Implementation of the Two-Phase Commit Protocol in Thor

by

Andrew Kirmse

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by .... [illegible]
Barbara Liskov
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by .... [illegible]
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Implementation of the Two-Phase Commit Protocol in Thor

by

Andrew Kirmse

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the detailed design and implementation of the two-phase commit protocol for Thor, a new object-oriented database management system. The protocol relies on an optimistic concurrency control scheme and loosely synchronized clocks for transaction validation. It incorporates a number of performance optimizations, including presumed abort, short-circuited prepare, the coordinator log protocol, and further optimization of read-only transactions. The thesis also discusses some features of the protocol unique to Thor, such as client cache invalidation and the assignment of new object names.

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

Contents

Chapter 1

Introduction

This thesis describes the implementation of the two-phase commit protocol for a distributed, object-oriented database called Thor. The commit protocol is the mechanism that causes modifications made by clients to become permanent at servers. The protocol also ensures that the modifications are applied to the database in a consistent order across all servers.

Distributed databases such as Thor face the challenge of ensuring that the servers present a consistent view of the database, in spite of server failures and conflicting requests arriving in parallel from clients. The solution to both of these problems is to group client operations into *transactions*. The system guarantees the serializability and failure atomicity of transactions, allowing transactions to run in parallel, and guaranteeing that a transaction either runs to completion or not at all, even when a server fails. The database survives failures by writing its progress to stable storage. It runs transactions in parallel by using *concurrency control* to restrict the modifications that clients are allowed to make.

Concurrency control schemes are divided into two categories. *Pessimistic* schemes acquire a read or write lock on a piece of data before attempting to read or modify it. If some other process already holds one of the locks that an operation requires, the operation waits until the lock is released. In an *optimistic* concurrency control scheme, clients can make modifications without acquiring locks, but the modifications become permanent only when the clients *commit* them as part of a transaction. The database runs a *validation* algorithm at commit time to verify that the transaction is consistent with other transactions in the system. If so, the transaction commits and its modifications are made permanent; if not, the transaction aborts.

The two-phase commit is a common algorithm in distributed systems. Thor's commit protocol is somewhat unusual in that it uses optimistic concurrency control. It includes a new validation method that takes advantage of the loosely synchronized clocks that are present in modern networked computers. The protocol is also considerably optimized so that the system can process transactions quickly.

The work described in this thesis extends an existing implementation that required a client to communicate with only one server. Allowing multiple servers complicates matters considerably, both because clients and servers must now communicate with potentially many other servers, and because a two-phase commit protocol is necessary to process transactions that involve more than one server. Extending the existing system involved removing the single server assumption, and implementing the commit protocol and a number of optimizations to it.

The rest of this thesis is arranged as follows. Chapter 2 gives a broad overview of Thor, paying particular attention to the parts that are related to the commit process. Chapter 3 describes the detailed operation of the two-phase commit protocol. We begin with a simple base protocol, add optimizations one at a time, and conclude with a discussion of parts of the protocol that are unique to Thor. In chapter 4, we tell how the commit protocol has actually been implemented, including the control flow and data structures involved with the protocol at both clients and servers. The last chapter summarizes the thesis and identifies areas where more work is required to finish and complement the implementation. In particular, timeouts, recovery, and replication are the essential missing pieces.

Chapter 2

Overview of Thor

Thor is a new client-server, object-oriented database system that provides highly available access to persistent objects. Applications written in different programming languages can operate concurrently and share objects. Object state is encapsulated so that it can be modified only by invoking methods of the object. Clients access objects within transactions, which consist of one or more method invocations. Using transactions guarantees that the entire group of calls executes atomically.

This chapter describes the features of Thor that are relevant to the implementation of the two-phase commit. Further discussion can be found in [1, 10]. The first section gives an overview of Thor's architecture, and the second section goes into a little more detail on concurrency control, which is central to the implementation of the two-phase commit protocol.

2.1 Architecture

Thor's objects are stored at highly available servers. Requests originate from clients, about which no availability assumptions are made. Clients and servers will generally run on different nodes connected via a network. A part of Thor called a *front end* runs on the client node and services application requests. The application views Thor's objects as residing in a single space, although they are actually distributed among the servers.

2.1.1 Servers

Each server is responsible for managing the objects that reside there. A server possesses a disk for persistent storage of objects, and volatile memory for fast access to objects and storage of data structures. The server reads objects in blocks from the disk and caches the blocks in memory to improve performance.

A server contains a special root object that acts as a directory of persistent objects at that server. When an object is reachable from the root of some server, it is persistent; others are garbage and are eventually reclaimed by a distributed garbage collector [12].

The state of an object includes internal variables and references to other objects. Objects that reside at one server may refer to other objects at any server by an *xref*, which is a two-part name for an object. The first part is the server identifier, an integer that uniquely identifies the server at which the object is located. The second part is an *oref*, which is a name for the object unique to the server. A server assigns an xref to an object when the object becomes persistent there, but the xref may change if the object migrates to another server or to another location at the same server [6]. When an object moves, a *surrogate* is left behind in the old location. The surrogate contains the object's new xref.

Servers will be replicated via a primary copy scheme [14] to give high availability. In this scheme, each object resides at both a *primary* server and a *backup* server. During normal operation, clients communicate with the primary; when the primary fails, the backup takes over the primary's role. All servers are equipped with uninterruptable power supplies that allow them to flush their volatile state to disk when the power fails. In chapters 3 and 4, we assume that servers are not replicated, except when stated otherwise. The impact of replication is discussed in the conclusion.

2.1.2 Clients

A client node runs one or more application programs, and usually a front end as well, although the front end might be located elsewhere. Applications contact the front end by messages or shared memory, and the front end handles all communication with servers. The application can be written in any programming language; a language-dependent *veneer* packages calls to the front end [3]. Where no confusion will result, we often refer to the front end as the "client."

The front end caches objects that it has fetched from servers. When an object arrives from a server, it contains references to other objects in the form of xrefs. In order to improve performance, xrefs are *swizzled* into virtual memory pointers that point directly to other objects in the front end's cache. Various swizzling strategies are possible; their implications are discussed in [5]. When the front end attempts to follow a reference to an object that is not present in the cache, it sends a *fetch* request to the server where the object resides. The server responds by sending the object, and perhaps some other objects that it expects the client may need soon. This technique is known as *prefetching*.

When the front end starts up, it makes a connection to a server and fetches its root object; the client calls methods of this object to get started. The front end fetches objects from servers as needed to carry out these calls.

