THE DESIGN & IMPLEMENTATION OF NESTED TRANSACTIONS IN LOCUS

Erik Mueller
Johanna Moore
Gerald Popek

1984
Report No. CSD-840214

The Design and Implementation of Nested Transactions in Locus

by

Erik T. Mueller

Johanna D. Moore

Gerald J. Popek

Computer Science Department

School of Engineering and Applied Science

University of California

Los Angeles, California 90024

February 1984

This research was sponsored by the Advanced Research Projects Agency, Department of Defense, under Contract No. DSS-MDA-903-82-C-0189, Secure Reliable Processing Systems.

Table of Contents

page

List of Figures

page

ACKNOWLEDGEMENTS

The implementation of this work could never have existed were it not for the Locus operating system under development at UCLA. Critical comments and suggestions were offered by many members of the Locus research group. We also owe many thanks to Richard Guy, Bruce Walker, and Greg Thiel who worked hard to develop a user-friendly testing environment and to Richard Guy and Evelyn Walton who kept the facility up and running, even when we did our best to break it!

This research has been supported by the U.S. Department of Defense, ARPA, under contract DSS-MDA-903-82-C-0189.

PREFACE

This report documents the design and implementation of a mechanism for full nested transactions within the Locus distributed operating system. The research reported here was carried out by the first two authors for their Master's theses. Chapters 4, 5, 6, and Appendix A are largely the work of Erik Mueller, and Chapters 7 and 8 are largely the work of Johanna Moore.

ABSTRACT

Atomic transactions are useful in distributed systems as a means of providing reliable operation in the face of hardware failures. Nested transactions are a generalization of traditional transactions in which transactions may be composed of other transactions. The programmer may initiate several transactions from within a transaction, and serializability of the transactions is guaranteed even if they are executed concurrently. In addition, transactions invoked from within a given transaction fail independently of their invoking transaction and of one another, allowing use of alternate transactions to accomplish the desired task in the event that the original should fail. Thus nested transactions are the basis for a general-purpose reliable programming environment in which transactions are modules which may be composed freely.

A working implementation of nested transactions has been produced for Locus, an integrated distributed operating system which provides a high degree of network transparency. Several aspects of our mechanism are novel. First, the mechanism allows a transaction to access objects directly without regard to the location of the object. Second, processes running on behalf of a single transaction may be located at many sites. Thus there is no need to invoke a new transaction to perform processing or access objects at a remote site. Third, unlike other environments, Locus allows replication of data objects at more than one site in the network, and this capability is incorporated into the transaction mechanism. If the copy of an object that is currently being accessed becomes unavailable, it is possible to continue work by using another one of the replicated copies. Finally, an efficient orphan removal algorithm is presented, and the problem of providing continued operation during network partitions is addressed in detail.

CHAPTER 1

INTRODUCTION

In the last few years, computers have become widely available. Microprocessors are in our cars, household appliances, and video games. Small business computers have become less expensive and personal computers are now commonplace. The second generation of microprocessors will soon give us inexpensive personal workstations with a capability equivalent to that of current-generation minicomputers.

With this proliferation of computers, people have begun to connect their processors using a communications network. Such networks range from the 300 baud telephone lines used to connect personal computers, and long-haul networks such as the 50 Kbps Arpanet, to high-speed low-delay local area networks such as the 10 Mbps Ethernet. It is easy to visualize the day when every home will connect into a world-wide integrated digital network which will carry audio and visual information, along with other data such as text and programs.

It is now feasible and natural to construct a computer system from a collection of processors connected by a communications network. Such *distributed systems* have several potential advantages over single-machine systems including decreased cost, incremental growth, increased parallelism, and enhanced reliability and availability.

Despite the increasing popularity of distributed systems, the task of building software in a distributed environment may be more difficult than on single-machine

systems for several reasons. First, in many distributed environments the method of accessing a remote resource such as a data object or program is different from, and more complex than, local resource access. A solution to this problem is the concept of *network transparency*. The network should not be of concern to most users and programs. An operation should have the same syntax and semantics, regardless of whether it is performed locally or remotely. For example, the same interface should be used to access a file whether it is local or remote. Programs should be able to execute at any site in the network with the same results. Of course, one still needs a way to control resource location for optimization purposes, but that control should be independent of the syntax and semantics of the use of the resource.

Second, writing distributed applications is made difficult by the richer set of failure modes that arise when both the network and processors are unreliable. For example, suppose one is running a program which updates several data objects located at different sites in the network. If the site on which the program is executing crashes, or if some of the data objects stored on other sites become inaccessible while the program is running, the system may be left in an inconsistent state unless corrective action is taken. One approach to coping with the failures which may occur in a distributed environment is the transaction concept.

1.1 Transactions

There are two problems which arise in the context of failures and concurrent computations and thus arise in distributed systems as well. Consider the following example of a banking system. Suppose there are two bank account data objects, each containing the balance of the account and other information. Further suppose it is desired to execute a *transfer* program, which withdraws some amount of money from one account, and deposits the amount into another account. Now suppose the

transfer program first subtracts the amount from one account, but, before it is able to add the amount into the other account, the processor on which the program is executing fails. In this case, the money withdrawn from the one account has been lost.

The second problem occurs when another program examines the amount of money in the two bank accounts after the transfer program has removed money from the one account, but before it has added the amount to the other account. In this case, the program views a state of the bank accounts which does not obey the constraint that the total amount of money in the system be constant.

The solution to these problems is for the transfer program to be an *atomic transaction,* i.e., the operations performed by the program take place indivisibly with respect to both failures and concurrent computations. That is, either all of the operations will take place or none of them will take place, and other programs executing concurrently cannot modify or observe intermediate states in which some of the operations have been performed but others have not.

Another important property of transactions is consistency. A program which performs manipulations on shared data objects assumes that the data satisfies certain *consistency constraints.* For example, in a banking system, it is assumed that the sum of the balances of all the accounts in the system totals to some particular constant. While such constraints are not usually explicitly stated by the programmer, programs nonetheless depend on the system data objects satisfying such assertions. The set of all data objects in the system, or system state, is consistent if the contents of the data objects satisfy the consistency constraints.

We assume that given a consistent system state, a transaction transforms the system state into another consistent state. That is, we assume that the programmer

of a transaction has taken care to ensure that the transaction preserves consistency. For example, in a banking system, a transaction which transfers money from one account to another will preserve the constraint that the sum of the balances in the system total to a particular constant. In practice, the task of assuring that a program preserves a set of constraints is quite difficult.

A *serial execution* of a set of transactions $\{T_1, T_2, ..., T_n\}$ is an execution in which the transactions are executed one at a time, i.e., each transaction runs to completion before the next one is started. Serial executions of transactions preserve consistency, since each transaction starts with a consistent system state and transforms the consistent state into another consistent state which is passed to the next transaction.

Although serial executions preserve consistency, they may lead to poor performance because they do not allow one to take advantage of possible concurrency. However, arbitrary concurrency between transactions can lead to inconsistency. For example, consider the following situation in which two transactions T_1 and T_2, which are running concurrently, wish to withdraw money from the same account. The withdrawal would be accomplished by first reading the amount of money in the account, subtracting the appropriate amount, and then writing the new amount back into the account. Suppose that T_1 reads the amount, calculates the new amount, but, before it writes the new amount, T_2 reads the amount and calculates the new amount. Next, T_1 writes back the new amount and then T_2 writes back the amount. Here the effect is as if only one of the withdrawals were performed. Thus concurrency can easily cause consistency to be violated.

In order to allow concurrency while preserving consistency, the notion of *serializability* [Eswaran 76] is introduced. An execution is serializable if it is computationally equivalent to a serial execution. That is, a serializable execution has the same effect as some serial execution. Since serial executions of transactions preserve consistency, so do serializable executions. There are two methods to guarantee the serializability of transactions: locking and timestamps. In this report, we employ locking.

1.2 Transactions as a Programming Tool

Although many transaction implementations currently exist, especially in database management systems [Borr 81] [Gray 79], these implementations have two significant limitations which make them unsatisfactory for a general programming tool in a distributed environment. First, existing transaction mechanisms are typically implemented as part of an application-level program such as a database manager. As a result, it is not possible for other clients of the system to use the transaction mechanism. Consider an application that invokes some database actions as well as performing several file updates directly. In case of abort, the database system undoes its updates, but it is the application program's responsibility to deal with its own actions in a way that is synchronized with the database system's behavior. For this reason, the transaction facility should be provided in the underlying operating system so that it is generally available. Given a transaction facility implemented in this manner, the example of the application calling the database system is straightforward to handle.

More importantly, in existing implementations, transactions cannot be initiated from within transactions. While not significant in a database management system, this restriction prevents users from constructing new transaction applications by

composing existing transactions. The programmer should instead be able to write transactions as *modules* which may be composed freely, just as procedures and functions may be composed in ordinary programming languages.

1.3 Nested Transactions

Transactions which are composed of other transactions, or *nested transactions*, have been the subject of much current literature [Moss 81] [Liskov 82] [Reed 78] [Svobodova 81]. A transaction invoked from within a transaction, or *subtransaction*, appears atomic to its caller. That is, the operations it performs take place indivisibly with respect to both failures and concurrent computations, just as for traditional transactions. Thus a nested transaction mechanism must provide proper synchronization and recovery for subtransactions. Such a mechanism guarantees that concurrently executing transactions are serializable. Another property of nested transactions is that subtransactions of a given transaction fail independently of their invoking transaction and of one another, so that an alternate subtransaction may be invoked in place of a failed subtransaction in order to accomplish a similar task. It has been pointed out that many applications naturally lend themselves to being implemented as nested transactions [Gray 81] [Moss 82].

1.4 Purpose of the Report

This report describes an original algorithm for full nested transactions and its implementation within the Locus distributed operating system [Walker 83] [Popek 81]. Full nested transactions builds on the simple nested transaction mechanism reported in [Moore 82a] and [Moore 82b]. Although the implementation has been produced in the Locus environment, we believe that the algorithms described in this report are generally applicable.

The majority of the design and implementation reported in this document is the work of Erik Mueller, with the following exceptions. An important part of the mechanism, the two-phase commit protocol which is used to atomically commit a top-level transaction, was designed and implemented by Johanna Moore. The mechanisms necessary to support remote process forking were designed and implemented by August-Wilhelm Jagau.

1.5 Points of Novelty

To our knowledge, this is the first actual implementation of nested transactions on a distributed system. So far, others have produced only a preliminary, centralized implementation as part of the Argus language [Liskov 82] and a centralized simulation of a distributed implementation [Moss 81]. Further, as discussed in Chapter 3, our nested transaction mechanism provides additional functionality beyond that which is usually proposed. First, our mechanism allows transparent access by transactions to objects at sites other than the site on which the transaction is executing. Second, unlike other environments, Locus allows replication of data objects at more than one site in the network, and this capability is incorporated into the transaction mechanism. If the copy of an object that is currently being accessed becomes unavailable, it is possible to continue work by using another one of the replicated copies. Third, transactions may be composed of processes running at many sites in the network, without the need to invoke separate subtransactions to accomplish work at the many sites. Finally, an efficient orphan removal algorithm is presented, and the problem of providing continued operation during network partitions is addressed in detail.

1.6 Contents of the Report

This document is organized as follows. Chapter 2 describes the particular distributed environment which this report assumes, namely that of the Locus distributed operating system. Chapter 3 describes the model of nested transactions which is presented to the programmer and describes those aspects of our model which differ from other nested transaction models. Chapter 4 presents the basic implementation of nested transactions, which is extended in Chapter 5 to deal with network partitioning. Chapter 6 explores several extensions and optimizations which can be made to the nested transaction algorithm developed in the preceding chapters. Chapter 7 describes the implementation of top-level transaction commit using the two-phase commit protocol. Chapter 8 looks at optimizations to the commit protocol described in the previous chapter. Chapter 9 reviews the work of others in order to put our work in proper context. Finally, Chapter 10 presents conclusions and suggestions for further research. Appendix A contains a detailed description of the nested transaction algorithm developed in Chapters 4 and 5. Appendix B presents summary statistics, including a breakdown of the number of lines of code in our implementation, and measurements of the performance of the mechanism.

CHAPTER 2

THE LOCUS ENVIRONMENT

This report assumes a particular distributed environment, namely that of the Locus distributed operating system [Walker 83] [Popek 81]. In this chapter, we first describe the underlying architecture on which Locus is built. We then give an overview of the Locus operating system, describe the Locus file system, and conclude with a discussion of the Locus partition manager.

2.1 Underlying System

A distributed system consists of a collection of processors, called *sites*, which communicate via the communications subsystem. Failures of both the sites themselves and the communications network are expected to occur. However, we expect all failed components to recover eventually. Our protocols view the permanent loss of a site as intolerable behavior and hence do not attempt to deal with such failures.

The sites in our environment are expected to be heterogeneous ranging from personal workstations with little or no permanent memory to large mainframes with vast amounts of permanent storage. Each site has both volatile and stable storage.* When a site fails, the state of the processor and the contents of volatile memory are

* It is not strictly necessary to require that each site have its own stable storage. A site which has little or no stable storage capacity could use that of another site in the network by arranging a suitable protocol with that site. However, we will assume that if this is the case, the protocol to make this occur will be hidden within the operating system of that site and hence, to our protocols, it will look as though the site itself has stable storage. We will not provide special protocols to handle this case. Thus, without loss of generality, we will assume for the remainder of this discussion that each site in the network has stable storage. Although Locus does not yet provide such a protocol, future plans include such support.

lost, while the contents of stable storage remain intact. Stable storage has two properties: (i) stable storage is reliable, i.e., its only acceptable behavior is to store and retrieve information correctly as instructed by the processor, and (ii) updates to stable storage take place atomically, i.e., each update must appear to have occurred in its entirety or not at all. Our notions concerning stable storage have come from the work of Lampson and Sturgis who have designed a method for implementing stable storage of arbitrarily high reliability using magnetic disks [Lampson 79]. Gray has suggested several alternative schemes in [Gray 78]. The actual form of implementation is not important for the remainder of our discussion and will not be considered further.**

We have stated that permanent loss of a site is viewed by our protocols as unacceptable behavior and as such would not be dealt with in our algorithm. What we really view as intolerable behavior is the unrecoverable loss of data stored in a site's stable storage and not of the site itself. Thus the problem comes down to building stable storage on a device which can easily be moved to another site, should the site it is currently connected to fail, and building a mechanism by which the system can handle such migration. In fact, the Locus operating system contains such a mechanism. This mechanism, known as *pack porting,* works in the following manner. In Locus, all data objects are associated with a *pack* and all packs are uniquely identified within the network by a *global pack identifier.* The system keeps a network-wide table mapping global pack identifiers to site identifiers. In this manner, it is possible to determine the site at which a pack currently resides. Whenever a

** Currently a strict form of stable storage is not implemented in Locus. However, Locus does provide an atomic file update capability. In our implementation we store and update all information which must be relied upon to survive crashes on magnetic disks as files. We assume that true stable storage will eventually be implemented in Locus.

pack migrates, the table is updated.* Thus through the use of stable storage accompanied by pack porting, we can, in practice, reduce the probability of the behavior which we consider intolerable to an arbitrarily small factor.

We assume no particular topology of the communications network. The network could consists of Ethernets, ring networks, connected by multiple gateways, and so on. However, a level of software called the *partition manager* enforces certain constraints regarding the communications system which our algorithms rely on. This will be discussed in the last section of this chapter.

