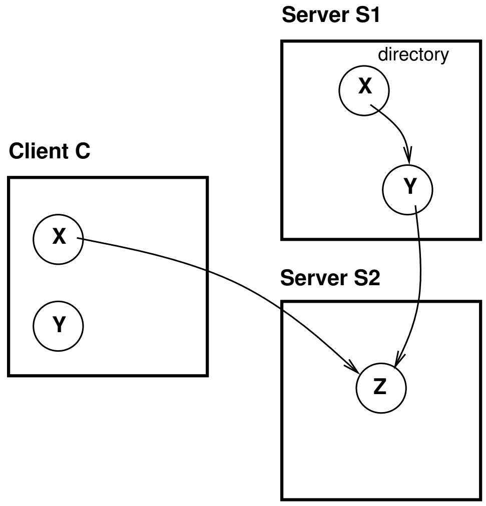

Figure 1: The client-server model

At some point, the client decides to *commit* the last sequence of modifications. A 2-phase commit protocol is used [Gra78]. The client sends the modified copies of objects to a server chosen as the *coordinator*. The coordinator propagates the copies to appropriate servers through *prepare* messages. A server receiving a prepare message, called a *participant*, logs the message and sends back an acknowledgement. When the coordinator receives acknowledgements from all participants, it logs a commit record and informs the client. The second phase of the commit protocol happens in the back-

*The work was guided by the author's advisor, Barbara Liskov. Author's address: umesh@lcs.mit.edu.

ground: the coordinator notifies the participants, upon which they *install* the modified versions of objects. In the absence of acknowledgements from any participant, the coordinator decides to abort the modifications, and notifies the client and the participants.

The clients and servers are fundamentally different processes. A client may terminate any time — normally or otherwise — and never recover. A server is persistent: if it crashes, it eventually recovers. Second, clients have no incoming references from any remote process.

3 The Goals

Since the servers are long-lived, it is necessary to recycle their storage by reclaiming unusable objects. The ideal goal for the collector is to reclaim as many objects (liveness) without affecting any client operation or sacrificing integrity of references (safety).

Safety dictates that the roots for garbage collection include the top-level directory objects, and the objects that clients hold references to (or cached copies of). It might seem redundant to include the latter, since clients always begin by fetching the directory objects. But in fact, a client could cache a reference to an object, and then a commit (by the same or another client) make that object unreachable from the directories. The object must not be garbage collected because the client may make a fetch request for the object or even commit another modification that makes the object reachable from the directories again.

When a client terminates abnormally, the servers must be able to collect objects that were referenced from only that client. Also, a server must be able to safely restart garbage collection when it itself recovers from a crash. The design should scale to a large number of servers and clients. We assume that even as the numbers increase, the number of remote references per process remain about the same.

Fetch and commit transfer references between processes. (In this paper we ignore migration of objects between servers.) The latency of these operations is directly visible to the clients. Therefore, the performance goal is to minimize the overhead added to these operations, in particular the fetch, which it is more frequent.

4 Reference Listing

We use a variant of distributed reference counting that we call *reference listing*. Instead of reference counts, a process P_1 keeps — for every other process P_2 — a list of objects in P_1 that P_2 might be holding references to. We call the list the *inlist* for P_2 at P_1, denoted as $IN(P_2, P_1)$. This scheme has been used before [SDP92]; our contribution is in introducing techniques that handle and optimize fetch and commit. In our client-server model, clients do not have inlists, while servers keep inlists for clients and other servers.

Each process does a *local* garbage collection based on tracing. The roots for this collection include the local directories and the entries in the local inlists. Thus, inlist entries serve to save remotely referenced objects from local collection. The *distributed* garbage collection is responsible for keeping the inlists up-to-date. In this paper, we focus on the distributed part.

Among the advantages of reference listing over counting is that it allows progress in the wake of abnormal terminations of processes. When a client terminates, the server simply reclaims the inlist for that client. In the reference counting scheme, it would not be possible for the server to figure out which counts to decrement.