All interaction between the application and the front end occurs within a transaction. The front end supports one client transaction at a time; as soon as the client ends a transaction by requesting a commit, a new transaction starts for it. The application makes calls to the front end to invoke methods on objects in the cache; some of these calls may create new, non-persistent objects. Client method calls have no effect on the copies of objects at servers until the client commits the current transaction. At this point, the front end contacts the servers. If the commit succeeds, the modified copies of objects are installed at servers, and new objects that are reachable from the modified objects become persistent.

2.2 Concurrency control

An important measure of a database's performance is the number of transactions it can process per unit time. For good performance, the database should use a concurrency control scheme that takes advantage of its architecture and expected usage patterns. Below we discuss the choice of concurrency control for Thor.

When contention is low—that is, when each object is likely to be accessed by at most one client—optimistic schemes perform better than pessimistic ones, because pessimistic schemes must communicate with servers to obtain locks, even when the objects to be locked already reside in the client's cache. Locking can also cause deadlocks, and so pessimistic schemes must expend some effort in preventing or detecting deadlocks. However, when contention is high, optimistic schemes abort often, causing their performance to degrade relative to pessimistic schemes.

Optimistic concurrency control fits Thor's architecture well. Because of prefetching, client caches are expected to hold most of the objects that the application requires. With an optimistic scheme, the application can operate on these objects without communicating with servers, making operations very fast. The penalty is that the operations may abort in the future; this is more likely when contention is high. Thor currently always uses optimistic concurrency control. Research is in progress on a hybrid scheme that adapts its concurrency control to the level of contention in the database, so that pessimism will be used for high contention objects [8].

Servers need to reach a consensus on whether a transaction should be allowed to commit. The two-phase commit protocol is a standard algorithm for solving this problem. The first phase of the protocol distributes the transaction to the servers and validates it. The second phase informs servers about the outcome, and makes the transaction's modifications permanent if it committed. The protocol admits to several extensions and optimizations in the context of Thor; these are discussed in the next chapter.

Chapter 3

Two-phase commit

This chapter first gives an overview of a very simple two-phase commit algorithm, ignoring the possibility of failures. We then describe how the protocol records its progress in a log and tolerates failures. Later sections tell how the protocol can be optimized. The chapter concludes with a description of the details of the two-phase commit protocol specific to Thor.

3.1 Basic algorithm

Committing a transaction involves communication between a set of servers (called *participants*) where objects read or written by the transaction reside. One of these participants is selected as the *coordinator* for the transaction. In the basic algorithm, the coordinator is the only participant that communicates with the client.

The basic two-phase commit protocol is divided into two parts, the *validation phase* (phase 1) and the *update phase* (phase 2). At the beginning of the validation phase, the client sends a *prepare* message to the coordinator. This message lists objects that have been read and written by the transaction, and includes copies of objects that the transaction has modified. The coordinator then sends prepare messages to the other participants. Each participant *validates* the transaction's operations, that is, it determines whether the transaction can commit without violating a particular set of consistency requirements (Thor's consistency requirements are described in section 3.5 below). The participant sends the result of validation (called its *vote*) to the coordinator. The vote can have one of two values, either COMMIT, meaning that validation was successful, or ABORT, meaning that validation failed.

If the coordinator receives an ABORT vote, it immediately tells the client that the transaction has aborted. Otherwise, the coordinator waits until COMMIT votes have arrived from all the participants, and then it informs the client that the transaction has committed. This marks the end of phase 1.

Phase 2 occurs in the background. The coordinator informs participants of the result of the transaction. If the transaction committed, participants *install* the new values of objects modified by the transaction, making the changes visible to subsequent operations on the objects. Participants then send an *acknowledgment* to the coordinator. The coordinator collects the acknowledgments; when all the acknowledgments have arrived, phase 2 is complete.

3.2 The log

In order to survive server crashes, participants in the two-phase commit protocol must record enough of their progress on stable storage (usually a disk) to ensure that they can continue operation after they recover from a crash. This stable state is usually organized into a *log* which is made up of *records* of various types. New records are appended onto the end of the log in volatile memory and are periodically written out to stable storage. Since records that have not yet reached stable storage are lost when the server crashes, the server must *force* the log to be written to the disk immediately whenever the server must ensure that a log record is stable. Further, the transaction that initiated the force may need to wait for the force to complete before continuing.

Sometimes a server must force a log record to disk before sending any information in the record to another server. If the first server crashes after the force but before sending the information to another server, it will find the record upon recovery and resend the message. Servers must be able to handle these duplicate messages that arise from crashes at other sites.

The following is a summary of the log records in the standard two-phase commit:

- A *prepare* record is logged by a participant after it validates a transaction. This record must be forced to stable storage before the participant sends its vote to the coordinator.
- When the coordinator knows the outcome of a transaction, it logs either a *commit* or an *abort* record. This record is forced before the coordinator informs the client and participants of the transaction's outcome. Each participant also logs and forces a commit or abort record when it learns the outcome of the transaction from the coordinator[1].
- After the coordinator receives an acknowledgment from every participant for a transaction, it logs a *done* record. This record need not be forced.

Participants send acknowledgment messages to the coordinator in phase 2 so that the coordinator can free storage used by the transaction. Once acknowledgments have arrived from all participants, the coordinator is sure that participants will never inquire about the transaction again, since participants have a stable record of the transaction's outcome. Thus the coordinator can delete its volatile state associated with the transaction. By the same reasoning, the done record logged at the end of phase 2 allows the coordinator to omit a transaction from its volatile state upon recovery, since all acknowledgments have already arrived.

The log must occasionally be garbage collected to keep it from growing too large. A record can be removed when the effects of the transaction that logged the record have been written on stable storage. A done record in the log signals that a transaction is complete, and that its records may be discarded. Management of the log is a current topic of investigation [7].

[1]Practical protocols do not force abort records because of the *presumed abort* optimization, which is discussed below.

3.3 Failures

One of the strengths of a distributed database is that it can continue operation even when part of the database fails. Thus, the commit protocol should be able to handle failures such as loss of messages due to network partitions, or server crashes because of power failures or software errors. Implicit in the two-phase commit protocol is a recovery process that rebuilds a server's volatile state after a crash, and a set of timeouts that are meant to be invoked when another site crashes or the network fails.

3.3.1 Timeouts

There are two alternatives to dealing with lost messages. The sender of the message can keep resending until it receives an acknowledgment, or the receiver can keep querying the sender until the message arrives. In either case, a node must time out and try again when it doesn't receive a message it was expecting within a certain time interval. The size of this interval depends on the properties of the network and the importance of delivering the message quickly.