Sites communicate using *messages* which are transmitted from a source site to a destination site by the communications system. Each message contains the source site, destination site, and a stream of data. We assume that messages are not altered by the communications system. Error-checking codes such as cyclic redundancy checks or checksums may be used to discard any messages which have been corrupted. Although such techniques cannot eliminate *all* corrupted messages, they can eliminate such messages with arbitrarily high probability. Thus we assume that messages are delivered to their proper destination and that the stream of data in the message remains unaltered.

We do not otherwise assume the communications system to be particularly reliable. The communications system may lose messages, may deliver duplicate messages, may deliver messages in a different order than the order in which they were sent, and may delay messages arbitrarily. However, we do assume that if a message is repeatedly sent, it will eventually reach its destination.

* See [Reiher 82] for the details of pack porting in Locus.

If techniques such as sequence numbering are used to eliminate duplicate, delayed, and out-of-order messages at the low level, i.e., that of the communications system, our high-level algorithms must still cope with lost messages. However, in order to cope with lost messages, our algorithms will have to retransmit messages. As a result, duplicate messages will be generated at the high level, and must be dealt with. Thus it can be seen that handling duplicate messages at the low level is wasted effort, since they will be generated and dealt with at the high level in any case. This is an example of the end-to-end argument [Saltzer 81].

2.2 The Locus Operating System

Locus is an integrated distributed operating system providing a high degree of network transparency, while at the same time supporting high performance and reliability. Locus makes a collection of computers connected by a communications network look to the user and application program like a single UNIX* [Ritchie 78] system. For example, there is one tree-structured hierarchical name space for files and one may run processes locally or remotely with identical semantics. In Locus, files may be replicated, i.e., one may store a file at several sites. The system in operational use at UCLA consists of a set of VAX 11/750 computers connected by a standard Ethernet.

Locus provides for graceful operation during network partitions, i.e., the situation where various sites in the network cannot communicate with each other for some length of time due to network or site failures. This is a very real problem in a distributed system. A partition occurs if a site becomes disconnected from the network for some reason, such as if the site's network interface fails. In some cases it may even be desirable to operate in partitioned mode, for example if the site is a

* UNIX is a Trademark of Bell Laboratories.

personal workstation which connects to the rest of the network by an expensive long-distance telephone line. General partitions are possible in an environment where local networks are connected by gateways. Even in the Ethernet, gateways can be inoperative for significant lengths of time while the Ether segments they normally connect operate correctly. Locus employs a sophisticated merge algorithm which allows sites to leave and return to the network gracefully without interrupting service [English 83].

One of the most important goals which drove the design of Locus was the desire to make the development of distributed applications no more difficult than single-machine programming. In fact, users of a Locus network have the illusion that they are operating on a single machine. The network is essentially invisible. Users have no need to refer to a specific site or to the network itself. There is a uniform interface to all resources in the distributed environment independent of their location. For example, if *open(filename)* is the method used to access local files, then it is also the method used to access remote files. Thus the network becomes invisible in much the same way that virtual memory hides secondary store. An extensive discussion of the principle of network transparency may be found in [Popek 83a].

Distributed systems providing redundant resources have substantial potential for reliable, available operation. However, in order to realize this potential, it must be possible to substitute alternate versions of resources when the original is corrupted or unavailable. For the sake of availability, operation must be allowed to continue in the face of network partitions. If the resources for an operation are available in a given partition, that operation must be allowed to proceed even if some of the required resources are replicated in other partitions. Although this policy can easily lead to consistency conflicts at partition merge time, we believe it is a reasonable

policy for the following reasons. First, we believe that conflicts of this type are likely to be rare, since actual sharing in computer utilities is known to be relatively infrequent. Furthermore, an algorithm has been developed to detect conflicts if they have occurred [Parker 83]. For those data objects whose update and use semantics is simple and well understood, it may be quite possible to reconcile the conflicting versions automatically [Faissol 81] [Faissol 83]. Our philosophy regarding replication and recovery is discussed in greater detail in [Popek 83b].

2.3 Locus File System

Locus provides for the manipulation of permanent objects called *files*. In this section we summarize the Locus file system; a complete discussion may be found in in [Walker 83]. As in UNIX, files are organized into sections of mass store called *file systems*. However, in Locus file systems may be replicated, i.e., we may store a file system at several sites. Files may be replicated at any of the sites storing the containing file system. However, replicating a file system does not imply that copies of all files contained in that file system must be stored at each such site. Files are uniquely identified in the network by a *filename*, which is the pair <FileSystem, FileNumber>. Each copy of a replicated file is uniquely identified by the pair <FileName, PackNum>.*

For each file system in a given partition, one of the sites storing the file system is designated the *current synchronization site* (CSS) for the file system. This site manages synchronization for all of the files contained in the file system. That is, all *open* requests involve a message to the CSS. It is not necessary for the CSS to be the site from which data access is obtained. Any site which has a copy of the file can

* Recall that the pack porting mapping scheme allows one to determine the site where a pack is located from the pack number; see [Reiher 82].

provide the data. Such a site is called the *storage site* (SS) for the open request.

An outline of how actual file operations are handled will serve to clarify the Locus synchronization mechanism and the roles played by the various sites participating in a file operation. When an *open* request for a particular file is issued, the user-level character string pathname is translated into a system-level filename at the requesting site.** This name is then sent from the requesting, or *using site* (US) to the CSS in an open request message. If the file is not locked in a conflicting mode, the CSS selects an SS and sends it a message. After the SS decides to provide service, it replies to the CSS, which records appropriate synchronization information and notifies the US. The US installs appropriate information and the open is complete. The US may now read and write file pages by conversing directly with the SS. When the US wishes to commit any modifications it has made to the file, it sends a commit message to the SS causing the new version of the file to replace the old version of the file in stable storage at the SS. As part of the commit operation, the SS sends messages to all the other SSs of that file as well as the CSS. At this point, the other SSs will bring their version of the file up to date by propagating the changes made to the file. When the US is finished using the file, it sends a close message to the SS which causes it to send a similar message to the CSS. Access information kept at the SS and the CSS is appropriately updated.

2.4 Locus Partition Management

Locus is designed to operate gracefully in the face of network partitions, i.e., when a group of one or more sites is isolated from other sites in the network due to communications failures or site outages. When the network is partitioned, we assume that the collection of sites making up the network is broken up into a number

** See [Walker 83] for details of this name translation.

of disjoint sets of sites. We assume that any site in a given partition can communicate with every other site in that partition, and that no site in a partition may communicate with a site which is not in that partition, i.e., that members of a partition are strongly connected.

This model of network partitioning differs in some respects from what may actually occur and some extra measures must be taken to enforce this model. For example, a communication link may fail in only one direction (or input buffers at one site may become full effectively causing communication failure), allowing one-way communication between a pair of sites.

In order to ensure that our model is accurate and that the protocols which we have designed based on this model operate correctly, Locus employs a partition management algorithm which enforces the partition model we have described above. Each site maintains a table of those sites with which it can communicate, called the *site table*. A *logical partition* is defined as the subnetwork indicated by the sites in the site table. The site table is managed by the partition management software and may lag behind the actual physical state of the network while topology changes are in progress.

The partition management software enforces the requirements of our model in the following manner. Before sending a message, the site table is checked to see if the intended recipient is available. If so, the message is queued to be sent out over the network. If not, the process attempting to send the message is notified of this condition.

Although, in general, messages will not be sent to sites which are not in the site table and transmissions from such sites will not be accepted, some special mes-

sages must be allowed to bypass this constraint so that new sites can be added to a partition, i.e., so that partitions may merge. Thus the partition management software employs a set of special messages which can be sent to and received from sites currently assumed to be unavailable. In this manner, sites may join a given partition.

Sites which are no longer available are detected in the following way. When attempting to transmit to a remote site, a timeout with retransmission strategy is used. If an acknowledgement from the remote site is not received before the specified time interval has elapsed, the message is retransmitted. This scheme is repeated a number of times until the retransmission limit is exceeded. Once this limit is exceeded, the recipient site, which has not responded to these repeated communication attempts, is assumed to be down and the site table is updated to reflect this topology change.

Whenever any sites join or leave the network, a *topology change procedure* is run. This procedure manages the site table and maintains consistency of the system data structures. One of its tasks in this respect is to locate any processes waiting for messages from sites which have become inaccessible, and to notify them of this event. This prevents processes from waiting indefinitely for messages from sites which have crashed or been partitioned away. Another important task of the topology change procedure is to select new CSSs for each file system and to reconstruct locking information. In addition, *recovery* must be performed for any conflicting file copies, i.e., they must be reconciled. For a complete description of the partition management software, see [English 83]. The algorithms employed in Locus are discussed in greater detail in [Edwards 82].

A large portion of this report is devoted to the design of topology change algorithms for properly maintaining consistency of the system data structures used by the nested transaction mechanism. In designing strategies for dealing with network partitions or site outages in our transaction system, we make the assumption that if a site is partitioned away, it is likely that the site will stay partitioned away for quite a while. Thus we abort anything which cannot continue work because of the partition and attempt to recover to a state where work can continue. We do not wait indefinitely for the inaccessible sites to become accessible again. We also free any resources which are held by processes not in our partition, so that processes which *are* in our partition may access the resources.

CHAPTER 3

NESTED TRANSACTION MODEL

This chapter describes the Locus model of nested transactions. Our model of nested transactions differs in significant ways from traditional views. We discuss these differences in the first section. In the sections which follow, we explain the model of nested transactions which is presented to the Locus user, including transaction invocation, completion, and file access. Finally we discuss the user-visible results of network partitioning and conclude with a summary of the chapter.

3.1 Differences From Other Models

In this section, we describe how our model of nested transactions differs from other models. First we discuss our philosophy regarding replication, and then we discuss how network transparency makes our model different from others.

3.1.1 Replication

Other models of nested transactions insist that replication of data objects be built on top of the transaction mechanism [Moss 82]. However, the mechanism for replication of objects at more than one site in the network is provided within the Locus file system. Thus our transaction mechanism assumes that each object may consist of several replicated copies. During partitioning of the network, transactions may continue gracefully if copies of the necessary objects are available within the partition. Of course, serious consistency problems can result when partitions merge and a given object has been independently updated in several partitions. Difficulties

are even worse if that object has already been used by other transactions as a basis to update other objects. Nevertheless, there is potentially great value in permitting these transactions to execute during a partition. First, an algorithm has been developed to detect any conflicts that have occurred upon partition merge; see [Parker 83] and [Parker 82]. Second, for many applications, including airline reservation and banking, it is usually possible, and feasible, to automatically reconcile all data values at partition merge time. The problem of automatic reconciliation of data in the context of network partitioning is dealt with extensively in [Faissol 81] [Faissol 83]. This work breaks the semantics of transaction operations into several classes, develops reconciliation algorithms for each class, and claims that most real cases of data management fall into the simpler of these classes.

One suspects that in many systems, automatic reconciliation will be feasible for the large majority of data objects. For those applications for which automatic reconciliation is not feasible, a scheme such as voting [Menasce 77] [Thomas 78] or primary sites [Alsberg 76] may be implemented at the application level, in order to limit object accesses to at most one partition. Of course, there will remain cases that require human intervention, such as when an external action has been taken that cannot be undone and for which a compensating action cannot be taken. These cases are the same ones for which general-purpose data management recovery is also impossible. The problem of consistency of application data objects in the environment of network partitions is beyond the scope of this report.

3.1.2 Network Transparency

Other models assume that a transaction directly modifies objects residing only at the site on which the transaction is executing [Moss 81] [Liskov 81]. In order to modify objects at another site, it is necessary to invoke another transaction at the

remote site. In contrast, we believe that the location of both data objects and processes should be transparent to the programmer. Thus a transaction may directly access objects at any site, the same way that local objects are accessed. The object may in fact be replicated at more than one site. Similarly, our model does not require that all processes within a single transaction execute at the same site as in other models [Moss 81] [Svobodova 81] [Liskov 82]. A transaction may transparently consist of processes at several sites just as it may be composed of several processes at a single site. In addition to transparency considerations, the ability to execute closely-cooperating processes within a transaction at multiple sites and the ability to access remote objects directly without artificially imposed mechanisms substantially improves transaction performance.

3.2 Transaction Invocation

We now explain the model of nested transactions which is presented to the programmer. In this section we discuss transaction invocation. The process model we use in this discussion is the same as that of UNIX in which a process may invoke another process only by creating a replica of itself, an operation known as *forking* a process. The new process, called a *child process*, can distinguish itself from the original process, the *parent process*, by the return value of the fork operation. The child process inherits all the open files of its parent. A process may wait for its children to complete via the *wait* system call.

A process starts a transaction with the following site-transparent* call:

relcall(load-module, args)

* Specific site requests are performed with a context mechanism, such as that proposed in [Popek 83a].

This system call causes *load-module* to be executed as a transaction with command-line arguments *args*, which is a list of character strings. Typically, *args* gives the names of files which are the input data objects to the transaction. Open files are inherited from the process issuing the call.** The call waits until the transaction completes and then returns with a completion code. When a transaction is started with *relcall*, it consists of only one process, the *top-level process* of the transaction. This process, however, may fork locally or remotely giving rise to transactions consisting of more than one process. Each process that is a part of a transaction, including the top-level process, is called a *member process*.

Processes running as part of a transaction are permitted to invoke other transactions. A transaction invoked by a process running as part of a transaction is called a *subtransaction*, since it is a transaction invoked from within a transaction. Subtransactions are frequently referred to merely as transactions. In Figure 1, transactions T_3 and T_2 are subtransactions of T_1 and transaction T_4 is a subtransaction of T_3. The transaction whose initiator is not a transaction, in this case T_1, is called a *top-level transaction*. To speak of the related group of transactions, i.e., a top-level transaction and all of its subtransactions, the term *entire transaction* is used. In the figure, the entire transaction consists of T_1, T_2, T_3, and T_4.

** Strictly speaking, inter-process communication through shared files or pipes should not be permitted between running transactions since this violates indivisibility of transactions. (However, communication is allowed between the closely-cooperating processes making up a transaction.) Thus passing open files with relcall is discouraged. Nevertheless, it is sometimes necessary for users to interact with running transactions and for error messages to be printed on an output device or logged to a file. Passed open files must only be used for these purposes. Any file operations performed on an open file passed from a caller are treated as if they were performed by the transaction that originally opened the file. When a transaction wishes to work on a file, it must open the file itself so that proper locking and recovery may be performed. It is not difficult to extend our algorithms to allow a transaction to explicitly pass open files to a subtransaction, so that it is not necessary to close a set of files before invoking a subtransaction that uses those files.