When a new remote reference appears, it must be added to the suitable inlist at the owner process. Suppose process P_3 sends to P_2 a reference to object Z at P_1. In this context, we call P_3 the *sender*, P_2 the *receiver*, and P_1 the *owner*. P_2 must send an *insert* message to P_1 so that P_1 may add Z to $IN(P_2, P_1)$. Entries are added to the inlist for a client at fetch time, and to the inlist for another server at commit time. We describe in Section 5.2 how insert messages are safely suppressed for fetches, and in Section 6 why the same cannot be done for the commits.

When P_2 discards all references to Z, the liveness requirement is that the entry for Z be eventually (perhaps lazily) removed from $IN(P_2, P_1)$. In fact, there are two ways of effecting the removal. P_2 may send a *delete* message to P_1. Alternatively, P_2 may periodically send the complete list of references that it holds to objects in P_1. We call the list the *outlist* for P_1 at P_2, denoted as $OUT(P_1, P_2)$. It is the responsibility of the local collector to trim the outlists. Upon receiving the outlist, P_1 uses it to replace the inlist for P_2. Timely insertion and lazy deletion in the inlist guarantees that $IN(P_2, P_1)$ is a superset of $OUT(P_1, P_2)$.

The advantage of sending outlists over delete messages is in tolerating message loss. If an outlist message is lost, the next one will cover up. Thus the outlists can be exchanged between the processes in the background using unreliable delivery. (If insert messages are not suppressed, as is true for the commit operation, it must still be ensured that a late outlist message — arriving after an insert message sent later — is rejected.)

Like other variants of distributed reference counting, this scheme does not collect cyclic garbage spanning multiple processes. It is possible to augment the scheme to collect such garbage through a variety of techniques [Mah93], but we have not decided on one.

5 The Fetch Operation

When a client fetches objects from a server, the server must enter in the inlist not only the objects actually fetched, but also all references contained in those objects. To see why, consider Figure 2. If Y is not recorded in the inlist, then if another client commits a modification that removes the reference from X to Y, Y will be exposed to local collection. (In our model, the objects cached by one client are not locked against modifications by other clients.) Hence the server must *scan* the objects fetched for contained references.

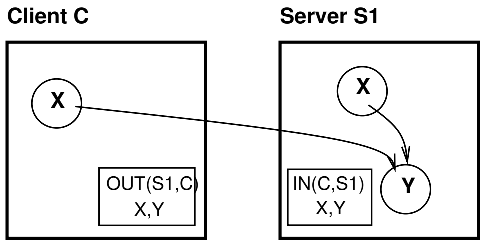

Figure 2: Fetching object containing references

5.1 Avoiding Scanning Objects

Recording references contained in the fetched objects delays the fetch and unduly expands the inlist. This is especially true when, for each object requested by the client, many others get prefetched.

Our *scan-on-need* scheme defers scanning a fetched object until a modification to it is about to be installed. The inlist for each client is split into an object-inlist that records only the objects fetched, and a reference-inlist that records the references contained in the fetched objects. At the time of fetch, entries are made only into the object-inlist — for X in the above example. Later, if a new version of X is about to be installed, the server checks if X exists in any object-inlist. If so, it scans the old version of X and records the contained references in the corresponding reference-inlist. Indeed, given that modifications are rare compared to fetches, most objects fetched will never need to be scanned, saving time and space.

Searching for X in the object-inlists of clients does delay the install; but recall that installs happen in the background, so the delay is not visible to the clients.

Many optimizations can be added to the scan-on-need scheme, but we will not discuss them here. Further, for simplicity, we shall ignore the use of this scheme in presenting the rest of this paper.