If a prepare message or a participant vote is lost during phase 1, the client waits while servers try to resolve the problem. Thus, it is important for the coordinator to either resend the lost message quickly, or to give up and abort the transaction. If the coordinator finds that some participant's vote is late, it can simply abort the transaction. This case handles the loss of both participant prepare messages and votes. Alternatively, the coordinator could ask participants to resend their votes. If the prepare message to the participant was lost, the participant could respond to the coordinator's query either by doing nothing, which will cause the transaction to abort, or by asking the coordinator to resend the prepare message. Similarly, if the coordinator notices that it has not received a participant's vote, it can abort the transaction right away, or retransmit the prepare message to the participant. All of these retries should be sent only once or a few times and should timeout quickly, so that the transaction can be resolved with little delay to the client.

If a participant finds that it hasn't received the outcome of a transaction some time after sending its vote, it periodically queries the coordinator for the outcome of the transaction. Although this takes place in the background, it is important that participants learn the outcome of a transaction fairly quickly, so that they can install that transaction's modifications if it committed. Doing installation quickly can avoid aborts of other transactions that need to see the first transaction's updates.

Loss of an acknowledgment message in phase 2 is not so serious. The coordinator collects acknowledgments only so that it can free resources associated with a transaction. This is not a time-critical operation; we only need to make sure that it is done eventually. To minimize the amount of work spent during phase 2, participants can send their acknowledgments at the same time as other messages, called *piggybacking*[2]. The coordinator periodically resends commit or abort messages to participants for which it is missing acknowledgments. This resend can have a long timeout and use backoff to minimize the number of retransmissions.

3.3.2 Recovery

When a server crashes, it recovers its state from the log on stable storage. However, all of the information that the server held in volatile storage before the crash is lost. When the coordinator crashes, it may lose the participant votes for a transaction in phase 1. When a participant crashes, it may forget the outcome of a transaction.

If a server finds that it has a prepare record for a transaction in its log without a matching commit or abort record, and if it is a participant for the transaction, it asks the coordinator to send the transaction's outcome. The participant treats the coordinator's reply just like a commit or abort message in the commit protocol. If a server finds that it is the coordinator for a transaction in the log, and there is no matching commit or abort record, it can simply abort the transaction.

Other situations are handled by the timeout mechanism. For example, if a participant crashes just before sending its acknowledgment to the coordinator, the coordinator will keep querying until the participant recovers. If the participant crashes

[2]In Thor, servers periodically exchange *liveness* messages to detect crashes; these messages are good candidates for piggybacking.

while validating a transaction, the coordinator will eventually abort the transaction.

3.4 Optimizations

Note that after phase 1, the client knows the result of the transaction and is free to begin another transaction. The primary goal of optimization is to reduce the delay visible to the client; this means reducing the foreground delay of phase 1.

The latency of a commit protocol is measured in terms of foreground communication delays and writes to stable storage. The basic scheme described above has a delay of four messages in phase 1 and requires two synchronous disk writes (for the prepare record at each participant and the commit or abort record at the coordinator). The schemes described below either provide a shortcut that eliminates some of the delay, or they speed up common cases at the expense of some extra complexity in the protocol.

3.4.1 Presumed abort

Suppose the coordinator did not force its commit or abort record at the end of phase 1. Then if the coordinator crashed, it would not know whether to tell participants that a transaction committed or aborted. However, if the coordinator forced **only** commit records or **only** abort records, then the lack of a log record would indicate the outcome of a transaction just as surely as the presence of a log record. This observation is the basis of the *presumed abort* and *presumed commit* optimizations, which can save a disk write in phase 1.

In the presumed abort strategy, the coordinator does not force abort records. If the coordinator ever receives an inquiry about a transaction for which it has no information in its log, it replies that the transaction has aborted. This is always safe because information is kept about committed transactions until no participant will ever inquire about the outcome. For transactions that abort, participants do not need to send acknowledgments to the coordinator, and the coordinator does not need to log a done record. Presumed commit is the symmetrical case, which doesn't force commit records. However, it requires an extra log force at the coordinator during phase 1 to record the fact that an active transaction has not yet committed.

Presumed commit sounds attractive because most transactions are expected to commit, and so it optimizes the common case. The extra log force required for presumed commit is a heavy penalty, but a more complicated variant has been devised to eliminate it in most cases [9]. Thor uses presumed abort in the interest of simplicity.

3.4.2 Read-only-at-site transactions

If a transaction doesn't modify or create any objects at a given participant, then the participant is said to be a *read-only site*. Read-only sites do not need to be informed of the outcome of the transaction; such sites don't care if a transaction commits or aborts, because there are no modifications to install. There is also no need for the site to log anything about the transaction. However, read-only sites must still perform validation in Thor's concurrency control scheme. These sites can send a special READ vote to the coordinator; the coordinator then ignores the site in phase 2.

3.4.3 Read-only everywhere transactions

When a transaction makes no modifications and creates no new objects at any participant, then it is *read-only everywhere*. Phase 2 is not necessary at all for these transactions, and no participant writes any log records. In fact, the only purpose of the coordinator is to collect votes and return the result to the client. Thus, the client itself can act as the coordinator for read-only everywhere transactions.

If the client finds that the transaction it is about to prepare is read-only everywhere, it notifies the participants of this fact in the prepare messages. The participants then perform validation and send their votes directly to the client. The client commits the transaction if it receives no ABORT votes.

3.4.4 Short-circuited prepare

Instead of sending a commit request to the coordinator, the client can send prepare messages to all participants directly. This *short-circuited prepare* saves one message delay in the basic commit protocol, but has some other consequences as well:

- In the basic scheme, the coordinator sends prepare messages to other participants. At that time it must decide where newly persistent objects should be placed. With a short-circuited prepare, these tasks are handled by the client, taking some of the load off the servers.

- In Thor, each transaction is given a timestamp which is used in validation (see section 3.5.1). Since the client now assigns the timestamp, we must require the client's clock to be loosely synchronized with clocks at the servers.

- It is possible that the coordinator will begin receiving votes for a transaction before it has received the prepare message from the client. This situation is unlikely, because it requires two messages and validation at a participant to occur before a single message is sent from the client to the coordinator. When the coordinator receives one of these votes, it can either abort the transaction immediately, wait to see if a prepare message arrives, or ask the client to resend the prepare message.

3.4.5 Coordinator log protocol

An optimization due to Stamos [15] can often save a force of the log to stable storage. In this *coordinator log protocol*, the coordinator logs prepare records on behalf of participants. Participants send their prepare records to the coordinator along with their COMMIT votes, and the coordinator forces these records to disk along with its commit record. Participants also add their prepare records to their own logs, so that they can install transactions later without retrieving the records from the coordinator. However, participants do not force these records to disk.

When a participant crashes and recovers, it must retrieve its prepare records from other servers. Thus, the participant must keep a stable set C of servers that have been coordinators for transactions involving the participant in the recent past. When the participant validates a transaction, it checks if the coordinator is in C. If so, it sends its prepare record along with its vote. If not, it adds the coordinator to C and forces the prepare record to stable storage. This force causes the new value of C to be added to the stable server state.