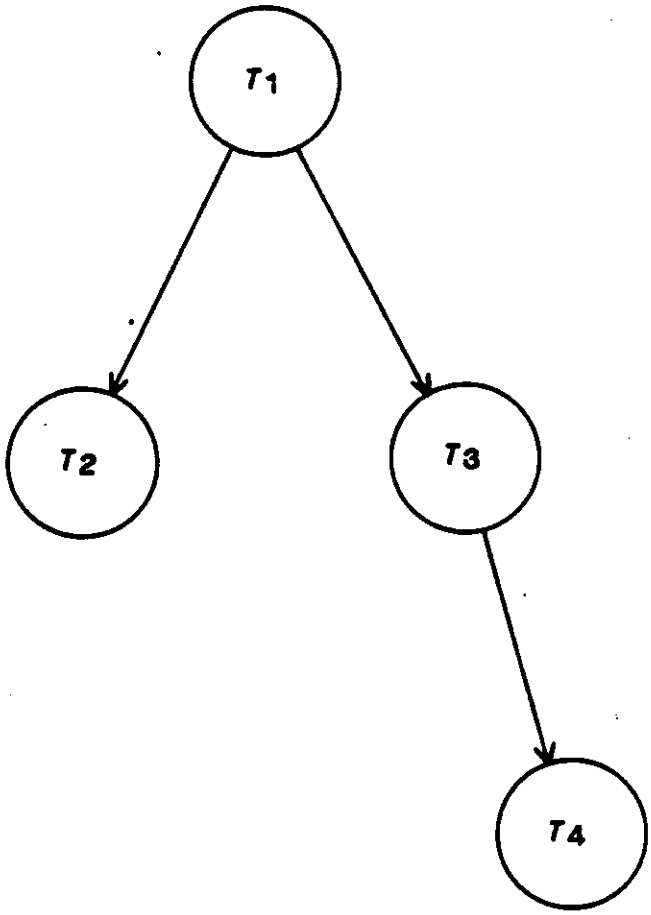

Figure 1: Transaction Tree

Tree terminology will be used in discussing relationships between transactions. When a transaction calls a subtransaction, the calling transaction is called the *parent* of the subtransaction, and the subtransaction the *child* of the calling transaction. In our example, transaction T_1's children are T_2 and T_3, and T_1 is the parent of T_2 and T_3, and so on. We also speak of *ancestors* and *descendants*. A transaction is an ancestor and descendant of itself. Thus the ancestors of T_4 are T_4, T_3, and T_1. The descendants of T_1 are T_1, T_2, T_3, and T_4. We also use the terms *superior* and *inferior*. A transaction is neither a superior nor an inferior of itself. Thus the superiors of T_4 are T_3 and T_1. The inferiors of T_1 are T_2, T_3 and T_4.

3.3 Transaction Completion

A transaction completes either by committing or aborting. A transaction may commit only after its children have all completed. However, a transaction may abort at any time. Processes terminate by issuing the *exit* system call which has an argument indicating the success or failure of the process. Thus in order for a transaction to commit, its top-level process must issue an exit call with a successful completion code. However, if a top-level process issues an exit call with an unsuccessful completion code, the transaction aborts. In addition, if a top-level process terminates before all other member processes have terminated, the transaction aborts. This is required to enforce the UNIX programming convention that each process waits for its children before terminating.

A subtransaction will be said to commit if certain operations which must be performed by the system complete successfully. If, for example, one of the objects accessed by the subtransaction has become inaccessible because of a network partition, then the commit will fail. The actual committing of any updates performed by the subtransaction is contingent upon the commit of each superior transaction all the

way up to the top-level transaction. If a superior transaction aborts, then the updates of all descendants of that transaction will effectively be undone. Thus no updates performed within an entire transaction are made permanent until the top-level transaction commits. A top-level transaction will commit if the two-phase commit protocol [Gray 78] [Lindsay 79] [Lampson 79] reaches the commit point. After a transaction commits or aborts, control returns to the process that invoked the transaction.

Since a calling process waits for the completion of a transaction, a single process may invoke at most one transaction at a time. A transaction may initiate several subtransactions concurrently by simultaneously invoking subtransactions from several of its member processes. For example, the following transaction code in the C language [Kernighan 78] executes two subtransactions subtrans1 and subtrans2 in parallel, committing only if both subtransactions commit:

```
if (fork() == CHILD) {
        exit(relcall("subtrans1", args1));
}
return_code2 = relcall("subtrans2", args2);
wait(&return_code1);
if ((return_code1 == COMMIT) && (return_code2 == COMMIT))
        exit(COMMIT);
else
        exit(ABORT);
```

This code, which we assume is running as the top-level process of a transaction, forks a child process which calls subtransaction subtrans1 and then exits with the transaction completion code. At the same time, the parent process invokes subtransaction subtrans2. When subtrans2 completes, the parent process waits for its child process to terminate and is passed its completion code, which corresponds to whether subtrans1 committed or aborted. The parent process then instructs the system to commit only if both subtransactions committed, and to abort otherwise.

3.4 File Access

A transaction requests a lock on a file with the *open* system call and releases it with the *close* call. A transaction holding a lock on a file reads and writes data with the *read* and *write* system calls. When a transaction commits, the transaction's caller and all the caller's inferiors see the updates of the transaction. If a transaction aborts, the updates of the transaction are undone.

In order to guarantee serializability between transactions, the locking rules of [Moss 81] are extended (we have added the last rule and slightly modified the others):

* A transaction may open a file for modification *(hold* a lock in write mode) if no other transaction holds the lock (in any mode) and all *retainers* of the lock are ancestors of the requesting transaction.
* A transaction may open a file for read (hold a lock in read mode) if no other transaction holds the lock in write mode and all retainers of write locks are ancestors of the requesting transaction.
* When a transaction aborts, all its locks are simply discarded. If any of its superiors hold or retain the same lock, they continue to do so, in the same mode as before the abort.
* When a transaction commits, all its locks are inherited by its parent (if any). This means that the parent retains each of the locks (in the same mode as the child held or retained them).
* When a transaction closes a file (for the last time), the held lock becomes a retained lock.

It has been noted in [Moss 81] and [Liskov 82] that if parents of transactions are prevented from running when their children are running, then the distinction between held and retained locks is not required. There is no easy way to enforce this policy in C without also preventing a transaction from starting concurrent subtransactions--if we stop all transaction processes whenever one of them invokes a subtransaction, then only one subtransaction may be invoked at a time. The programming language Argus [Liskov 82] solves this problem by introducing a language

construct used only to start several concurrent subtransactions.*

3.5 Network Partitioning

We now consider what happens when a *network partition* occurs, i.e., if one or more sites leave the current partition. Under certain conditions, such an occurrence will cause some transactions to be aborted. First, if a transaction is separated from its caller, the following will occur. If the transaction is a subtransaction, it is aborted and its caller is made aware of this fact by an appropriate completion code of *rel-call*. However, if the transaction is a top-level transaction, the caller is notified that the transaction has been partitioned away, although it is impossible to determine whether the transaction has committed or aborted.** Second, if a transaction holds or retains a lock for a file which has become inaccessible, the transaction is aborted. If another copy of the file is accessible in the current partition, transactions left unaborted by the network partition may then open the file.

3.6 Summary

In this chapter we have presented the model of nested transactions which is provided to the programmer. For a sample application which has been run using Locus nested transactions, see [Mueller 83b].

---

* Specifically, the construct is

```
action foreach element in list
        subtransaction(element)
end
```

The meaning of this construct is: For each element of a list, concurrently invoke a subtransaction with that element as its argument.

** There are two simple solutions to this problem. One is to build a mechanism to record completed top-level transactions. The other is advise users not to invoke remote top-level transactions.

Our model differs from traditional views in two significant ways. First, we assume that files may be replicated at more than one site in the network. For the sake of availability during network partitions or site failures, we allow a transaction to access a file if one of the replicated copies of the file is available. Although this policy can lead to consistency problems, in many cases such conflicts can be reconciled automatically. Second, in order to ease programming in a distributed environment, we have argued that the network should be invisible to most programs. Remote resources, such as processes and data, should be used the same way as local resources are used. We adhere to the concept of network transparency in our transaction mechanism. Thus unlike most other models, we allow a transaction to access data at any site the same way it accesses local data. Similarly, since transactions may be composed of closely-cooperating processes located at a single site, they may just as easily be composed of processes executing at many sites in the network.

In the chapters which follow, we present our implementation of the nested transaction mechanism described in this chapter. Chapter 4 presents the basic implementation of nested transactions. A detailed discussion of top-level transaction commit is deferred until Chapters 7 and 8, and the handling of network partitions is deferred until Chapter 5. In Chapter 6, we present several extensions and optimizations to the algorithms developed in Chapters 4 and 5.

CHAPTER 4

BASIC IMPLEMENTATION

This chapter describes the basic implementation of nested transactions in Locus. Several aspects of the implementation will be deferred until later chapters to simplify the presentation of our algorithms. We believe that our algorithms depend little on the Locus operating system and could be adapted to many distributed environments. However, the design and implementation of our mechanism was greatly simplified by the high degree of network transparency which Locus provides and by the partition management and recovery schemes which are part of Locus.

We have made the assumption that the underlying communications system may lose, duplicate, delay, and reorder messages. However, these characteristics of the communications network will be ignored in this chapter in order to simplify the description. We will assume here that messages are not lost, except in the case of network partitioning, and that they are not duplicated, delayed, or reordered. In Chapter 6, we will show how our protocols may be amended to handle these communications failures.

4.1 Transaction Data Structures

The site on which a transaction begins executing is called the *transaction home site*. Each transaction is uniquely identified in the network by its *transaction unique identifier* (Tid). We assume that it is possible to determine the home site of

a transaction from its Tid.* Processes are uniquely identified in the network by a *process unique identifier* (Pid), from which one may determine where the process is executing. Associated with every process executing at a site is a *process structure* which, among other information, contains a flag which indicates whether the process is executing on behalf of a transaction, and if this flag is set, the Tid of the transaction. Processes running as part of a transaction may fork, giving rise to transactions which have more than one member process. Our implementation has been designed in an environment which allows processes to fork remotely as well as locally. Thus it is possible for a transaction to have member processes at a site other than the transaction home site. Such processes are called *remote member processes.* However, in this chapter we assume that all processes making up a transaction reside at a single site. In Chapter 6, we will amend our algorithm to handle remote transaction processes.

Associated with each transaction, be it a top-level transaction or a subtransaction, is a volatile data structure called the *transaction structure* which resides at the transaction home site:

Trans	=	Struct[Tid, Super, Status, Pid, Members, Files]
Super	=	List[Tid]
Status	=	Oneof[UNDEFINED, COMMITTED, ABORTED]
Members	=	List[Struct[Pid, Subtrans]]
Files	=	List[Struct[FileName, PackNum, Mode]]
Subtrans	=	Oneof[NULL, Tid]
Mode	=	Oneof[READ, WRITE]

* Tids may be implemented as the pair <HomeSite, LocalUid>, where HomeSite is the transaction home site and LocalUid is an identifier unique to the transaction home site. Thus the transaction home site generates a Tid by generating a local unique id, and pairing it with its site identifier. However, in our implementation, we will require the generation of a Tid by the site invoking the transaction, not the home site of the transaction. Thus the above pair is not easy to generate without a message exchange between the transaction home site and the generating site. Instead, we employ the triple <HomeSite, GeneratingSite, Uid>. We are simply generating a globally unique id <GeneratingSite, Uid> and pairing it with some extra information–the transaction home site. Note that such a Tid scheme does not allow transaction migration. However, we do not allow transaction migration in the current mechanism.

The transaction structure contains the Tid of the transaction and a list of the Tids of the transaction's superiors. If the transaction is a top-level transaction, the list of superiors is empty. The status of a transaction is UNDEFINED from the time it is initiated until its fate is determined, at which time its status will be changed to COMMITTED or ABORTED. A Pid identifies the process which invoked this transaction and thus indicates where to return control when this transaction completes. The Members field contains a list of the member processes of the transaction. This list is called the *member process list* and it contains an entry for each process making up the transaction. Each entry contains the Pid of the process and any active subtransaction of the process. An active subtransaction is a subtransaction which has been invoked, but has not yet completed. The Files field contains a list of the files involved with the transaction, i.e., the files for which the transaction holds or retains locks. This list is called the *participant file list* and each of its entries contains the filename and a pack identifier which together uniquely identify a physical copy of a file. Mode indicates the type of access (READ or WRITE) the transaction has to this file.

4.2 Transaction File Operations

For simplicity, we will only discuss file operations performed by transactions. Non-transaction file operations in Locus are treated in [Walker 83] and [Edwards 82]. In this section, we first discuss the protocols which are used to perform the various file operations, and then we describe file recovery and locking in more detail.

4.2.1 File Protocols

As mentioned earlier, a transaction process requests a file lock via the *open* system call. Data is read and written with the *read* and *write* system calls, and a

lock is released with *close*. Thus the following operations must be provided to a transaction process:

```
Open(FileName, Mode, Tid, Super)
Read(FileName, Index, Tid) => Data
Write(FileName, Index, Data, Tid)
Close(FileName, Tid)
```

The site of the transaction accessing a file is called the ***using site*** (US).

When a file is first opened by a transaction, one of the sites storing the file is designated the *transaction synchronization site* (TSS) for the file. This site manages synchronization for the file and provides data access.* Other copies of the file are brought up to date after top-level transaction commit. The TSS provides access to the file and performs locking decisions for transactions using the following operations:

```
TssOpen(Tid, Super, Filename, Mode)
TssRead(Tid, Filename, Pagenum) => DataPage
TssWrite(Tid, Filename, Pagenum, DataPage)
TssClose(Tid, Filename)
TssCommit(Tid, Super, Filename)
TssAbort(Tid, Super, Filename)
```

When a transaction process invokes the Open operation,** a message is sent to the TSS for the file.*** Upon receiving the message, the TSS executes the TssOpen rou-

---

* In Locus, the site which manages synchronization for a file (CSS) may be different from the site which provides data access (SS). However, since our transaction algorithms require the site which provides data access to be the same as the site which manages file locking once the file is open for modification by a transaction, we do not make this distinction here. Our implementation could easily be extended to allow many SSs for a file, but only if the file is not being modified by any transaction. In this case, transactions must still be aborted if the TSS for a file becomes inaccessible, as will be discussed. However, if only SSs become inaccessible, while the TSS remains accessible, an alternate SS may be substituted and no transactions need be aborted as a result.

** We assume that any pathname searching has already been performed; see [Walker 83].

*** From now on, we will speak of sending messages with the understanding that if the site to which we are sending the message is local, we really do not send a message. Instead, we directly invoke the appropriate routine.

tine and makes a locking decision. The results of the decision are returned to the US. If the open was successful, the US adds the file to the transaction's participant file list and returns control to the caller of the open system call. For a US to read or write a data page, a message which contains the Tid of the transaction is sent to the TSS. When a US closes a file, it sends a message to the TSS and waits for a response. The close causes the transaction's held lock to become a retained lock. Finally, when a transaction commits or aborts, it informs the TSS.

In Locus, all open requests are first directed to the CSS for the file. If the file is not already open, the CSS chooses a TSS, and that TSS is used for subsequent opens until the top-level transaction commits or if all transactions involved with the file abort. There is an optimization to this Locus open protocol. If a US wishes to open a file and the US happens to be the TSS for the file, the locking checks may be performed directly without contacting the CSS.

4.2.2 Recovery and Locking

To enable recovery in the event of transaction abort, for each file modified by a transaction we must save the state to be restored should the transaction abort. Following is a discussion of what state information must be kept and how file state restoration is accomplished. When a file is first opened, the current version (or state) of the file is maintained in volatile storage at the TSS. When a transaction opens a file for modification, a copy of the current version of the file is saved. When the held lock is released and becomes a retained lock, this copy, along with the Tid of the transaction, is pushed onto a *version stack* stored in volatile memory. The version stack is a stack of file versions, with an entry for each transaction retaining a write

lock.* The locking rules constrain write retainers to form a line in the transaction invocation tree, and thus a stack, rather than a tree, may be used. A transaction opening a file for read only access will not cause a new entry to be added to the version stack.

The operation performed when a transaction commits is called a *commit-lock-update*. For retained read locks, the parent of the committing transaction (if any) inherits the retained read lock, unless it already retained a read or write lock. For retained write locks, if the parent of the committing transaction has an entry on the version stack, the committing transaction's entry is merely popped. If the parent of the committing transaction does not have an entry on the version stack, then the committing transaction's entry becomes the parent's entry. That is, the Tid of the version on the top of the version stack is changed to that of the committing transaction's parent.