5.2 Avoiding Insert Messages

What if the object fetched has a remote reference to another server? Consider Figure 3. When the client fetches Y from $S1$, $S1$ has effectively transferred it a reference to Z at $S2$. By the reference listing protocol, the client must send an insert message to $S2$, which must then create an entry for Z in $IN(C, S2)$. The following illustrates what might

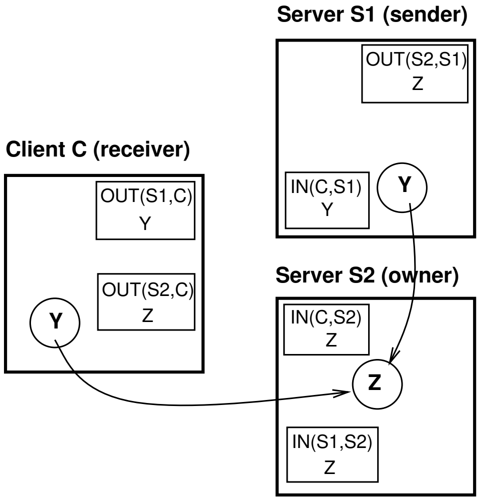

Figure 3: Fetching object with remote references

go wrong otherwise. Another client may commit a modification that deletes Y's reference to Z, which will reflect in the removal of Z from $OUT(S2, S1)$. Then $S1$ will send an empty outlist to $S2$, removing Z from $IN(S1, S2)$. Now, without $IN(C, S2)$, Z would be exposed to the local collection.

We avoid the insert message by having $S1$ (the sender) save the entry for Z in its outlist for $S2$ (the owner) on behalf of the client (the receiver). To this end, $S1$ enters Z in $IN(C, S1)$ although Z does not belong to $S1$ (see Figure 4). This entry saves the entry for Z in $OUT(S2, S1)$ — even if the reference from Y to Z is deleted. On the other side, the client includes Z in $OUT(S1, C)$, which ensures that the entry for Z in $IN(C, S1)$ will not be deleted. Thus, the sender server takes on the responsibility to save the remote object as long as the client holds a reference to it. Similar schemes, called *indirect protection*, have been used in [Piq91, SDP92].

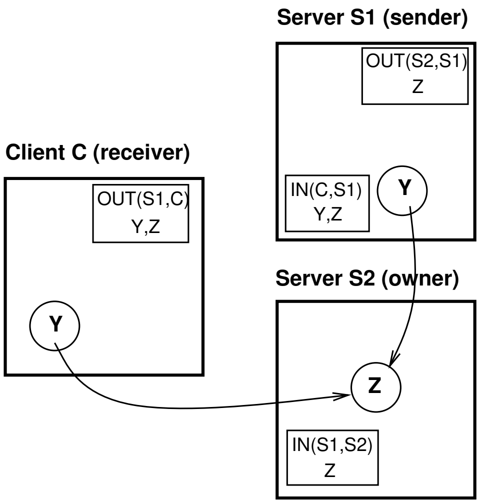

Figure 4: Suppressing the insert message

5.3 Avoiding Disk Writes

Ideally, objects referenced from a client should not be collected despite server crashes. To this end, when the server recovers from a crash, it must retrieve the inlists of its earlier clients that are still alive. The straightforward way is for the server to store the inlists on disk. However, this implies that each fetch request will be delayed by a disk write — an unacceptable cost.

As an alternative, the server could store on disk only the *net-addresses* of its clients (the very first time the clients contact it). When the server recovers from a crash, it contacts the clients, asking them to send their outlists so that it may reconstruct its inlists. The server defers a local collection until, for each earlier client, it has the inlist or the client is known to be dead. The problem is that the server cannot defer a local collection for ever while waiting for a client to respond. Ultimately, the server must time out and collect without the unavailable inlist. Thus this scheme provides safety with a high probability, but does not guarantee it.