3.4.6 Summary

The two-phase commit protocol for Thor includes all of the above optimizations. With all of the optimizations in place, there is a delay of three messages and one log force (at the coordinator) before the client learns the result of a committing read-write transaction. If a read-write transaction aborts, this delay is reduced to just the three messages (and no log forces). The latency of a read-only everywhere transaction is two messages and no log forces.

3.5 Two-phase commit in Thor

Thor's commit algorithm only allows a transaction to commit if it preserves two consistency requirements. First, it provides serializability: there must exist a total ordering of committed transactions such that the effects of applying the transactions serially in this order are the same as the effects actually observed on the database. Second, if transaction T_1 commits before transaction T_2 begins, then T_1 must appear before T_2 in the total order. This second condition is known as *external consistency.*

Below we give Thor's validation algorithm, which ensures that the consistency requirements are preserved.

3.5.1 Validation

Thor uses a new validation scheme based on the assumption that clients and servers have clocks synchronized to within a maximum skew, via a mechanism such as the

Network Time Protocol [13]. The validation algorithm is summarized below; it is described in more detail in [2].

When a client wants to make its current transaction permanent, it sends prepare messages to all servers that store objects involved in the transaction. The prepare message to server S contains the following information:

- A globally unique transaction identifier, which includes the client's name and a timestamp taken from the client's clock.
- Read object set (ROS): The xrefs of objects read by the transaction at S.
- Modified object set (MOS): Copies of existent objects at S that are modified by the transaction.
- New object set (NOS): Copies of new objects that should be made persistent at S if the transaction commits.

To simplify the validation algorithm, the client ensures that MOS $\subseteq$ ROS; in other words, a transaction that modifies an object is assumed to read it also. This mild restriction is unlikely to cause many transactions to fail validation, because it is rare in practice to change an object without first reading it.

Thor orders transactions by their timestamps, so validation amounts to placing the incoming transaction in the order given by its timestamp and checking the consistency requirements. Two transactions are said to *conflict* if one modifies an object that the other reads or modifies. The validation algorithm ensures that conflicting transactions are serializable in order of their timestamps; this is enough to guarantee that a total serialization order exists.

A participant checks incoming transactions against all previously prepared transactions, a scheme known as *backward validation.* The ROS and MOS of previously prepared transactions are stored in a *validation queue* (VQ). Each participant checks incoming transactions against all other transactions in the validation queue. If the transaction passes validation, the participant adds the transaction's ROS and MOS information to the VQ. Otherwise, the participant throws out the transaction's information.

Validation checks

An incoming transaction T is allowed to validate successfully if it passes the following tests against transactions in the VQ:

1. For every uncommitted transaction S with a timestamp earlier than T, S's MOS and T's ROS are disjoint. If S commits, T can commit only if it has read S's modifications, since S is serialized before T. But T has not read these modifications, because they have not yet been installed. Therefore, T must abort.

2. For every transaction S with a timestamp later than T:

   - T hasn't modified any object that S read. Since S has already validated, T must be aborted if it affects S's serializability.

   - T hasn't read any object that S modified. This condition is not needed for serializability, but it preserves external consistency.

3. T has read the most recent version of every object in its ROS.

The last check ensures that T reflects the results of all previously committed transactions. In Thor, this check is done by maintaining an *invalid set* for each connected client. This set holds xrefs of objects that are cached at a client, but have been modified since the object was sent to the client. A transaction from a client C fails validation if it reads an object in C's invalid set.

Servers send *invalidation* messages to clients in phase 2 to inform them that objects have been added to their invalid sets. Clients drop these objects from their caches and send *acknowledgments* to the server. (Clients also abort their current transactions if they have used any of the invalid objects.) When it receives an acknowledgment, the server removes the objects from the client's invalid set.

Truncation of the validation queue

If the VQ were allowed to grow without bound, validation would become increasingly slow, and the server would eventually run out of memory. The idea behind truncating the VQ is to reduce its memory requirements and keep validation fast, while trying to minimize the number of aborts caused by throwing out some of the validation history.

We keep a timestamp *threshold* which is later than any transaction that has been removed from the VQ, and we abort any incoming transaction whose timestamp falls below the threshold. The threshold marks the point after which we retain all read object information for validated transactions. Modification information for committed transactions is kept in the invalid sets. Thus, it is safe to discard from the VQ committed and read-only transactions whose timestamps lie below the threshold. We never discard VQ entries for uncommitted read-write transactions.

The threshold should not be kept so high that many incoming transactions abort because their timestamps are below the threshold. But the threshold should be set as high as possible so that the VQ remains short. A good choice for the threshold is

$$\textit{local time} - \textit{maximum message delay} - \textit{maximum clock skew}$$

since then incoming transactions are very unlikely to miss the threshold, because of the assumption that clocks are loosely synchronized.

If a server crashes and then recovers, it must set the threshold to a value that will ensure that only serializable transactions can commit. We want to avoid logging the value of the threshold every time a transaction validates, since that would add a disk write to the latency of read-only transactions. Instead, we keep a *stable threshold* that the server uses to set the threshold on recovery. The stable threshold is always later than the timestamps of all validated transactions; whenever a transaction arrives with a timestamp later than the stable threshold, we must log a new stable threshold value. We can minimize the amount of logging required by increasing the stable threshold in big steps. Then read-only transactions will only occasionally need to perform a log force. However, the stable threshold must be kept within reason so that new transactions will stand a chance of committing after the server recovers.

3.5.2 Garbage collection

Thor's distributed garbage collector works in parallel with other activities in the system, including the commit protocol. The garbage collector reclaims an object when it is no longer persistent, that is, when it can never become reachable from some server's root. Servers must update some tables during the two-phase commit to ensure correctness of the garbage collection algorithm.

A client's cache may hold the only reference to an object at a server, and the object may become persistent later as a result of committing a transaction. Thus the garbage collection algorithm must use client caches as roots when locating reachable objects. The algorithm keeps *client tables* at servers to determine what objects are cached at clients. It is sufficient to keep a superset of the objects in the client's cache in a client table, since this guarantees that no object that is not garbage will be collected. (This is in fact how client tables are implemented; cached objects are remembered at a very coarse grain to save space.)

Client tables at a server S store the names of objects that reside at S and are cached by clients; each server keeps a table for every client connected to it. Initially, each table is empty. Whenever a client receives an object as a result of a fetch or prefetch, the object's xref is added to the client's table. Objects that become persistent as a result of a transaction commit are also added to the table for the client that requested the commit.