The operation performed when a transaction aborts is called an *abort-lock-update*. Retained read locks are simply removed. For retained write locks, the entry on the top of the version stack is popped and this version of the file from the stack replaces the current version of the file, thus recovering the file to its state prior to modification by the transaction.

For example, suppose transaction T_1 has invoked transaction T_2, and both transactions have modified a file F, as shown in Figure 2. Since both transactions have modified the file, both have an entry in the version stack for file F. F_0 is the original state of the file, F_1 is the state of the file after T_1 has performed its

* Each entry in the version stack does not have to be a complete version of the file. For each entry, enough information is required in order to restore the file to the proper state should the transaction fail. In the implementation of version stacks in Locus, we are able to save file versions incrementally, i.e., only those file pages that are new since the last version need be recorded in the new version. This scheme is described in Chapter 6.

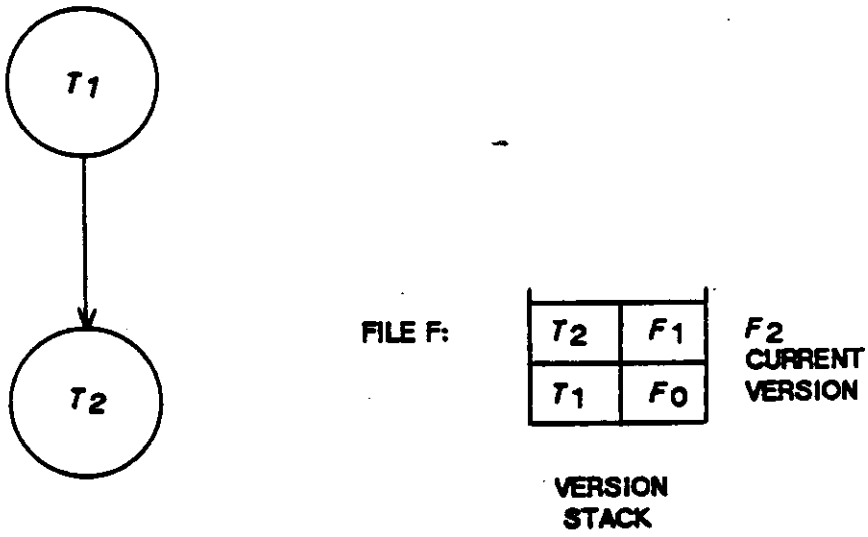

Figure 2: Initial Version Stack

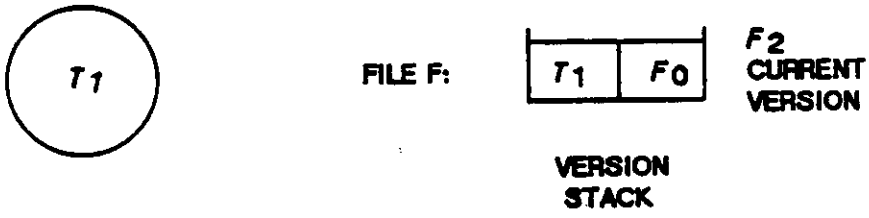

Figure 3a: Version Stack If T_2 Commits

Figure 3b: Version Stack If T_2 Aborts

modifications but before T_2 has performed its modifications, and F_2 is the state of the file after T_2 has performed its modifications. F_2 is the current state of the file at this point. Now suppose T_2 commits. In this case, the entry for T_2 on top of the version stack is simply popped and discarded as shown in Figure 3a. If T_2 instead aborts, the version for T_2 which is on top of the version stack is popped and replaces the current version as shown in Figure 3b.

Locking decisions are performed by the TSS because the transaction locking information is stored there. The locking information is stored there because the version stack is a part of that information and the version stack is best kept at the site where modifications are recorded.

The locking and recovery information maintained at the TSS for a file involved with a transaction is referred to as a *t-lock*. A t-lock consists of the current volatile file state as possibly modified by transactions, a list of held and retained locking transactions, and file state restoration information for write locks:

```
Tlock =      Struct[FileState, Holders, ReadRetainers, WriteRetainers]

Holders          =    Oneof[NULL, ReadHolders, WriteHolder]
ReadHolders      =    List[Struct[Tid, Super]]
WriteHolder      =    Struct[Tid, Super, FileState]
ReadRetainers    =    List[Struct[Tid, Super]]
WriteRetainers   =    Struct[VersionStack, Ancest]
VersionStack     =    List[Struct[Tid, FileState]]
Ancest           =    List[Tid]
```

For each transaction holding a read lock, the t-lock contains the transaction's Tid and the Tids of the transaction's superiors. If a write lock is held by a transaction, the t-lock contains the Tid, the superiors, and the state of the file that was current before the write lock was obtained. For each transaction retaining a read lock, the t-lock contains the Tid and the superiors. For write retainers, a version stack is kept. The t-lock also contains a list of ancestors of the top element of the version stack.

The need for lists of superiors and ancestors in the t-lock structure will be explained in the next chapter.

The TSS denies a request to open for modification if any other transaction holds a lock or there is a retainer of a lock that is not an ancestor of the requesting transaction. An open request for read access is denied if any other transaction holds a write lock or there is a retainer of a write lock that is not an ancestor of the requesting transaction. These locking checks are necessary to assure serializability; see [Moss 81].

In Locus, a file may be open several times by a single process. Because of this, and since a file may be open for read by more than one transaction at the same US, the US must keep a separate reference count for each transaction which indicates the number of times the file is open by that transaction. When the count for a transaction becomes zero, a close message is sent to the TSS.

A transaction may lock a file in the current partition even if there are other copies of the same file in other partitions. Given this policy, there is the possibility that when partitions merge there will be more than one TSS maintaining lock information for a particular file. This situation is called a *t-lock conflict*. The topology change software is responsible for electing new CSSs, gathering the locking information from all accessible TSSs, and handling any such lock conflicts. Methods for handling lock conflicts are discussed elsewhere; see [Edwards 82], [English 83], and [Rudisin 80]. Essentially, the two conflicting locks will both be handled, and recovery will be invoked when the operations complete, just as if the partition merge had occurred after the operations had completed.

4.3 Transaction Control

In this section, we describe the basic control of transactions, i.e., transaction invocation and completion. In the next two sections, we will detail our algorithms for transaction commit and abort.

4.3.1 Transaction Invocation

This section describes how a process invokes a transaction. The process which initiates the transaction is called the *calling process*. If the calling process is running as part of a transaction, the invoked transaction is a subtransaction. Otherwise, the invoked transaction is a top-level transaction.

In order to invoke a transaction, the following steps are performed at the site of the calling process. First, a Tid is generated for the new transaction. If the calling process is itself running on behalf of a transaction, we note in the member process list of the caller's transaction structure that the calling process has a subtransaction, i.e., we record the called transaction's Tid. Now, depending on where the user (or context mechanism) wishes the transaction to be run, we either invoke a local routine to start the transaction, or converse with a remote site. If we are starting a remote transaction, we pass to the remote site the name of the load module to be executed, the command-line arguments, the Tid of the new transaction, the Tids of its superiors, and the Pid of the calling process. The details of forking a remote process are dealt with elsewhere [Jagau 82].

At the home site of the new transaction, we create a transaction structure for the new transaction and fill in the fields appropriately. We enter the Tid of the new transaction and the Tids of its superiors, initialize the transaction status to UNDEFINED, note the Pid of the calling process, create a member process list containing

one entry--an entry for the process we are about to start--and enter an empty participant file list, since as yet the transaction has no participants. In addition, in the process structure of the new process we note that this process is running on behalf of a transaction and indicate the Tid of this transaction. Finally, we mark the new transaction process as runnable so that it may be scheduled to execute.

While the invoked transaction is running, the calling process waits for the called transaction to complete. Once the called transaction has committed or aborted, control will be returned to the calling process, as will be described.

If a transaction process performs a fork operation, we record the fact that this transaction now has an additional member process. We accomplish this by adding the Pid of the newly created process to the member process list in the transaction structure.

4.3.2 Transaction Completion

In order for a transaction to be able to commit, its top-level process must exit with a successful completion code. If the top-level process exits with an unsuccessful completion code, the transaction aborts. The member process list is used to manage the processes which make up a transaction and to determine when they have all terminated. Each time a member process of a transaction terminates, we remove its entry from the member process list associated with that transaction. If a top-level process terminates before all the other member processes have terminated, then we force all the other member processes to terminate in order to abort the transaction, as will be described.

When the member process list becomes empty, i.e., all the member processes have terminated, the transaction is ready to complete. We will return to the problem of committing and aborting a transaction in the following sections. After the transaction has either committed or aborted, control is returned to the calling process. If the calling process is running as part of a transaction, it has an associated member process list. In this list, there is an entry corresponding to the calling process whose subtransaction Tid is the Tid of the just completed transaction. We remove the subtransaction Tid from this entry. Finally, at the home site of the just completed transaction, we remove the transaction structure associated with the transaction.

4.4 Transaction Committing

We take different actions to commit a transaction depending on whether the transaction is a top-level transaction or a subtransaction. We will discuss subtransaction commit first, and then we will describe the commit of top-level transactions.

4.4.1 Subtransaction Commit

If the transaction is a subtransaction, we must commit-lock-update all the files in the subtransaction's participant file list, in order to pass the transaction's locks to its parent. Thus our algorithm for subtransaction commit performed at the home site of the subtransaction is as follows. We first check to see if all TSSs for files in the participant file list are accessible. This is intended to minimize the chance that partitioning will be detected during the subtransaction commit procedure. If any are not accessible, we abort the subtransaction and send an SUBABORT message to the home site of the parent transaction. If however, all are accessible, we send a REQCOMMIT message to the home site of the parent transaction, containing our

participant file list. This message informs the parent transaction home site that the invoked transaction wishes to commit and allows participant files of the called transaction to be added to the parent's participant file list. If a particular file is already in the parent's list, the stronger mode of that which is in the parent's list and that which is in the child's list is taken. The parent transaction must be aware of the participant files from this point on, so that it can properly recover should the commit fail as a result of subsequent partitioning. After adding the files to its list, the parent sends a GRTCOMMIT message to the child transaction home site, informing it that it may commit.

Upon receiving the GRTCOMMIT message, the child home site accomplishes the commit-lock-updating of files required for subtransaction commit by sending a TSSCMT message to all sites acting as TSSs for files in the participant file list. The TSSCMT message contains a list of files to commit-lock-update. Each site receiving the message performs the commit-lock-update and returns a RTSSCMT response message along with a success code. If the child transaction home site receives a RTSSCMT response message from all sites and all sites have succeeded,* the subtransaction has successfully been committed and a SUBCOMMIT message is sent to the parent home site. When the parent home site receives the SUBCOMMIT message, control is returned to the calling process.

If the child home site does not receive all RTSSCMT response messages because of partitioning or if a site was unsuccessful at performing the updates for some reason, a SUBCMTFAIL message is sent to the parent home site. Upon receiving this message, or if the child home site becomes inaccessible to the parent home site

* This operation can be performed in parallel: We send out all TSSCMT messages and then wait for all RTSSCMT messages to come back before proceeding. In the rest of our discussion, whenever we speak of sending out several messages and waiting for responses, we imply this parallel algorithm.

after the parent sends the GRTCOMMIT message, the parent transaction must abort itself in order to recover properly.

Thus we consider a subtransaction committed if and only if we are successful at commit-lock-updating all the files in the subtransaction's participant file list. To ensure this requirement, it might first seem that the commit-lock-updates must be accomplished using a two-phase commit protocol, since the lock update is a distributed operation--the lock information is located at the TSSs for the files which can be at any site on the network.

However, two-phase commit is an expensive and complicated operation that we wish to avoid. Fortunately there is a way to avoid two-phase commit at the cost of some reliability in the slim window of subtransaction commit. We notice that if a subtransaction commit cannot be completed once it is begun, either because we are partitioned away from a TSS or a TSS is unable to perform the commit-lock-update for some reason, we can recover simply by aborting the parent of the subtransaction.

There are two reasons why this works. First, the parent inherits any locks of the subtransaction and thus since the parent cannot commit until the commit of the subtransaction has completed, only the parent and inferiors of the parent can obtain locks to those files of the subtransaction that *have* been successfully commit-lock-updated. Thus the only transactions which may view some of the subtransaction's committed files are those which will be aborted.

Second, it is possible to properly recover the files and their t-locks. If a particular file was not commit-lock-updated, because subtransaction commit failed, then we must still have the version to restore when we abort the parent transaction. The version to restore is either in the version stack entry associated with the subtransac-

tion that was trying to commit, or in an entry for the parent if the parent modified the file. If, on the other hand, the file was commit-lock-updated in the subtransaction commit algorithm, then either 1) the version to restore was passed to the parent if the parent did not already have an entry in the version stack, or 2) the parent already had an entry in the version stack to restore. In both cases, we have the proper version to restore. Figures 4a and 4b show what happens if the version stack has been commit-lock-updated, or not commit-lock-updated, for the cases in which the parent has an entry in the version stack and the case in which it does not. We show how the abort-lock-update initiated when the parent aborts recovers the version stack (and thus the t-lock) to the proper state. Our abort-lock-update algorithm must be enhanced to handle these cases of course; how to do this will be explained in the next chapter.

4.4.2 Top-Level Transaction Commit

Now we discuss the problem of top-level transaction commit. In this case, the transaction's participant files are first commit-lock-updated. This will cause all files that were only opened for read by the entire transaction to have their t-lock structure released at the TSS.* If any commit-lock-update fails, we must abort the top-level transaction, and this is accomplished by abort-lock-updating all participant files.** If the entire transaction has modified a particular file, then after the top-level commit-lock-update the version stack will be empty, and the current version is the one that we wish to commit to stable storage. We invoke a distributed two-phase commit protocol to accomplish these updates atomically. The participant file list

* In Locus, the CSS is informed at this point, unless transactions which are outside this entire transaction also retain or hold read locks.

** This is feasible at the top level because although we have discarded the version to restore from the version stack, we still have the original version on disk. The abort-lock-update algorithm must be able to deal with this case.