Yet another alternative is to have the clients *ping* the servers periodically. The server need not store any client information on the disk. Instead, when the server recovers from a crash, it waits for a period longer than the ping period before doing a local collection. Hopefully, all earlier clients that are still alive will ping the server in this duration, allowing the server to retrieve their inlists. This scheme does not guarantee safety either. Our design involves a mix of the two alternatives which strengthens safety and reduces the delay before a local collection.

We argue that it is indeed *impossible* to guarantee safety — even without server crashes — given the following liveness goal: a server should not retain the inlist of an uncommunicative client for ever. Indefinite retention of an inlist will withhold the collection of unusable objects. If a server does not hear from a client for a long time, it assumes the client is dead and reclaims the inlist. The classic problem here is that it is impossible to distinguish a crash of a remote process from a long-term communication failure between the processes (*i.e.*, a partition). As such, a server might mistake a partition for a client's crash and reclaim the inlist of a live client.

Thus, objects referenced from a client might accidentally be reclaimed at the servers. However, we do guarantee that the integrity of the persistent store at the servers is not compromised. For example, objects at the server will never come to contain dangling references. This requires that the use of stale references by a client at commit time be detected. The task would be easy if references of reclaimed objects were never reused. However, in Thor, references are location-dependent and are reused for efficiency reasons [DLMM93]. Our solution is based on a lease mechanism [GC89] that alerts the client of impending reclamation of an inlist, but we will not discuss it here.

6 The Commit Operation

The commit operation may create a new remote reference at a server. Consider Figure 5 as a stage following Figure 4. The client fetched W from $S3$, and copied the reference to Z from Y into W. Then it committed the modification, changing W at $S3$. In effect, the client has sent a new remote reference to $S3$. Actually, the transfer happens through a coordinator server, which may be different from $S3$, but it merely acts as a go-between. Also, note that the remote reference in question was originally sent by $S1$ during a fetch operation.

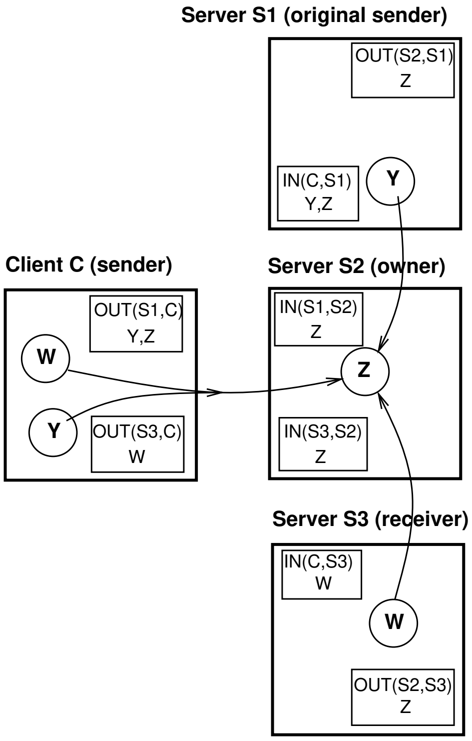

Figure 5: Commit transfers a remote reference

We now argue that, given our safety goals, it is imperative to send an insert message to $S2$ to create an entry for Z in $IN(S3, S2)$ *before* the commit succeeds. The following illustrates what might go wrong otherwise. Another client commits a modification that deletes Y's reference to Z. As arranged in Section 5.2, the entry for Z in $OUT(S2, S1)$ is still protected by the entry in $IN(C, S1)$. But suppose that the client — done with its work — reclaims its copies of Y and W and sends an empty outlist to $S1$, which effectively nullifies $IN(C, S1)$. Now the entry for Z in $OUT(S2, S1)$ is removed, eventually removing Z from $IN(S1, S2)$. In the absence of an entry for Z in $IN(S3, S2)$, Z may be reclaimed by the local collection, leaving W with a dangling reference.