The garbage collector also needs to know about all inter-server references. References from other servers to objects at server S are stored in S's *inlists*; references from objects at S to objects at other servers are stored in an *outlist* at S. These structures must always contain a conservative estimate of the actual set of inter-server references, so that reachable objects are never collected. In particular, transactions that create new inter-server references must update the inlists and outlists at all affected servers before their changes become visible.

In order to keep the inlists and outlists updated during the commit, participants must ensure that all newly created inter-server references are stored at the servers at both ends of the reference before the commit is finalized. However, these references may point to objects stored at servers that are not otherwise involved in the transaction, and these servers must update their inlists. This means that there may be additional participants besides those that receive prepare messages. As described in [12], participants send *insert* messages to servers that are destinations of newly created inter-server references. These servers then update their inlists and send an acknowledgment directly to the coordinator. The coordinator must wait for an acknowledgment from these servers before proceeding with the commit.

3.5.3 Crash recovery

Client tables are kept only in volatile memory, and so they are lost when a server crashes. When the server recovers, it must restore its client tables before allowing any other normal operations, since garbage collection must not proceed while roots are missing. To rebuild its client tables, the server can ask each client that was connected before the crash for the names of all objects in the client's cache. Thus the server must store the identities of connected clients on stable storage; this can be done by forcing a log record when a client connects or disconnects.

Invalid sets are also lost during a crash. They can be approximately recovered by intersecting the set of objects in the client tables with the set of modified objects stored in the log. The rebuilt invalid sets may be too large, since the clients may have acknowledged some invalidation messages. To solve this problem, the server can maintain an *invalidation number* for each client that increases monotonically with successive invalidation messages. The invalidation number's current value is included in the invalidation message and is also stored in the commit record. When a client sends its set of cached objects to the server at recovery time, it also sends the number of the latest invalidation message it has acknowledged (this assumes that messages are delivered in order). The server then skips transactions with lower or equal invalidation numbers when rebuilding the invalid set from commit records in the log.

Inlists and outlists are kept on stable storage and thus can be reconstructed upon recovery without any communication.

Chapter 4

Implementation

Below we describe the way the two-phase commit protocol has been implemented in Thor. First we give the details of communication between clients and servers. Then we discuss the most important data structures involved in the commit protocol at servers. The last two sections describe the control flow of the commit protocol in the basic scheme before and after optimization.

Throughout this chapter, we discuss the protocol with all of the optimizations from chapter 3 applied. The system does not deal with lost messages or server crashes, because timeouts and recovery are not implemented yet. A more complete implementation would timeout and retry as described in section 3.3, where the current implementation continues to wait for a message that may never arrive.

4.1 Communication

Thor uses TCP/IP to provide reliable message delivery. Where possible, messages are piggybacked to amortize the overhead of sending a message. For example, invalidation messages can be piggybacked on other messages to clients (such as fetch replies), since delivery of invalidation messages is not necessary to guarantee correctness.

Servers await connections on well-known port numbers. To allow more than one server to run on a single machine, each server on the same machine listens on a different port. A server is uniquely identified by its machine name and port number.

Initiating client-server communication requires a method of mapping server names to network addresses and port numbers. A *locator* object at each client and server provides this mapping. When a client or server starts up, it instructs its locator to find the addresses of all known servers. In the current implementation, server locations are simply stored in a file; eventually they will reside in a highly available location such as the Internet Domain Name Service, and they will be retrieved only as needed.

4.2 Server threads

The server is a multithreaded program, capable of handling many connections and transactions concurrently. There are three types of threads that handle the two-phase commit in the server: the listener thread, client threads, and server threads.

The listener thread waits on a well-known port number for incoming client connections, and on another port for incoming server connections. When a client or server makes a new connection, it first sends its name across the connection to identify itself. Client names are a combination of several items, such as the client's IP address, process number, and process start time, which are meant to distinguish even between clients running on the same machine. The name of a server is the server identifier that appears in xrefs of objects that reside there. After it receives this identifier, the listener thread starts a server or client thread to handle messages over the new connection.

A client thread handles all communication with a given client. Clients always initiate the first communication between a client and a server, either to fetch the server's root object or to send a prepare message to a participant. Thus, it is never necessary for a server to start a client thread to establish a new connection to a client[1].

[1]Servers that receive insert messages (see section 3.5.2) send acknowledgments directly to the coordinator of a transaction, and for read-only transactions, the client acts as the coordinator. It might seem that a server that receives an insert message may have to open a new connection to a client acting as a coordinator. However, no new references are created by a read-only transaction, so no insert messages are sent in this case.

Client threads also perform phase 1 of the commit process at a coordinator. Since the client blocks waiting for the result of a transaction, the client thread cannot receive any messages in the meantime. The client thread at the coordinator performs validation and then waits on a condition variable until phase 1 is complete. Then the thread wakes up, informs the client of the result of the transaction, and continues to wait for more messages from the client. While waiting for incoming messages, the thread also periodically checks the invalid set for the client. If any invalidations are pending, the thread sends them to the client.

Each server thread handles communication to one other server. The threads are meant to be reclaimed if their connection is idle for a long period of time, so that connection resources can be reused. To send a message to a server B that has not been contacted for a long time, server A calls the locator to find the network address of B, opens a connection, and starts a new server thread to send the message and listen for replies.

4.3 Server data structures

4.3.1 Transaction manager

The central structure in the two-phase commit is the *transaction manager* (TM), which handles validation, logging, and maintenance of tables and the threshold. The TM is created when the server starts up; it makes an empty validation queue and sets the threshold to the current time. Then it sets the stable threshold to the current time plus some constant and forces this value to disk. After this initialization, the server is ready to process transactions.

Methods of the TM provide several of the basic functions of the commit process. The public methods process prepare messages from the client, as well as commit and abort messages from the coordinator. Private messages perform validation, create and install log records, and update client tables and invalid sets. The use of these methods is described below in section 4.4.2.

The TM contains a validation mutex to prevent different transactions from trying to read and modify the VQ and threshold simultaneously. Holding this mutex at the appropriate times is essential for the correctness of the commit protocol.

4.3.2 The log

The log is implemented as a queue of records. Each record contains a *log sequence number* that uniquely identifies the record and serves as an index into the log. Log records can be in one of several states: *valid*, meaning that the record has been added to the in-memory log, *stable* when the record has been flushed to the disk, and *installed* when the record's contents have been applied to the database. Log records have install methods that apply their modifications, if any.

The implementation adds records to the log and flushes it at appropriate points during the protocol. Flushing the log currently does not actually write records to disk, but only marks the records as stable. Because recovery has not yet been implemented, it does not matter whether the log makes it to disk, since the log is never read in again.

In addition to the standard log records described in section 3.2, one other kind of records is logged during Thor's two-phase commit. A *committing* record is very similar to a prepare record, except that it indicates that the server is the coordinator for the transaction, while a prepare record indicates that the server is a participant. The server needs to know if it is the coordinator for a transaction at recovery time. Prepare and committing records hold values of objects modified by transactions. When a transaction commits, the install method of its prepare or committing record is called, causing the object modifications to appear in the database.