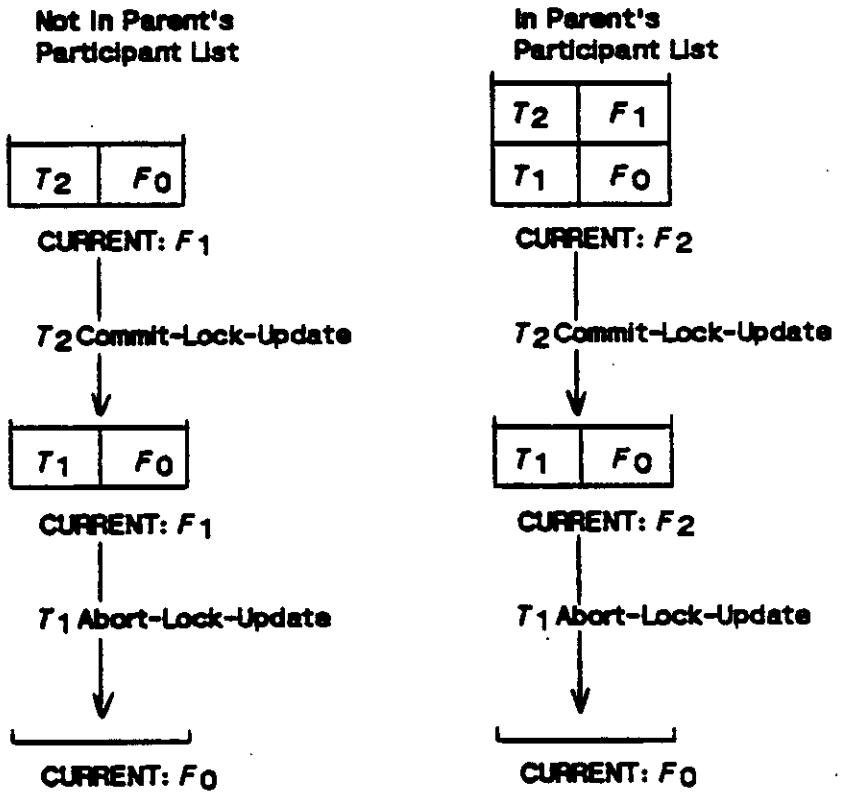

Figure 4a: Commit-Lock-Update Performed

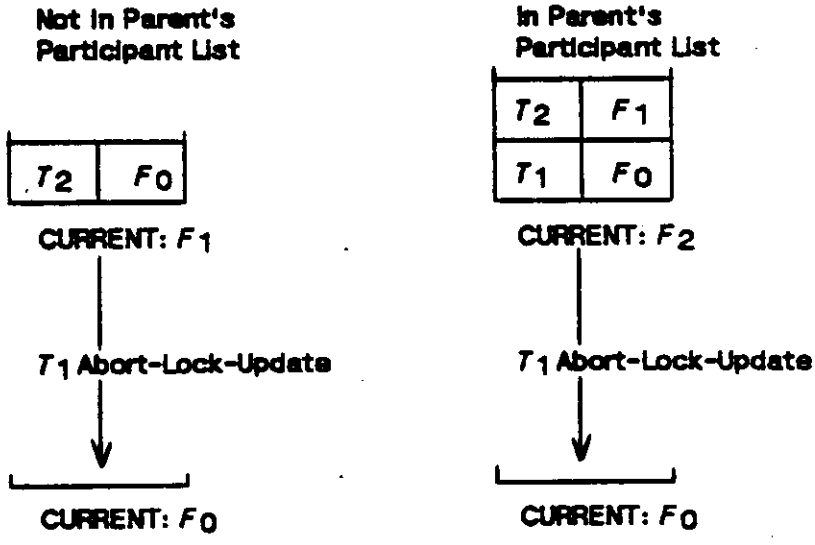

Figure 4b: Commit-Lock-Update Not Performed

minus the files that were only opened for read becomes the participant list for the two-phase commit protocol. We defer discussion of the two-phase commit protocol until Chapters 7 and 8. The commit-lock-update algorithm may be incorporated into the first phase of the two-phase commit protocol, but this is ignored here for simplicity.

Once the first phase of the protocol is complete, we return control to the calling process by sending a TOPCOMMIT message to its site. If the protocol fails or if the top-level transaction is aborted for some other reason, a TOPABORT message is sent to the site of the calling process. It is impossible for the caller to know for sure whether the transaction has committed or aborted if a partition occurs before the caller has been notified (unless we want to wait until we contact the site again, and even then we have to add more mechanisms to remember the fate of old transactions). Even if we add a message exchange to ask the caller if we may commit, a partition after this exchange and before the final completion message is received will still leave the caller in doubt as to whether the transaction has committed. This may be a good reason to advise users not to invoke top-level transactions remotely.

4.5 Transaction Aborting

Although a transaction may commit only when all of its children complete, a transaction may decide to abort at any time. Thus an aborting transaction may have running descendant subtransactions. In order to abort a transaction and each of its running descendants, we wish to kill all the transaction's processes, abort each of the transaction's running descendants, and abort-lock-update the transaction's participant files. This releases locks held or retained by the transaction and if the transaction held or retained a write lock, restores the current version of a file to its state prior to modification by the transaction.

Thus our abort algorithm is as follows. First we destroy each of the transaction's member processes. We then send a FORCEABT message to each of our running children and wait for RFORCEABT responses. After we have received all responses, we send a TSSABT message to each site having a participant file of the aborting transaction and wait for RTSSABT responses. A child receiving a FORCEABT message in turn follows the same algorithm, destroying *its* member processes, aborting *its* children subtransactions by sending FORCEABT messages, waiting for responses, sending out TSSABT messages and waiting for responses, and finally returning a RFORCEABT response. In the absence of partitions, this algorithm will abort all descendants of the aborting transaction and cause files to be abort-lock-updated in the proper order. Handling network partitions will be discussed in detail in the next chapter.

If two transactions decide to abort at the same time, where one transaction is a superior of the other, the inferior transaction will receive a FORCEABT message when it has already initiated an abort. In this case, the inferior simply waits for the existing abort operation to complete before responding. Alternatively, this situation is handled by the mechanisms developed in the next chapter.

CHAPTER 5

HANDLING NETWORK PARTITIONS

This chapter discusses algorithms for dealing with network partitioning. First, we extend our abort algorithm to handle partitions. Then we consider the problem of aborting transactions that are separated from their calling transaction home sites as a result of a network partitions. This problem has come to be known as the "orphan problem" in the literature. Finally, we treat the situation in which a TSS for a participant file is partitioned away from the transaction.

5.1 Extensions to Abort Algorithm

If an aborting transaction cannot send a FORCEABT message to its child transaction because that child is partitioned away, the aborting transaction must ignore that child in its abort procedure. As a result, when the aborting transaction abort-lock-updates its files, some of those files may be locked by the inaccessible child and its inferiors. Thus we need the capability of abort-lock-updating a file for all descendants of a transaction. In addition, since any of the inferiors of such an inaccessible child may also attempt to abort themselves, we must be able to ignore abort-lock-update messages for updates that have already been performed. This is required in case the aborting superior has completed the abort-lock-update before the inferior attempts the update.

In order to achieve these two capabilities we keep a list of superiors for each transaction holding or retaining a lock. For write locks, we need only keep a list of

the superiors of the top element of the version stack. Now we must modify our abort-lock-update algorithm as follows. In order to abort-lock-update the locking information for all descendants of an aborting transaction, we simply search through all holders and retainers and abort-lock-update those having the aborting transaction as an ancestor. If the abort-lock-update has already been performed, no such holders or retainers will be found. For write locks, we can perform the following optimization: If the aborting transaction is an ancestor of the transaction associated with the top element of the version stack, we simply pop elements off the stack until the aborting transaction is associated with the top element and then pop it off and restore the current version of the file from this entry.

If a subtransaction commit fails, we must abort the parent of the subtransaction that we were attempting to commit. In this case, the aborting transaction may not necessarily be on the stack and thus we must amend this algorithm to pop elements off the stack until the stack is empty or the top element is a superior of the aborting transaction. The restored version is that of the last popped entry.

There is an optimization which we can now make to the abort algorithm given in the preceding chapter. This optimization will enable us to send only one abort-lock-update message per file involved in a branch of aborting transactions. The extension to the algorithm is to pass the transaction's participant file list to children in the FORCEABT message. The children then only abort-lock-update files not in the passed down participant file list. When the children send FORCEABT messages to their children, they add their participant file list to the participant file list passed to them from their parent. In this way, lock information is updated only once in each branch of the transaction invocation tree.

5.2 Orphan Removal

If a network partition occurs, we wish to abort any transactions which no longer have a path in the transaction invocation tree to the top-level transaction. We wish to eliminate such *orphan transactions* and abort-lock-update files for which they hold or retain a lock.

Our algorithm for accomplishing the abort of orphan transactions is driven by both the transaction home sites and the TSSs. As part of the topology change procedure at a transaction home site, if a superior transaction home site is inaccessible, we destroy all of the transaction's member processes and remove the transaction structure, thus aborting the transaction without abort-lock-updating files or aborting child subtransactions. We call such an abort a *silent abort*. The abort of a child subtransaction is effected when the topology change procedure detects the same condition for the child subtransaction, i.e., one of its superiors is inaccessible.

The abort-lock-updating of files is accomplished by the TSS: Since each lock held or retained by a transaction has associated with it a list of all the transaction's superiors, we can drive the lock cleanup from the TSS as follows. The topology change procedure simply goes through all t-locks at the site, abort-lock-updating any locks held or retained by a transaction which is inaccessible or a transaction which has a superior which is inaccessible to the TSS. There is an optimization for write locks: We need only find the holder or bottommost retainer in the version stack which is inaccessible or has a superior which is inaccessible to the TSS and abort-lock-update the file for that transaction. This causes the stack to be popped up to and including that point, restoring the appropriate version of the file. This scheme assures us that locking information is properly updated in the case of partitions.

Note that when the topology change procedure abort-lock-updates a file, the transaction holding a lock to the file may continue to run for a short period of time and may attempt to read or write the file. Since each such read or write request contains the Tid of the requesting transaction, requests from orphan transactions whose locks have been released may be denied.

Our lock cleanup strategy is correct because if a transaction is inaccessible to the TSS, then the transaction must eventually abort since one of its files is inaccessible.* If one of the transaction's superiors is inaccessible to the TSS but the transaction *is* accessible, then by network transitivity a superior of the transaction is inaccessible and the transaction will silently abort.

An example will serve to clarify our orphan removal algorithm. Assume transaction T_1 invoked T_2 which invoked T_3 which invoked T_4. Assume these transactions each execute at a different site and that each retains a write lock for a particular file F, whose TSS is at yet another site. For brevity we will refer to the sites as T_1, T_2, T_3, T_4, and F. Suppose now that T_2 and T_3 leave our partition and that those two sites can communicate in a new partition, i.e., the network is organized as the partitions $\{T_1, T_4, F\}$ and $\{T_2, T_3\}$, as shown in Figure 5a. We can see that we would like T_2, T_3, and T_4 all to be aborted, and for their retained locks to be abort-lock-updated so that the only retainer of a write lock is T_1. The following actions will be performed by the topology change procedure. Since transactions T_2, T_3, and T_4 all have an inaccessible superior, they are all aborted as described above. The bottommost

* So far, we know that a transaction cannot commit if one of its files is inaccessible. However, suppose the file becomes accessible again, just before the commit. In this case, if we have abort-lock-updated the file as part of our cleanup strategy, then this can be detected when the transaction attempts to commit and an error code is returned. In this case the transaction will not commit. Thus our strategy is self-satisfying in that if we abort-lock-update the file, then it is impossible for the transaction holding a lock to the file to commit. Of course, we would not wish to perform such an update unnecessarily. In the next section, we will describe a method whereby transactions having inaccessible TSSs are immediately aborted.

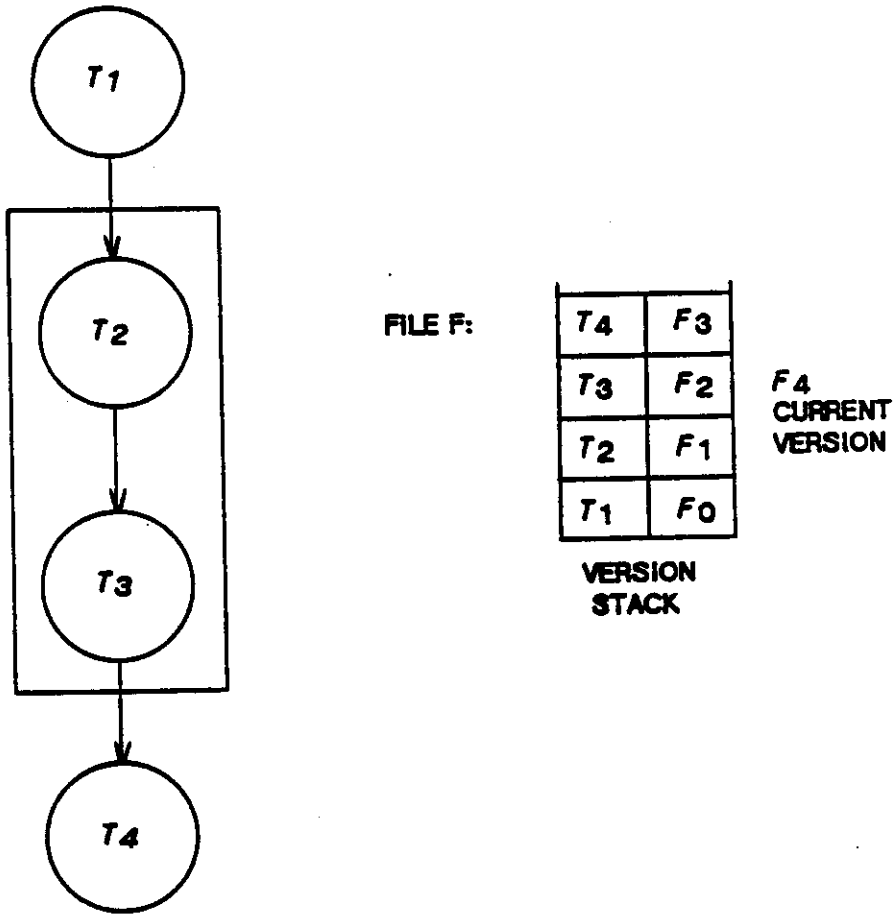

Figure 5a: Before Orphan Removal

Figure 5b: After Orphan Removal

retainer in the version stack (highest in the transaction invocation tree if the root transaction T_1 is at the top) that is inaccessible or has a superior which is inaccessible to the TSS is T_2. Thus we abort-lock-update the file for T_2. Now the only remaining retainer of a write lock for the file is T_1, as shown in Figure 5b.

Imagine a variation of the previous example in which only T_4 retains a write lock for F. The same transactions will be aborted, however now T_4 is the bottommost retainer in the version stack with an ancestor that is inaccessible to the TSS. In this case, we abort-lock-update the file for T_4, leaving no lock holders or retainers and thus the t-lock structure may be removed at the TSS.

Now we move on to an example in which the file is separated from the top-level transaction. We have the same situation as in our first example, except that T_2, T_3, and F leave our partition and that those three sites can communicate in a new partition, i.e., our network is organized as the partitions $\{T_1, T_4\}$ and $\{T_2, T_3, \mathrm{F}\}$. What we would like in this case is for T_1, T_2, T_3, and T_4 all to be aborted, and for all their retained locks to be abort-lock-updated so that there are no retainers left and the t-lock is released at the TSS. The following actions will be performed by the topology change procedure. Since transactions T_2, T_3, and T_4 all have an inaccessible superior, they are all aborted. The retainer bottommost in the version stack which is inaccessible or has a superior that is inaccessible to the TSS is T_1. Thus we abort-lock-update the file for T_1 which causes the t-lock to be freed. Now, transaction T_1 will eventually abort because one of its files is inaccessible.

If our orphan removal algorithm aborts a transaction, the transaction's caller must be notified. This is accomplished by adding the following function to the topology change procedure. For any process having an active subtransaction whose home site is inaccessible to the calling process, an abort completion code is returned

to the waiting process. Note that returning control to a transaction may be unnecessary if that transaction or one of its superiors is also aborting as a result of the partitioning. Thus we should not return control unless all the transaction's superiors are accessible.