Why not use the indirect protection scheme to suppress the insert message as in the case of the fetch operation? The analogy would be for the client (the sender in this case) to take on the responsibility to keep the entry for Z in $OUT(S1, C)$ on behalf of the $S3$ (the receiver). The reason this does not hold is that the client is temporary, and when it terminates the indirect protection will wear off. In the case of the fetch operation, the sender was a server — a stable entity.

Now consider the following proposition that would at least defer the insert message until after the commit so that the commit is not delayed: The client takes the responsibility of protecting Z in its outlist until $S3$ has sent an insert message to $S2$. The insert message can thus be sent lazily. This proposition does not hold either because the client may indeed crash before the insert message is through, upon which $S1$ will reclaim $IN(C, S1)$, thus opening a window in which Z is vulnerable to collection.

From another perspective, the reference has been transferred from the original sender server, $S1$, and one might consider having it save the reference in its outlist until the receiver server has sent an insert message to the the owner. However, when $S1$ sent the reference to the client, it did not know which servers, if any, will finally receive it. Therefore a message must be sent to the $S1$ at commit time to inform it, which is no better than sending a message to the owner itself.

The insert message must be sent during the prepare phase of the commit protocol. This allows the protocol to abort if the insert message fails (say, because the owner server is unavailable), so that the modifications to objects can be annulled, thus avoiding the danger of creating a dangling reference in the persistent store. It is possible that the owner is not a participant of the commit protocol, in which case extra messages are inevitable. (Even the original sender server might not always be a participant.)

Further, unlike inlists for clients, those for servers are maintained on the disk, since we do not allow breach of safety for the persistent store. Also, there is no liveness requirement for a server to discard inlists for uncommunicative servers because servers are persistent.

6.1 Serial vs. Parallel Insert

The straightforward way to send an insert message to the owner server is as follows. When a participant receives a prepare message, it checks for new remote references in the modified copies. For each such reference, it sends an insert message to the owner. Only after receiving an acknowledgement from the owner, does it itself send one to the coordinator. If the insert message fails, the coordinator will not receive a acknowledgement from the participant, and the protocol aborts.

If an old outlist message from $S1$ to $S2$ gets reordered behind a more recent insert message from $S1$ to $S2$, it must be rejected. This is achieved by simply timestamping the messages.

This scheme adds a message round trip to the latency of the commit protocol (see Figure 6-i). We have designed 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 (Figure 6-ii). Further, if the owner also happens to be a participant of the commit protocol, the insert and the prepare messages can be merged, thus eliminating message overhead. The scheme has its own drawbacks — it complicates orderly delivery of outlist messages — but we will not discuss them here.

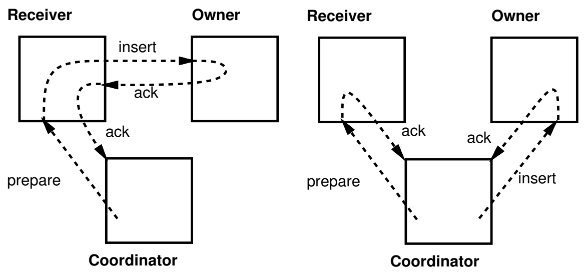

Figure 6: Insert message at commit time

7 Summary

The distributed collector described here handles caching of objects by the clients and commits of modifications back to the servers. The use of reference listing allows servers to collect in the wake of dead or uncommunicative clients. Outlist messages exchanged in the background do not require reliable delivery. If a server crashes, it retrieves the GC information on recovery. Fetch operations are expedited: objects sent to the clients need not be scanned for contained references; no insert message need be sent to other servers; no disk write is needed to store GC information. Commit operations might need extra insert messages, but we propose a scheme that hides the latency by sending them in parallel with prepare messages. We still need to address the collection of cyclic garbage spanning multiple servers.

References

[DLMM93] M. Day, B. Liskov, U. Maheshwari, and A. Myers. Object References in Thor. Submitted to *LOPLAS'93*.

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

[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.

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

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

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