If the coordinator crashes in phase 2, it will have to retrieve acknowledgments from the participants in the transaction, so the participants must be logged somewhere. We log the names of read-write participants in commit records at the coordinator, skipping read-only participants because we do not need acknowledgments from them. However, we also log the names of all participants in committing records. While not necessary for correctness, this may save a transaction from aborting in a replicated system when the coordinator primary crashes in phase 1, but its backup is able to detect the crash quickly and continue the transaction.

4.3.3 Transaction status table

In order to keep track of transactions currently in progress, each server keeps a *transaction status table.* When a transaction arrives at a server, the server creates an entry in the table; the entry is maintained until the server has completed processing the transaction (at which time the entry is removed). An entry for the transaction T contains the transaction's identifier, and a state flag. This flag can have one of three possible values:

Coordinator: This server is the coordinator for T, which is currently in phase 1. In this case, T's entry in the table contains a list of participants that have voted for T to commit, and a list of participants that have yet to send their votes. In addition, the entry contains the name of the client that prepared T and the log sequence number of T's committing record.

Phase 2: This server is the coordinator for T, which is currently in phase 2. Now the entry contains the names of participants that have yet to acknowledge the server's COMMIT message.

Participant: This server is a participant for T, which is in phase 1. The entry contains the log sequence number of T's prepare record. Participants do not keep entries for transactions in phase 2.

The entries also contain some other information, which is described below.

Information in the table entries is updated when messages arrive from the client or from other servers. The status table's methods are designed to make updating the entries easy. For example, when a participant sends a COMMIT vote, the coordinator calls a method of the table to tabulate the vote. This method adds the vote to the transaction's entry, creating a new entry if necessary (this can occur if the coordinator has not yet validated the transaction). It then checks to see if all participants have voted for the transaction to commit, and indicates this fact in its return value. By moving this kind of low-level detail into methods of the status table, the TM's logic can remain uncluttered.

Several different threads need access to the coordinator's status table, namely the client thread responsible for the commit, and the server threads for the transaction's participants. A minor complication this introduces is concurrent access; these threads must hold a mutex associated with the table before reading or modifying it. A more serious problem is that the transaction may be processed at various rates at different servers or by different threads. For example, votes from participants may begin arriving before or after the coordinator has finished validation. In fact, the coordinator may successfully validate a transaction, only to find that it has already aborted at some other participant. To handle these situations requires some extra flags in the table's entries.

Entries in the **coordinator** state contain an *aborted* flag and a *prepared* flag, both of which are initially false. The aborted flag is set to true when an ABORT vote arrives from a participant. When the coordinator successfully validates the transaction, it calls a method of the table that checks the flag and exits with a special value if the aborted flag is true. The prepared flag is set to true in this method, indicating that the coordinator is done validating the transaction. The method that handles COMMIT votes verifies that the prepared flag is true before saying that all participants have agreed to commit the transaction.

Entries in the **phase 2** state can be removed after phase 2 of the commit is complete. The commit is complete only after all acknowledgments have arrived and the transaction's operations have been installed. These entries contain an *installed* flag that is set by the update thread after it has finished installing the transaction. An entry is only removed when this flag is true and the done record has been written to disk.

Participant entries are removed after the participant learns the outcome of the transaction, and has forced a commit record if the transaction committed. The log sequence number of the prepare record, which is stored in the table entry, is used during transaction installation (see section 4.4.2).

Maintenance

It is possible for a participant vote to arrive for a transaction whose entry has already been removed from the table, causing another entry to be created for the transaction. For example, the coordinator may abort a transaction and remove the transaction's entry from the table (following the presumed abort strategy). Participants that have not yet voted might later send votes after they finish validation. Because of this and other similar scenarios, the server must periodically clean the table of very old entries; this can be done lazily in the background. The server can use a transaction's timestamp to determine the age of an entry.

The status table is kept in volatile memory, so when the server crashes, the table is lost. One of the primary tasks of recovery is rebuilding the table from the stable log on disk.

4.3.4 Per-client information

For each connected client, a server keeps a structure protected by a single read-write lock. The structure contains all of the per-client information stored at the server, including the client's name, its invalid set, its client table, and the handle of its thread.

A client's invalid set contains a numbered sequence of object collections, where each collection holds the xrefs of objects invalidated at the client by a single transaction. Associated with a collection is a sequence number that increases monotonically with subsequent transactions (this is the invalidation number mentioned in section 3.5.3). Within the per-client structure, the server records the highest collection number that has been used, and the highest collection number that has been sent to the client. When these two are unequal, the server sends the client an invalidation message containing all the collections that it has not already seen, along with their collection numbers. In its acknowledgment, the client need only mention the highest collection number that appeared in the invalidation message (assuming that messages are delivered in order). The server then removes all collections with equal or lower number from the client's invalid set.

4.4 Control flow in the basic scheme

4.4.1 Client behavior

Whenever the client application reads, modifies, or creates an object, it communicates this information to the front end. The front end saves up the identities of read objects, and makes a backup copy of modified non-persistent objects (in case the transaction aborts). When the application requests a commit, the front end becomes unavailable to the application until the front end learns the transaction's outcome from the coordinator.

The front end's first step when submitting a transaction is building up the transaction's read, modified, and new object sets for inclusion in prepare messages. Each participant receives only those objects involved in the transaction that reside there. Thus, the front end must separate the transaction's persistent objects into disjoint sets based on the server identifiers in the objects' xrefs.

At servers, objects contain xrefs that refer to other objects. In client caches, where traversing references must be done quickly, objects hold virtual memory pointers that point directly to other objects in the cache. In order to send new and modified objects to a server during a transaction commit, these pointers must be converted back into xrefs, a process called *unswizzling*. As each modified object is unswizzled, it is added to the MOS for the appropriate server. New objects don't yet have xrefs; they must be handled specially (see section 4.4.4).

Next, the front end selects the coordinator for the transaction. The coordinator could be selected from among the participants of a transaction according to a number of criteria, such as server load or expected network delay. We choose the server that has the largest combined ROS and MOS, with the assumption that the transaction is most likely to abort there. If the transaction aborts because validation fails at the coordinator, the client will get a response after only two message delays, versus three for an abort at a participant.

The front end sends prepare messages to each participant in turn. In addition to object information, a prepare message contains the identity of the coordinator, and a flag that indicates whether the transaction is read-only everywhere. This tells participants where to send their votes after validation. The coordinator's prepare message also contains the identities of all the participants, since it needs to know which votes to wait for. After sending its prepare messages, the front end waits for a response from the coordinator, or, in the case of a read-only everywhere transaction, for votes from participants.