In the scheme that we have described, there are two outstanding problems which must be dealt with. First of all, when a lock on a file is requested, it may be impossible to grant the requested lock because aborted transactions that have not yet completed their abort algorithm hold or retain a conflicting lock on the file. This problem can be handled either by waiting and later retrying the lock request, or requiring the TSS to query the home site of the transaction supposedly holding or retaining a conflicting lock. If the response to the query is that the transaction is ABORTED or NONEXISTENT, we can clean up the transaction's locks and its descendants' locks for all files at the SS. This will generate extra message traffic for opens that truly are lock conflicts; however, these may be rare. It is probably sufficient simply to retry up to some limit, as is done in our current implementation, since in the normal case the calling process of an aborting transaction does not regain control until the abort has completed, i.e., all locks have been properly updated. That is, in the normal case a transaction invoked as an alternate to an aborted transaction which wishes to lock some of the same files as the aborting transaction will not begin execution until the abort has completed. It is only when a transaction is separated from its child that the alternate transaction may request a lock before the abort has completed. In addition, note that orphan extermination takes only as long as it takes the topology change procedure to run and this may be short.

The second problem is that when a transaction wishes to commit and it begins to commit-lock-update its files, it may have to deal with locks that are held or

retained by aborted transactions whose aborts have not yet completed. This can be handled simply by abort-lock-updating any inferiors of the committing transaction before performing the commit-lock-update. This strategy works because all descendants of a transaction must be resolved--either committed or aborted--in order for the transaction to commit, and therefore any descendants still holding or retaining locks may be considered aborted since if they committed they would not still hold or retain locks.

5.3 Inaccessible Synchronization Sites

If a TSS for a file becomes partitioned away, in general many transactions will as a result be unable to commit. Any transaction having the inaccessible file as a participant will thus abort. We may wish to force such an abort immediately upon detecting that a TSS is inaccessible. Suppose we added a condition to our topology change procedure which immediately aborts a transaction if any of its participant files is inaccessible. This will accomplish what we desire, however this algorithm is inefficient: Since some of the transactions having inaccessible files have superiors who also have inaccessible files, we will perform many simultaneous aborts, generating unnecessary processing and network traffic. For example, if transaction T_1 invoked T_2 which invoked T_3, and all three transactions have file F as a participant, then if F is separated from T_1, T_2, and T_3, we will abort T_1, T_2, and T_3.* We need only abort T_1. Thus what we would like to do is to abort simply the *topmost* transaction involved with the inaccessible file. What this means is clear in the case of write locks: The transaction which is the topmost retainer or holder (bottommost in the version stack) of a write lock must be aborted. It is not immediately clear what this means in the case of read locks, since they do not form a line.

* We cannot silently abort the transactions, because this condition does not cause all transactions in a branch to be aborted. It only causes those with inaccessible TSSs to be aborted.

Let us present a method for determining who is a "topmost" involved transaction. When a file is opened for the first time by a transaction, a *topmost token* for the file is passed to the transaction. We will call a transaction that has such a token a *topmost retainer* of the file. This token is passed to a parent when a transaction commits and discarded when the transaction aborts. When a file which is already locked by a transaction is opened, then a topmost token is given to the transaction if no superiors of the transaction hold or retain a lock to the file. In the case of write locks this means that no other transactions hold or retain locks. In the case of read locks this is the topmost retainer in one branch of the transaction tree, distinct from other branches.

Given this, we abort a transaction if a TSS is inaccessible and the transaction is a topmost retainer of the file. This causes the write retainer bottommost in the version stack for a file to be aborted; otherwise there are no write retainers and each of the topmost read retainers are aborted.

Note that this algorithm does not prevent multiple aborts caused by more than one file becoming inaccessible to several transactions where different transactions are topmost retainers of different files. For example, if T_1 is the topmost retainer of F_1, T_2 is the topmost retainer of F_2, T_1 is the parent of T_2, and now if F_1 and F_2 become inaccessible both to T_1 and T_2, then both T_1 and T_2 will be caused to abort.

CHAPTER 6

EXTENSIONS AND OPTIMIZATIONS

This chapter explores several extensions and optimizations which can be made to the nested transaction algorithm we have developed in Chapters 4 and 5. First, we describe the extensions which are necessary to support remote member processes. Second, we describe the modifications which are necessary in order to handle lost, duplicated, delayed and out-of-order messages. Finally, we discuss the optimization of version stacks in Locus.

6.1 Remote Transaction Processes

This section presents the extensions which must be made to our algorithms in order to support remote transaction processes. We first deal with the implications with respect to the control of transaction member processes, and then we cover file locks.

6.1.1 Member Process Management

The extension to manage remote transaction processes uses the transaction home site as a centralized coordinator for the transaction's member processes. In this way, we can limit the impact on other parts of our algorithm. As an example of this strategy, consider the following situation. Suppose the home site of a transaction T_1 is S_1, and a remote member process of the transaction is executing on site S_2. Now suppose that S_1, the transaction home site, becomes inaccessible to S_2, the site where the remote member process of the transaction is executing. In this case, we

wish to abort T_1. At S_2 we simply destroy the remote member process and do nothing else. That is, we do not take actions to abort the transaction in S_2's partition, because existing mechanisms already handle this. Any inferior transactions of T_1 whose home sites are in S_2's partition will be silently aborted when the topology change procedure detects that the home site of their superior T_1 is inaccessible. Similarly, any files locked by T_1 or inferiors of T_1 will be abort-lock-updates by the topology change procedure.

We now describe what additions must be made to our member process control algorithms. First note that the member process list is already capable of listing processes which are not at the home transaction site. This is because Pids are unique in the network and one may determine the site where a process is executing from its Pid. If a local transaction process forks a remote process, we may simply add the Pid of the remote process to the transaction's member process list as usual. However, if a remote transaction process forks locally or remotely, the remote site must send an ADDMEMB message containing the Pid of the new process to the transaction home site and wait for its response in order to add a new entry to this list. When a remote member process terminates, the remote site must send a REMOVEMEMB message to the transaction home site and await a response in order for the home site to remove the entry for the remote process from the member process list.

If a remote member process wishes to start a subtransaction, the remote site first sends a STARTSUB message containing the new transaction's Tid to the transaction home site and waits for a response. When the subtransaction of the remote process completes, a FINISHSUB message must be sent to the transaction home site and a response awaited in order to remove the entry for the subtransaction from the remote process's entry in the member process list. With these additions, the transac-

tion structure is always kept up to date. If responses were not used, then a carefully timed partition might allow a member process or subtransaction to be created without the message being received by the home site. In this case it would be quite impossible for the home site to destroy all member processes and abort all subtransactions.

The transaction home site is responsible for accomplishing commit and abort as previously described, except that the algorithm for aborting a transaction must be modified as follows. If a running member process of the transaction is remote, a KILLMEMB message containing the Pid must be sent and a response awaited in order to destroy the process.

It is possible for a remote member process of a transaction to become partitioned away from the home site of the transaction. In this case, the remote process must be destroyed and the transaction must be aborted, as in our example above. Thus our topology change procedure must perform the following two functions. If a remote process of a transaction becomes inaccessible to the home site of the transaction, the transaction must be aborted. Conversely, if a transaction home site becomes inaccessible to the remote process, the process must be destroyed.

6.1.1 File Management

The addition of remote member processes also adds additional complexity to our file management algorithms. In order to allow a transaction to access a file from both the home site of the transaction as well as other sites that are running processes of the transaction, we must make the following changes to the t-lock data structure:

```
ReadHolders  = List[Struct[Tid, Super, UsingSites]]
WriteHolder  = Struct[Tid, Super, UsingSites, FileState]
UsingSites   = List[Site]
```

That is, we keep a list of USs for each transaction holding a read or write lock. The US must be recorded since now the site accessing the file is not necessarily the transaction home site. A list is required since the same file may be open from the same transaction at more than one US. We modify our algorithm so that when a US sends a close message to the TSS, the US is removed from the list of USs. When the list becomes empty, the transaction's held lock on the file becomes a retained lock.

In addition, recall that if a process forks remotely the child process inherits the open files of its parent, as in UNIX. Since the file is open from more than one site, *file access tokens* are required to coordinate access to the file. This is discussed in detail in [Jagau 83].

We now consider the possible partitionings between the transaction home site, remote member process site, and TSS, and show how our topology change procedure allow us to recover from each case. First, if the site of a remote member process using a file is partitioned from the TSS, there are two cases to consider. Suppose the transaction home site is still accessible to the TSS. Then by transitivity of the network, the remote member process site is inaccessible to the transaction home site. As a result the transaction will be aborted, causing the file to be abort-lock-updated for the aborted transaction. Otherwise, suppose the transaction home site is not accessible to the TSS. In this case, the topology change procedure will abort-lock-update the file as usual, since the home site of an ancestor of the transaction holding or retaining a lock on a file is inaccessible.

We must also consider what happens if the remote member process site can still communicate with the TSS, but they are both partitioned away from the transaction home site. In this case the topology change procedure again will abort-lock-update the file and destroy the member process which is separated from its home

site.

There is one more change which must be made to our algorithms to support remote member processes. The participant file list, which keeps track of all files accessed by a transaction, is stored at the transaction home site, and must be kept current. This is accomplished as follows. When a remote member process opens a file, the remote site must send an ADDFILE message to the transaction home site and wait for a response. This allows the transaction home site to add a file to its participant file list.

If the site of the remote member process crashes or is partitioned away from the transaction home site before it can send the ADDFILE message, then a file will be open by a transaction but not listed in the transaction's participant file list. There are two cases to consider. If the network partition separates the transaction home site from the TSS for this file, our existing algorithms will abort-lock-update any files held by the transaction. However, if this is not the case, our algorithms will not abort-lock-update these files. To handle this case we must enhance our topology change procedure to abort-lock-update any file having a US that is partitioned away. This works because if the transaction home site is not partitioned away, then by transitivity of the network the remote transaction site is inaccessible to the home site, and thus the transaction will be aborted.

6.2 Handling An Unreliable Network

So far in our design we have ignored the possibility of lost, out-of-order, duplicate, and delayed messages. One expects lost messages to be rare in local networks. They can effectively be eliminated by keeping a copy of each outstanding message until a low-level acknowledgement for that message is received and re-

transmitting the message if an acknowledgement is not received within a certain timeout period.* Out-of-order, duplicate, and delayed messages may be eliminated using a sequence numbering scheme, and thus may be modeled as lost messages. Furthermore, lost messages can be modeled as a network partition, provided that they can be detected. Our algorithms are already robust in the face of network partitions.

However, simulating a network partition every time a message is lost is unsatisfactory, since it may cause unnecessary disruption of service. In addition, even if the communications network itself is reliable, a site may run out of buffer space, effectively causing messages to be lost. Thus our algorithms should be able to handle lost messages gracefully. In this section, we present modifications to our algorithms which make them robust in the case of lost, as well as out-of-order, duplicate, and delayed messages.

Lost messages are detected by using timeouts while waiting for a response. If a response is not received within a suitable timeout interval, the message in question is retransmitted until a response is received or the site is determined to be inaccessible, in which case our topology change procedure will handle recovery. We have already provided high-level responses to most of our messages. We have not yet provided responses to the TOPCOMMIT, TOPABORT, SUBABORT, REQCOMMIT, GRTCOMMIT, SUBCOMMIT, and SUBCMTFAIL messages. We will describe these messages shortly.

Retransmission requires our algorithms to be able to cope with duplicate, delayed, and out-of-order messages. To handle these cases, the operations performed by

* Here we assume that any message received by the low level will eventually be processed by high-level protocols except in the case of site failure. Our protocols are robust in the case of site failure.

the messages must be idempotent. In most parts of our algorithm, operations performed by sending messages can be made idempotent, usually by comparing unique identifiers such as Pids, Tids, and filenames. Once a response to a message is received, any future duplicate responses are simply ignored.

The ADDFILE operation which adds a file to the participant list is easily made idempotent by ignoring the request and sending a response if the file is already present in the list. The ADDMEMB, REMOVEMEMB, STARTSUB, and FINISHSUB operations, which add or remove Pids from the member process list and add or remove active subtransactions from an element of the member process list, can also be made idempotent as follows. For each element which has been removed, a dummy entry must be kept around so that delayed and duplicate add and remove messages may be ignored. The assumption here is that the same element will not be added and removed and then added again. This is true for member processes and subtransactions. We also assume that different elements can be distinguished. In fact we can, since Pids and Tids uniquely identify processes and transactions.

Now, when adding an element to the list, we ignore the request and send the appropriate response if the element is already in the list or a dummy entry exists for the element. When removing an element from the list, we ignore the request and send a response if a dummy entry exists for the element. If the element does not exist at all, then the remove message must have arrived before the corresponding add message. However, this situation cannot occur since our algorithms always wait for a response before continuing and thus the remove message would never be sent until the add message was acknowledged. Since we assume that a message cannot arrive before it is sent, i.e., that the system is causal, many out-of-order sequences such as this one can be ruled out.

Killing a remote process via KILLMEMB and forcing a transaction to abort with FORCEABT can be made idempotent as follows. If the process or transaction is already destroyed or aborted, the request is ignored and the appropriate response is sent.

Duplicate TOPCOMMIT and TOPABORT messages can be ignored and the appropriate response sent when either the calling process no longer exists, did not call a transaction, or if the Tid of the returning transaction does not match the Tid of the transaction which the process invoked. Duplicate SUBABORT messages may similarly be ignored when either the calling process no longer exists, the calling process does not have an active subtransaction, or the Tid of the returning transaction does not match the Tid of the process's active subtransaction.

The situation for the subtransaction commit protocol is more complex. In this protocol, which is described in Chapter 4, a child subtransaction which wishes to commit sends a REQCOMMIT message to its parent. The parent then grants the commit by responding with the GRTCOMMIT message. The child then attempts to commit and responds with either a SUBCOMMIT or SUBCMTFAIL message, depending on its success.

In the following discussion, we will say that a site is *interested* in a message if it is waiting for the message as part of its normal-case algorithm. Specifically, a site is interested in a REQCOMMIT message if there is a transaction process running at that site which has successfully started the subtransaction which is requesting the commit and the transaction is waiting for a REQCOMMIT or SUBABORT message from the subtransaction. Thus if the calling transaction's site has already received a REQCOMMIT message for the subtransaction, then the site is no longer interested in the message. In addition, if no such transaction exists, then the site is also not in-

terested in the message. A site is interested in a GRTCOMMIT message if there is a subtransaction at that site which has sent a REQCOMMIT message to its parent and is waiting for a GRTCOMMIT message. Thus if the subtransaction has already committed or aborted, the subtransaction's site is no longer interested in the message. A site is interested in the SUBCOMMIT and SUBCMTFAIL messages if there is a transaction process running at that site which has successfully started the subtransaction which is sending the message, and has sent a GRTCOMMIT message to the subtransaction,* and the transaction is waiting for the messages.

Now we specify our subtransaction commit algorithm which is immune to lost, delayed, repeated, and out-of-order messages. A subtransaction which wishes to commit sends the REQCOMMIT message and retransmits the message until a corresponding GRTCOMMIT message is received. A site which receives a REQCOMMIT message in which it is not interested simply ignores the message, sending no response. The parent's site which has received a REQCOMMIT message sends the GRTCOMMIT message and retransmits the message until a SUBCOMMIT or SUBCMTFAIL message is received. A site which receives a GRTCOMMIT message in which it is not interested also ignores the message, sending no response. A subtransaction which has succeeded or failed in its commit algorithm sends the appropriate SUBCOMMIT or SUBCMTFAIL message and retransmits the message until a RSUBCOMMIT or RSUBCMTFAIL response, respectively, is received. A site which receives a SUBCOMMIT or SUBCMTFAIL message in which it is not interested sends a RSUBCOMMIT or RSUBCMTFAIL response. Any unexpected RSUBCOMMIT and RSUBCMTFAIL responses are ignored.

* If a site receives the SUBCOMMIT or SUBCMTFAIL message for a transaction, it *must have* previously sent a GRTCOMMIT message to the transaction.

Messages sent to the US to the TSS to open, close, read, and write can be made immune to communications failures by attaching sequence numbers to the messages, as is done in Locus. It is difficult to avoid the use of sequence numbers in this case, since the order of updates and reads matters. Finally, the commit-lock-update and abort-lock-update operations which are initiated via the TSSCMT and TSSABT messages must be idempotent; we have already shown how to do this in Chapter 5 in order to cope with network partitions.

6.3 Version Stack Optimization

There is an optimization to version stacks which is employed in the Locus implementation of nested transactions. Locus uses a modified *intentions list* [Lampson 79] scheme for single-file atomic commit. In this scheme, a file is represented by an array of pointers to physical data pages in stable storage. This array is called the file's *inode*. Each entry in the inode array corresponds to a logical page of the file. When a file is opened at the TSS, a volatile copy of the inode, called an *incore inode*, is maintained. Associated with each pointer in the incore inode is a *modified* bit, which indicates if the page has been modified since being opened. When one wishes to modify a logical page of a file, the page is not directly modified. Instead, a new physical page is allocated, the data from the old page is copied into the new page, and then the page may be modified. The pointer in the inode is set to point to the new page and the corresponding modified bit is set. In order to commit changes, the file's volatile inode (with the modified bits removed) replaces the inode for the file in stable storage. In order to abort changes, the modified pages are freed and the volatile inode is simply discarded since the old inode is still in stable storage.

Using this mechanism, the implementation of version stacks can be optimized as follows. The current version of the file is represented by the *current inode* of the file. Each element of the version stack is an inode with modified bits as well. We will call the version on top of the stack the *top inode*. When saving the state of a file by pushing the current inode onto the version stack, the modified bits in the current version are reset. If a transaction aborts, we free those modified pages listed in the current inode, pop an element off the version stack and restore the current inode from the popped top inode. If a transaction commits, we free those pages which have been modified in the top inode but which are not present in the current inode. We also set the modification bit in the current inode for those pages which have been modified in the top inode and which *are* present in the current inode, and pop the element off the version stack. In this way, pages which are not needed are freed, and the current inode reinherits the modification bits for those pages which were modified since the previous version. Additional mechanism is also present to support large files that are structured through indirect pages that contain page pointers.

Thus version stacks allow us to incrementally save file versions: Only those file pages that are new since the last version need be copied.

CHAPTER 7

TOP-LEVEL TRANSACTION COMMIT

In this chapter, we discuss the implementation of top-level transaction commit. The scheme makes use of a two-level log structure and the well-known two-phase commit protocol. After a brief review of this protocol and an overview of the log structures, we will present a detailed description of the algorithm. We begin by considering only normal case behavior, leaving the discussion of how exceptional conditions are handled until last.

7.1 Two-Phase Commit Protocol

To commit a transaction, we employ the two-phase commit protocol which has been described by Gray [Gray 78], Lampson and Sturgis [Lampson 79], and Lindsay [Lindsay 79]. The two-phase commit protocol is used to allow multiple sites to coordinate transaction commit so that all participants in the transaction agree on the final outcome of that transaction, i.e., whether the transaction has been committed or aborted. In this protocol, one site must play the role of coordinator and it is the coordinator which, after conferring with each of the participants, will make the final decision regarding the fate of the transaction. During the first phase of the protocol, known as the *prepare phase*, the coordinator queries each participant to determine whether that participant can and will commit its portion of the multi-site, multi-file transaction. If all participants successfully prepare to commit, the coordinator will then make its decision and will command all participants to commit in the second phase of the protocol. If for any reason (e.g., communications failure, site

failure, or participant refuses to prepare for local autonomy reasons) a participant cannot or will not prepare, the coordinator will decide to abort the transaction and this will be done in the second phase.

7.2 Log Structure

As we have seen, the form of the two-phase commit protocol we have chosen utilizes a centralized coordinator to synchronize the activity of its distributed participants.* To facilitate the implementation of this protocol, we apply a two-level logging scheme. At the top level is the *coordinator log* which presents a global view of the transaction. This log includes the identities of all participants and the state of the transaction as it changes throughout the protocol. This collection of information resides in stable storage at the site designated as coordinator for this transaction.

We have seen that the coordinator log contains the identities of the participants, but not the information necessary to commit or abort each participant at will. This information is distributed among the many participants in the form of *participant logs*, each of which contains the information necessary to commit or abort the corresponding participant according to the coordinator's command. The log corresponding to a given participant is created as part of the prepare actions of each participant and is stored in stable storage at the participant's site. Once the participant log is created and a low-level name assigned to it, the coordinator is informed of the name of the participant log. Once all participants have created associated participant logs, the coordinator log will be updated to include the names of all of these participant logs. Thus we see that the coordinator log actually contains centralized information as to the identities of all of its participants as well as references to the distributed information which is necessary to commit or abort the transaction

* See [Lindsay 79] for alternative forms of the two-phase commit protocol.

as illustrated in Figure 6.

In the sections that follow, we describe in detail how and when each of these logs is created and the interaction between them throughout the two-phase commit protocol. We first describe the operation of the protocol under favorable conditions and then extend our discussion to include handling of failure conditions. Some optimizations to our two-phase commit algorithm are possible and we discuss these in the next chapter.

7.3 Normal Case Behavior

The two-phase commit protocol is initiated when the top-level process of the top-level transaction completes successfully and after all of the transaction's participant files have been commit-lock-updated. Since the two-phase commit protocol is invoked only during completion of a top-level transaction, we no longer need to distinguish between transactions and top-level transactions. For simplicity, we will use the term transaction and top-level transaction synonymously throughout the discussion of the two-phase commit protocol.

The two-phase commit protocol is driven by the messages sent by the coordinator to all of the sites which store files which must participate in the two-phase commit of this top-level transaction. Up until the commit point, the functions of the coordinator are performed by the process running on behalf of the application program which invoked the top-level transaction. When the commit point is reached, the transaction is added to a completion queue and control is returned to the application process which invoked the transaction as described in Chapter 4. The second phase of the two-phase commit protocol is performed by a daemon which services the completion queue. In the discussion that follows, we use the term coordinator to

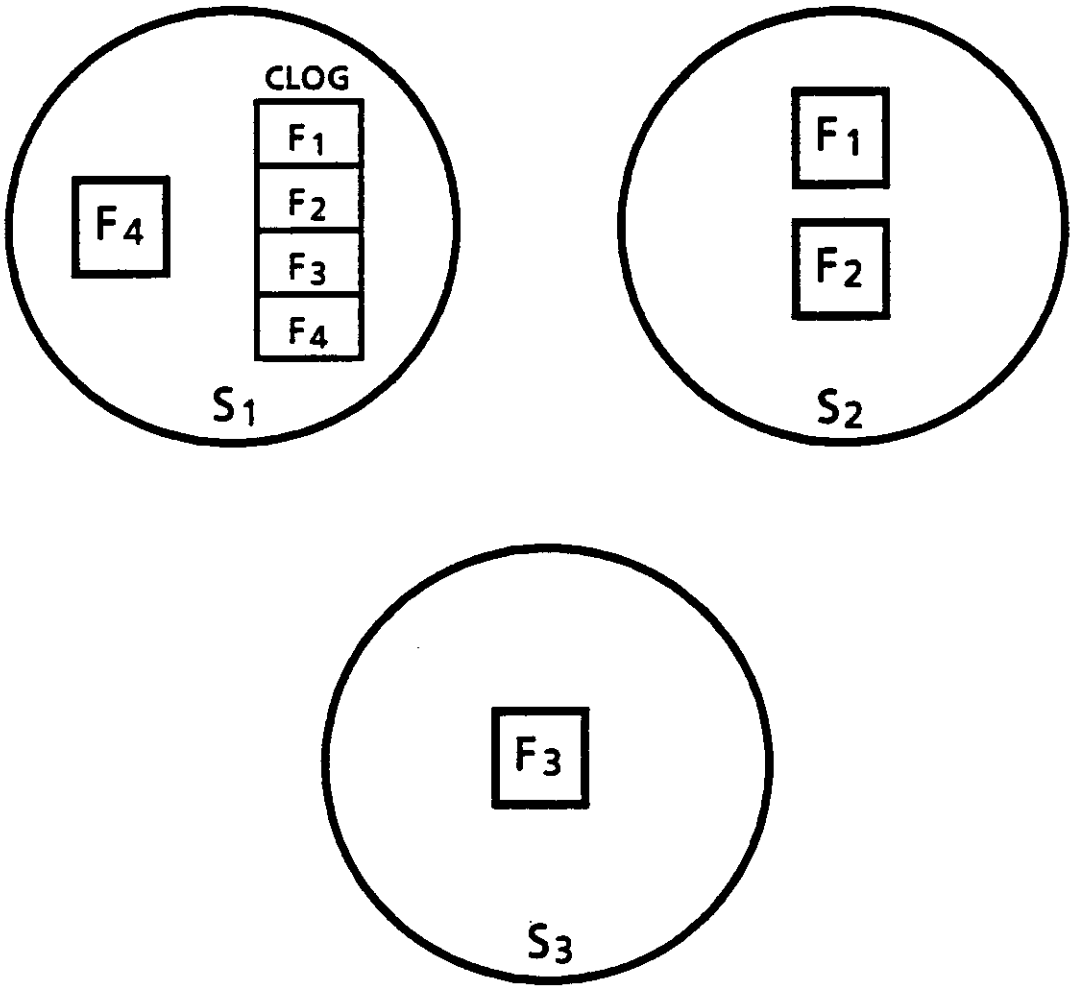

Figure 6: Two-Level Log Structure

refer to the process which is performing the first phase of the protocol as well as the daemon which later services the completion queue.

Before beginning the prepare phase of the two-phase commit protocol, a coordinator for the two-phase commit of this transaction must be selected. In our scheme, the coordinator is simply the site where the two-phase commit protocol is invoked, i.e., the home site of the top-level transaction. Designating a site as coordinator consists of atomically writing the coordinator log to stable storage. Once this is done, this site assumes coordinator responsibilities for the duration of the two-phase commit of this transaction. The coordinator log is essentially a stable version of the transaction structure of previous chapters and it includes the Tid of the transaction, the transaction's status (at this point UNDEFINED), and the participant file list. Once the log is written to stable storage, we begin the prepare phase of the protocol.

During the prepare phase, we must cause each participant to take such actions as are necessary to guarantee that the participant can later commit or abort as commanded by the coordinator. If the participant is local, we take the prepare actions locally. Otherwise, a message is sent to the participant site causing the prepare actions to be taken at the remote site. What comprises the prepare actions is determined by the single-file commit mechanism provided by the system. Since the single-file commit mechanism employed in Locus uses *intentions lists* [Lampson 79], the prepare actions consist of stably storing the intentions list and returning the name of this stable intentions list to the coordinator. This stably-stored intentions list is the participant log and there is one such such log for each participant file. Recall that in the intentions list scheme, a file is represented by an array of pointers to physical data pages, called the file's *index*. When we wish to modify a logical page

of a file, we do not write over the old version of the page. Instead, we allocate a new physical page, copy the old page data into the new page, modify it, and record the address of the new page in the intentions list. The act of committing consists of atomically carrying out the intentions, i.e., atomically updating the file's index so that the new pages are referenced.*

When a local participant completes the prepare actions, we change its status to PREPARED and record the participant log name in the appropriate entry in the volatile participant file list. When a remote participant completes the prepare actions it sends a response to the coordinator, and we similarly update the appropriate entry in the participant file list. Once all entries in the participant file list have their status set to PREPARED, the transaction is ready to go on to the commit phase.

Here note that preparation of the participants is done in parallel to the degree that is feasible. We traverse the participant file list entry by entry. If a local participant is encountered the local prepare actions are taken, the participant file list is updated, and then we move on to the next entry. However, if a remote participant is encountered, we simply send a prepare message to the remote site and move on to the next participant entry. Upon reaching the end of the list, we return to the head of the list to traverse it once again. This time we check the status of each entry. If the status of an entry is PREPARED, we move on to the next entry. If the status is still UNDEFINED, we wait on this entry until its status becomes resolved using the sleep and wakeup mechanism described above.* If the status indicates that an error

* Note that our algorithm does not depend on the particular single-file commit mechanism provided by the system. If the system used a logging mechanism instead of intentions lists, the prepare actions would consist of making the appropriate log entries. Furthermore, the log associated with the participant file would act as the participant log.

* The topology change procedure which is part of Locus will notify us if we are waiting for a response from a site which is no longer in the current partition. This eliminates the need for timeouts in many parts of our algorithm.

occurred or that a participant wishes to unilaterally abort, we change the status in the transaction structure to ABORT and put this transaction on the completion queue. The completion queue is a list of transactions which are waiting to be completed, either committed or aborted based on the status in the transaction structure. Transactions are added to this queue as soon as their final disposition is determined. The queue is serviced by one or more daemons, the number of which should be set to satisfy performance criteria. The details of the abort procedure will be described later.

If we reach the end of the participant file list, we know that all of the participants are prepared and the transaction can be committed. We update the status field in the transaction structure to COMMITTED and atomically write the information in the transaction structure--the Tid of the transaction, status of COMMITTED, and participant file list including the participant log names--to the coordinator log, overwriting the version written out prior to the prepare phase. This atomic action is the *commit point* of the transaction, i.e., the atomic action which irrevocably commits the transaction. Once this point is reached, we are assured that the effects of this transaction will eventually occur irreversibly at all sites involved in the transaction. As in the abort case, the transaction is added to the completion queue where the commit phase of the protocol will be performed when a daemon becomes available.

The daemons servicing the completion queue remove the first item from the queue and check its status to determine what actions must be taken. If the status indicates that this transaction is to be committed, the daemon traverses the participant file list and does the following for each entry. If the participant is local, the commit actions are performed locally. Otherwise, a message is sent to the partici-

pant site causing the commit actions to be taken at the remote site. The commit actions consist of retrieving the intentions list from the version stored in the participant log* and carrying out those intentions, i.e., changing the file's index so that the new data pages belong to the active version of the file. Once the participant is committed, we delete the participant log and release the locks held on the participant file. When a local participant completes the commit actions, its status is changed to COMMITTED in the appropriate entry in the participant file list. When a remote participant completes the commit actions it sends a response to the coordinator which causes the status to be updated in the appropriate participant file list entry. Once all entries in the participant file list have their status set to COMMITTED, the transaction is complete and all that remains is the cleanup phase. The commit phase is carried out in parallel among the participants as much as is feasible as described in the discussion of the prepare phase. The cleanup phase consists of deleting the coordinator log and releasing volatile data structures used for this transaction, e.g., the transaction structure and participant file list.

Before considering the abort procedure, we look at some of the possible causes of transaction abort at this stage. An abort will be caused if any of the participants cannot be reached during the prepare phase (e.g., due to a site failure or network partition) or if a participant decides to unilaterally abort the transaction during the prepare phase for local autonomy reasons. Recall that being prepared to commit means that any resources held on behalf of the transaction must remain sequestered until the outcome of the transaction is determined and the participant site is notified to commit or abort. This compromises local autonomy, but is a necessary constraint if the coordinator is to assume that all participants will remain able to commit or

* In the normal case, when the site storing the participant does not crash or become partitioned from the coordinator between the prepare and commit phases, the intentions list will be in volatile storage and will not have to be retrieved from the stable participant log.

abort at its command. Hence, if a participant wishes to abort at any time, it must be done before it responds affirmatively in the prepare phase. If such a condition occurs during the prepare phase, the status of the participant wishing to abort the transaction is set to ABORTED in the appropriate entry in the participant file list. When the coordinator discovers any participant whose status is set to ABORTED it immediately initiates the abort procedure.