If the transaction aborts, the front end restores non-persistent objects in its cache to their previous values stored in the backup copies, and evicts from the cache any persistent objects that have been invalidated. If the transaction commits, the front end installs the new values of objects in its cache and deletes the backup copies, and newly persistent objects are assigned their xrefs given in the commit message. In either case, the front end clears its transaction information and waits for further requests from the application.

4.4.2 Participant behavior

When a client thread receives a prepare message, it checks to see if the message is meant for the coordinator or for a participant. If it is for a participant, the server calls the participant prepare method of the TM. This method grabs the validation mutex and attempts to validate the transaction. In addition to the validation checks mentioned in section 3.5.1, the TM makes *reservations* for the copies of new and modified objects created by the transaction. These reservations set aside space at the server, making sure that there is enough room to accommodate the transaction. If not, then validation fails and the transaction aborts.

If validation succeeds, the participant logs a prepare record. Because of the coordinator log protocol optimization discussed in section 3.4.5, this record is either forced to disk at the participant or sent to the coordinator and forced to disk there (see section 4.5.3 below). This prepare record contains the identity of the coordinator and the reservation information for the transaction. The record also includes the stable threshold; it is convenient to log the stable threshold here since we are writing to disk anyway. Since the record is not forced, we cannot update the copy of the stable threshold in memory. Next, the TM adds the transaction to the status table, including the log sequence number of the prepare record. The TM returns the result of validation, and the client thread sends the participant's vote to the coordinator.

Later a server thread at the participant receives the result of the transaction from the coordinator. If the result is abort, the server thread calls a method of the TM to abort the transaction. This method logs but does not force an abort record, and then retrieves the log sequence number of the transaction's prepare record from the status table. The TM uses this number to locate the log record and cancel its reservations. The transaction is then removed from the status table.

If the result of the transaction is commit, the participant first forces a commit record to disk and sends its acknowledgment to the coordinator. Then it starts an *update thread* to install the results of the transaction in the background, and removes the transaction from the status table.

The update thread calls the install method of the transaction's prepare record to make the transaction's modifications permanent. Afterward, the thread adds the transaction's modified objects to clients' invalid sets. The order of these two operations is essential for correctness; to see why, consider an object A that is added to some client's invalid set before a transaction installs its modified value for A. Suppose that the client fetches A before the installation is complete. A might be removed from the client's invalid set as a result, so the server may believe that the value of A cached by the client is the most recent value, when in fact A has been modified by a committed transaction. The client could then use the old value of A in subsequent transactions, which violates serializability.

Invalidation

When a transaction modifies some objects at a server, the server checks its client tables to see if the modifications affect objects in client caches. If so, the server adds these objects to each affected client's invalid set. (The objects are not added to the invalid set of the client that initiated the transaction, since its cache already reflects the results of the transaction.) Client threads then send invalidation messages to affected clients, ensuring that these clients will not try to commit transactions that read old values for the modified objects. The client responds to the invalidation message by evicting the listed objects from its cache, aborting its current transaction if it has read or modified any of the objects, and sending an acknowledgment to the server.

Invalidation messages are sent in the background during or after phase 2, so they do not contribute to the delay observed by clients in committing transactions. These messages are optimizations: they save work by allowing clients to abort immediately transactions that would abort later anyway, but even without invalidation messages, the commit protocol would still operate correctly.

4.4.3 Coordinator behavior

The coordinator performs the same basic steps as participants. The main differences are that the coordinator collects participant votes during phase 1, sends the result of the transaction to the client and participants, and then collects acknowledgments. When a transaction commits, the coordinator moves the entry for the transaction in the status table from the **coordinator** state to the **phase 2** state. It informs participants about the commit afterward, ensuring that no acknowledgments can arrive before the entry is in the **phase 2** state.

4.4.4 New objects

When the front end finds a newly created object during unswizzling, it must assign a temporary name to the object. This temporary name is used as a reference to the object until after the commit is complete, when the temporary name is replaced by a newly assigned xref.

A temporary name is made up of two parts: the name of the server at which the object will reside, and an index into that server's NOS. For example, the first new object that will reside at server A is given the temporary name $(A, 1)$.

Determining at which server a new object should be placed is an important decision with wide performance implications. On one hand, related objects should be clustered at a single server so that prefetching is most useful, and on the other, the load should be balanced over all servers. In the current implementation, the client places a new object at the same server as the first object encountered during unswizzling that contains a reference to it. This is an easy scheme to implement; as references are traversed during unswizzling, we simply use the server name from the xref of the referencing object as the server for the referenced object. However, a more complicated scheme might provide better performance.

When a participant makes reservations for an incoming transaction, it also reserves xrefs for the transaction's new objects. The participant sends these xrefs to the coordinator along with its COMMIT vote. It is important that these xrefs be arranged in the same order as they appear in the participant's NOS, since the temporary names are used to index the set of new xrefs.

The coordinator stores these new xrefs in the status table for the transaction. When all the votes have arrived, the coordinator sends all the new xrefs to every participant along with its commit message. In this message, the coordinator makes clear which xrefs have come from which participants. As part of the installation of the transaction's prepare record, each participant replaces temporary names with the appropriate new xrefs.

The client also receives the new xrefs from the coordinator. After the commit message arrives from the coordinator, the client uses each new object's temporary name to extract the matching xref from the coordinator's message. The client stores this new xref in the object.

Participants' update threads add newly created objects to the client table of the client C that committed the transaction, since C's cache contains these objects. The objects must be added to C's client table before the transaction's prepare record is installed, because once the record is installed, other clients can fetch the objects and use them in transactions. If another client commits a transaction that modifies one of these new objects, the object must already be present in C's client table so that the participant can add the object to C's invalid set.

4.5 Optimizations

4.5.1 Single server transactions

When the coordinator finds that a transaction's participant set includes only itself, it can perform the entire commit protocol alone. The coordinator immediately performs validation; if this succeeds, it logs a committing record to record the transaction's new and modified objects, and a commit record to record the fact that the transaction committed. The coordinator flushes these records to disk and installs the committing record. Finally, it tells the client the result of the transaction.

4.5.2 Read-only transactions

A transaction is read-only at a participant if no modified objects are listed in its prepare message. In this case, the participant performs validation as usual. If the threshold rises above the stable threshold value as a result of validation, then the participant must force a log record containing a new value for the stable threshold. Otherwise, the participant logs nothing, and doesn't add the transaction to its status table.

The participant sends a READ vote to the coordinator, whose TM treats the vote like a COMMIT vote. The status table, however, removes the participant from the set of missing voters without adding it to the list of received votes. This ensures that the coordinator will not inform read-only participants of the outcome of the transaction.

If the prepare message indicates that the transaction is read-only everywhere, then the participant calls the same TM method used by single server transactions, and simply sends its vote to the client. Forcing a log record is necessary only if the threshold climbs above the stable threshold.

4.5.3 Coordinator log protocol

Each server records in memory a set C of servers that have been coordinators for it. Each entry in this set contains the name of a server, and the timestamp of the most recent transaction for which the server acted as the coordinator. When a server successfully validates a transaction and finds that the coordinator is not in C, the server logs a record containing the identity of the coordinator, and forces it to disk along with its prepare record. Otherwise, the server logs but does not force a prepare record, and sends this record to the coordinator with its COMMIT vote. The coordinator adds this prepare record to its log. When the coordinator forces its commit record, these participant prepare records are also written to disk.

The set C may eventually grow so large that the server would have to contact many other servers on recovery, making recovery potentially slow. Anytime the server forces its log to disk, it knows that all of its prepare records are stable, and thus there is no need to contact any other server on recovery. The server could just discard C, but that would mean that new transactions would require forcing a prepare record. Instead, after forcing the log, the server occasionally truncates C by removing entries with very old timestamps. To remove a coordinator from C, the participant logs a special record containing the coordinator's identity. This record need not be forced, since it is always safe to have extra servers in C.

Chapter 5

Conclusion

This thesis has described the implementation of the two-phase commit protocol in Thor, a distributed, object-oriented database system. The protocol is optimized to reduce the foreground delay of a transaction commit to three messages, one coordinator log force and no participant log forces in most cases, versus four messages, one log force at the coordinator, and one at participants, in the most basic algorithm. Transactions that are read-only everywhere require a single message round trip and usually no writes to stable storage.

5.1 Summary and current status

Thor's two-phase commit protocol guarantees failure atomicity of all transactions, so that the database can survive server crashes. Its optimistic concurrency control provides serializability and external consistency for all committed transactions, and takes advantage of the presence of client caches for good performance. The validation algorithm relies on transaction timestamps derived from loosely synchronized clocks. Data structures used by the commit protocol are kept small by periodically removing old entries.

The commit protocol incorporates a number of optimizations to reduce the foreground delay visible to clients. Some of these optimizations put additional constraints on other parts of Thor. The coordinator log protocol sacrifices autonomous recovery, since a server may have to contact other servers to find its prepare records during recovery. The basic validation algorithm requires the presence of loosely synchronized clocks at servers only. Because of the short-circuited prepare optimization, clients must also have these clocks. In addition, the implementation currently assumes that messages are delivered reliably and in order, and that servers never crash.

The basic protocol and all of the optimizations discussed in chapter 3 are currently operating in Thor. We have successfully run distributed transactions using the OO7 benchmark [4] and some hand-made examples. More complete testing should be possible once a substantial application has been written on top of Thor.

5.2 Future work

The major parts of the system related to the commit protocol that are missing are timeouts, replication, and crash recovery. Server garbage collection structures (inlists and outlists) are also still missing, and the protocol currently ignores them.

When timeouts are implemented, the protocol will be able to continue even when some messages are lost. Timeouts will also allow us to dispose of old entries in the transaction status table. The two other missing parts are discussed below.

5.2.1 Replication

In Thor's replication scheme, a primary server normally handles client requests, and the backup takes over the primary's duties when the primary fails. A primary uses its backup's memory as stable storage instead of its disk. In the two-phase commit protocol, where a server now flushes a log record to disk, it can instead send the record to its backup. The primary must wait for the backup to acknowledge receipt of the log record, to make sure that it is indeed present at the backup. So forcing the log involves a delay of a message round trip; assuming that the primary and backup are located near one another, this is faster than an average disk write. The two-phase commit protocol requires no foreground disk writes when servers are replicated in this way. Object modifications will still be written to disk in the background (see [7]).

When a failure occurs, servers execute a *view change* algorithm to select a new primary and backup if necessary. This algorithm is based on the one used in the Harp file system [11]. After a view change, the new primary uses the information in its log to recover from the failure, as discussed below.

5.2.2 Recovery

The purpose of keeping a stable log in the commit protocol is to allow a server to recover from crashes. Implementing recovery will help to verify that the commit protocol places sufficient information about transactions in the log. It will also test the timeout mechanisms, since a long recovery time may cause delayed transactions to abort and messages to be resent to the server that crashed.

Part of recovery in Thor is similar to recovery in nonreplicated systems. In a nonreplicated system, when a server recovers from a crash, it rebuilds its volatile state from the log on disk, which contains information put there by the commit protocol. In a replicated system, the backup performs a *failover* that causes it to become the new primary, and then rebuilds its volatile state from the log. A nonreplicated system is unavailable while the server is crashed; in the presence of replication, a new server quickly takes over the crashed server's role, and the system continues uninterrupted.

Another aspect of recovery is unique to a replicated system like Thor. While a server is down, the new primary and backup continue operation and accumulate new log records. If a crash is lengthy, a server's log may be very out of date when it recovers. It can contact the current primary or backup to retrieve the new log records. A view change is also required to include the recovering server in normal database operation. However, these are details of the failover algorithm, and require no changes to the commit protocol.

References

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

[2] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrecy control using loosely synchronized clocks. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[3] P. Bogle. Reducing cross-domain call overhead using batched futures. Master's thesis, Massachusetts Institute of Technology, 1994.

[4] M. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*, pages 12–21, 1993.

[5] M. Day. *Client Cache Management in a Distributed Object Database.* PhD thesis, Massachusetts Institute of Technology, April 1995.

[6] M. Day, B. Liskov, U. Maheshwari, and A. Myers. References to remote mobile objects in Thor. Programming Methodology Group Memo 79, MIT Laboratory for Computer Science, Cambridge, MA 02139, 1993.

[7] S. Ghemawat. *Disk Management for Object-Oriented Databases.* PhD thesis, Massachusetts Institute of Technology, Forthcoming.

[8] R. Gruber. *Temperature-Based Concurrency Control.* PhD thesis, Massachusetts Institute of Technology, Forthcoming.

[9] B. Lampson and D. Lomet. A new presumed commit operation for two phase commit. Technical report 93/1, DEC Cambridge Research Laboratory, February 1993.

[10] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In M. Tamer Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management.* Morgan Kaufmann, 1993.

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

[12] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. *Proceedings of the Third Conference on Parallel and Distributed Information Systems*, September 1994.

[13] D. L. Mills. Network time protocol: Specification and implementation. DARPA-Internet Report RFC 1059, DARPA, July 1988.

[14] B. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, August 1988.

[15] J. W. Stamos and F. Cristian. A low-cost atomic commit protocol. Research report RJ7185, IBM Almaden, CA, December 1989.