The abort procedure is much the same as the commit procedure, with minor differences. To abort a transaction, the status in the transaction structure is set to ABORTED and the information in the transaction structure--the Tid of the transaction, status of ABORTED, and participant file list with the statuses and participant log names--is atomically written to the coordinator log, writing over the version stably stored before starting the prepare phase.* Again, the participant file list is traversed with abort actions for local participants taking place immediately, and abort messages being sent to remote participants to invoke the abort actions remotely. The abort actions consist of deleting the participant log for the participant (if one exists) and releasing the t-lock for the participant file. Once every participant has taken the abort actions, the transaction is complete and we go on to the cleanup phase just as in the commit case.

7.4 Handling Partitions and Outages

Through careful examination of the effects of crashes and network partitions once the two-phase commit protocol is invoked, we attempt to convince the reader

* Any participant which does not have a associated participant log will have a null entry in this field in the coordinator log. The participant log name is simply stored for efficiency reasons and is not strictly necessary. As we will see when failure conditions are discussed in the next section, each site maintains a table mapping participants to their associated participant logs. Storing the participant log names in the abort case allows us to delete the participant logs directly thus bypassing this table lookup.

that the protocol is robust. As will become evident throughout the discussion, the algorithm relies not only on the transaction management code, but also on the topology change procedure and consistency check programs which are regularly run during system restart.

At this point we recall the policy we wish to enforce. If a failure occurs before the commit point, we must abort the transaction leaving the files it involved in the state they were in before the transaction began. If a failure occurs after the commit point, the transaction must be completed so that all participant files reflect the effects of the transaction. Furthermore, we wish to ensure that the effects of any transaction whose final disposition is unknown are not seen outside the scope of the transaction. To commit a transaction, we must do the following:

1) For each participant:
   a) Replace the old version of the file by the new version as instructed by the intentions list.
   b) Delete the participant log which was created for this participant.
   c) Release the lock held on the participant file.
2) Delete the transaction coordinator log.
3) Release any volatile structures used in processing the transaction.

To abort a transaction, we perform the same actions with the exception of replacing the old version of the file with the new (step 1a). To see that our protocols operate correctly in the face of failures, we will describe how these the actions are eventually accomplished either when the failure occurs, at partition merge time, or site restart.

Recall from our discussion of operation under normal conditions that we begin the two-phase commit protocol by atomically writing the coordinator log to stable storage at the site which invokes the two-phase commit protocol. This site then acts as coordinator for the remainder of the protocol. If a failure occurs in writing out the log, (e.g., the coordinating site has run out of space and hence cannot write out the log), the two-phase commit software returns to its caller with an error code that indicates that the transaction cannot be committed. As a result, the caller will take the actions necessary to abort the transaction, i.e., it will abort-lock-update the participant files and inform the transaction's caller that the transaction has been aborted. Since writing the coordinator log is an atomic action, a crash which occurs before this act is completed is indistinguishable from a crash which occurs before the two-phase commit protocol is invoked. The handling of such failures was described in our discussion of transaction abort in Chapter 4.

Once a coordinator log has been successfully written to stable storage at a site, that site is designated coordinator for this transaction and will be responsible for seeing that it is completed--either committed or aborted. The coordinator then enters the prepare phase as described in the preceding section. From this point, it is useful to consider the effects of failures and network partitions first from the participant's viewpoint, and then from the coordinator's perspective.

7.4.1 Participant Viewpoint

Each file which is a participant in a transaction has a t-lock structure in volatile memory. A flag in this data structure indicates whether or not the participant is

prepared and if so, it contains the name of the corresponding participant log.* Whenever the topology change procedure discovers that a site has left the current partition, it examines all t-locks to find those files participating in some operation involving the site which has just disappeared. These t-locks are then removed from memory in order to prevent tying up volatile storage for an indefinite duration and to allow continued operation during network partitions.** In particular, participants which have become separated from their coordinators as a result of the topology change are located and their associated t-locks removed.

If a participant has not yet been prepared when it becomes partitioned from its coordinator, the orphan removal algorithm discussed in Chapter 5 will release the lock on the file and free the t-lock structure. The algorithm becomes somewhat more complex if the participant has already been prepared when it loses communication with the coordinator. There are two problems which must be handled in this case and both arise because the t-lock structure associated with the participant file must be removed from volatile memory as a result of the topology change. This data structure contains two vital pieces of information which are not stored in the stable version of the file for performance reasons: the prepared flag and the name of the participant log. We first consider the ramifications of losing the knowledge that this file is prepared. Whenever a partition occurs, locks on files held by sites no longer in the current partition are released. This allows users in the current partition to access the file independent of what is going on in other partitions thus increasing availability. Of course, copies of files which are replicated in various parti-

* This flag and the name of the participant log are not stored in the stable version of the file because this would add two I/O operations to the prepare actions of every participant and this was deemed to be a significant performance penalty.

** However, access to files which are participants in an incomplete transaction will be prohibited until the transaction completes. In our case, we will prohibit access to the file until the transaction is either committed or aborted.

tions may be in conflict when the partitions merge, but this is handled by the recovery software provided in Locus. For files which are not participants in a transaction, such a policy poses few problems. However, prepared participant files must remain locked until the fate of the transaction can be determined. Hence we need some means by which to indicate that this file (or at least this copy of a replicated file) is involved in an as yet unresolved transaction and hence cannot be accessed by others until the transaction completes. The topology change procedure assumes this task by setting the appropriate flag in the stable version of the file when it removes a prepared participant's t-lock from volatile storage. Subsequent requests to open this copy of the file can thus be denied whenever this flag is set.

The other important bit of information which will be lost when the volatile data structure is removed from memory is the name of the file's participant log. Thus we need some mechanism to allow us to find the participant log when we regain communication with the coordinator. The solution is to keep a volatile table of participant logs whose entries are of the form: <FileName, ParticipantLogName>. Whenever a prepared participant is removed from volatile storage by the topology change procedure, an entry is added to this table. In this manner, the participant log for this file may be found later.

To complete our discussion of the protocol from the participant's vantage point, we must describe what happens to participant files when the site upon which they reside crashes and subsequently comes back up and wishes to merge with the rest of the network. When a site crashes, all of the information contained in its volatile memory is lost. In particular, the t-locks containing the prepared flag and the name of the participant log, if one exists, are lost. We must show that this information can be reconstructed from information in stable storage. In fact, this in-

formation can be regained through inspection of the file storage system. Whenever a site crashes, various inconsistencies in the file storage system may result. These are routinely handled by running consistency check programs which look for and attempt to repair any problems which exist. These programs are run before the site allows any access to files stored on that site and before the site attempts to rejoin the network. In order to gather the information needed by the transaction mechanism, some additional checks are added. In particular, the check programs find all participant logs and rebuild the volatile table of participant logs. They then set the appropriate flag in the stable version of all those files for whom a participant log has been found. Thus by the time a site rejoins the network, its participant log table has been rebuilt and the stable version of all prepared participants has been flagged. In this manner, all participant logs may be located and all subsequent attempts to access prepared participants can be denied until such time as these participants become resolved.

Thus we have seen that the appropriate actions are taken at the participant site in all possible cases. We have seen that all unprepared participants can and will be aborted if they become separated from their coordinator as a result of a network topology change. We have also seen that participants which are unprepared at the time of a site failure are implicitly aborted since all volatile information is lost as a result of site failure. Furthermore, we have seen that all prepared participants remain inaccessible and prepared to commit or abort until they receive orders from the coordinator in both the network partition case and the site failure case. It remains to be shown that our algorithm is robust from the coordinator's vantage point.

7.4.2 Coordinator Viewpoint

We will begin by considering how the coordinator handles partitions which separate it from one or more of its participants. Again, we will see the interaction between the transaction mechanism and the topology change procedure. Partitions during the prepare phase which separate participants from the coordinator will cause the transaction to be aborted, while those during the commit phase will cause processing on that transaction to be abandoned until a partition merge. From there we will go on to describe how the coordinator recovers from a crash and continues its transaction processing.

Recall that during the prepare phase the coordinator sends prepare messages to all remote participants and then awaits their responses. Clearly some mechanism is needed to prevent the coordinator from waiting forever for a response from a participant which has been made unavailable by a failure or network partition. Again we rely on the topology change procedure to solve this problem. If any of the remote participants becomes unavailable before it has responded to the prepare message, the topology change procedure will set that participant's status in the participant file list to SITEDOWN and notify the coordinator. When the coordinator discovers any participant whose status is set to SITEDOWN, it sets the status in the transaction structure to ABORTED, and immediately initiates the abort procedure. As we saw earlier, this procedure atomically updates the coordinator log and then adds this transaction to the completion queue where it will be aborted in the second phase of the protocol. If the transaction completes the prepare phase successfully, its status is set to COMMITTED, its coordinator log is atomically updated, and it is added to the completion queue to be committed.

The second phase of the protocol works on the transactions which are on the completion queue. At this point we will discuss management of this queue. The actual queue is an incore cache consisting of a finite number of transaction structures. A transaction structure (plus the associated participant file list) is just a volatile version of the coordinator log. The cache is used for performance reasons. If there is room in the cache when a transaction completes phase one, it can be put directly into the cache. In this case, we do not have to read the coordinator log before performing the second phase. If the cache becomes full, we do not add any further transactions to it, but instead allow it to empty. Whenever the cache becomes empty, we try to refill it by going through the coordinator logs looking for all those transactions which can be completed. Those which are eligible are those with their status set either to COMMITTED or ABORTED and which are not flagged as impossible to complete due to the current network topology. Those transactions which are still in the prepare phase will have their status set to UNDEFINED and hence will not be put on the queue.

Now we consider what happens if a partition separates the coordinator from a remote participant during the second phase of the protocol. As in the prepare phase, the coordinator sends commit (or abort) messages to remote participants and then waits for their responses. Here again we rely on the topology change procedure to guarantee that the coordinator is informed if a participant which has not yet responded is no longer available. If such a partition occurs the transaction cannot be completed until the lost participant again becomes available. Work on this transaction is suspended, its transaction structure is flagged to indicate that it cannot be completed until a network merge, and the transaction is removed from the cache. Whenever a site rejoins the network, the topology change procedure unmarks the appropriate transaction structures making them eligible to be put into the cache when

it is refilled.

Finally, we must consider the effects of coordinator failure. Whenever a site crashes and comes back up, it must complete all of the transactions for which it is responsible, i.e., all of those transactions whose coordinator logs reside at this site. This requires little more than refilling the cache and initiating the daemons which service the completion queue. However, before allowing any transaction activity to continue, we must examine the status field of each log. Transactions which have completed the prepare phase, i.e., those whose final disposition was determined before the crash, have a status of COMMITTED or ABORTED and need not be changed. But, any transaction which was still in the prepare phase when the coordinator crashed will have a status of UNDEFINED. The protocol requires that these transactions be aborted. When such a transaction is encountered, its status is changed to ABORT. We cannot simply leave their status UNDEFINED because as soon as we allow new transactions to be initiated, coordinator logs with UNDEFINED status will be created for these new transactions. In this case, we would not be able to distinguish the transactions which are active and in the prepare phase from those which must be aborted as a result of an earlier crash. Once these have been attended to, the cache can be refilled, the daemons invoked, and new transaction activity can be allowed.

CHAPTER 8

OPTIMIZATIONS TO TWO-PHASE COMMIT

In the previous chapter, we gave a detailed description of the two-phase commit protocol employed in Locus nested transactions. The presentation of the algorithm was organized so as to facilitate the reader's understanding. However, several optimizations to the mechanism as described are possible. We present them in this chapter.

8.1 Batching Messages

One of the most trivial and obvious optimizations to the algorithm is to batch messages for participants which are stored at the same site and to batch all responses from these co-resident participants to the coordinator. Recall that in our description of the algorithm, we sent a message to each participant even if several participants were located at a single site and that each of these participants subsequently sent a response to the coordinator. Considerable message overhead may be saved by grouping together the messages to and from participants which reside at the same site. Sending one long message rather than several short ones saves both message processing and message space. We save processing at both the sending and receiving sites since there are fewer messages to send and receive. We also save message space because one long message requires only one header and trailer whereas each of several small messages requires its own header and trailer information. Furthermore, the data in each message at any phase of the protocol is the same to all participants, e.g., the prepare message tells each site to prepare and hence we

need include this only once if one large message is sent. Note however that responses from participants at a given site to the coordinator may differ. For example, one participant may choose to respond negatively to a prepare message while all others at a site respond positively. Hence the response of each participant will need to be included in the batched response message.

Extending this batching scheme, we can reduce the four message protocol as described, to a protocol requiring three plus epsilon messages. Recall that the final message of the two-phase commit protocol is simply used in order for the coordinator to know when all participants have completed so that it can delete the coordinator log. If, instead of sending these responses promptly when a participant completes, we batch the responses of several participants even across transactions and send these responses only periodically, the protocol basically becomes a three message protocol. The basis for this optimization comes from the SafeTalk protocol reported in [Menasce 80].

8.2 Removing Completed Participants

Another optimization concerns the handling of completed participants. Recall that in our original discussion, we kept a volatile version of the coordinator log during the entire protocol for transactions currently being processed and did not update the non-volatile version after the commit point until we deleted the log. Thus each time a coordinator log was processed after an exceptional condition occurred, we sent messages to all of the participants and awaited their responses. If instead, we write out the coordinator log (with completed participants removed) whenever a group of responses is received, we can avoid sending redundant messages to sites whose participants have already completed. In this manner, we do not require all participants of the transaction to be present in order to complete the transaction. Note that the

scheme previously described did impose this restriction. Thus we see that once the commit point is reached, we can complete a transaction even if all of the original participants never come together again. We require only that the coordinator talk to each of these participants at some time subsequent to the commit point.

This optimization further expedites the protocol since we can flush the coordinator log from core immediately after sending the command messages (commit or abort) in the second phase. Then, when responses are received, we will read in the coordinator log, make the necessary updates and write it out again. This may increase the parallelism between the processing of transactions which have the same coordinator.

8.3 Replication of Coordinator Logs

Another optimization to the simple scheme described in this report is replication of coordinator logs. The goal of this addition is to free resources as soon as possible in extraordinary situations. As we noted in an earlier section, participants in a transaction wait for the coordinator to command them to commit or abort. Once a participant has positively responded to a prepare message, that participant can take no further action until instructed to do so by the coordinator. By replicating coordinator logs, we believe that significant benefit can be gained especially in an environment where partitions are expected to be frequent.

Having decided that replication of coordinator logs is desirable, we are faced with an important question. When is it useful to replicate the coordinator log? We will argue that substantial benefit may be derived from replicating the coordinator log once the fate of the transaction is known but not before this time. To see this recall that the motivation for selecting a single centralized coordinator is to provide

an atomic action which can act as the commit point of the transaction. If we were to replicate the coordinator log before this single actions occurs, we would require a scheme for coordinating the replicated copies of the log, i.e., we would need a two-phase commit strategy (or some similar scheme) to ensure that all copies of the coordinator log record the same value of the transaction status. If such a scheme is not employed, a carefully timed partition could cause there to be two partitions which have copies of a coordinator log with different statuses. For example, in one partition the status may be COMMIT while in the other it is UNDEFINED. In this case, participants which are in the partition containing the log whose status is COMMIT could be resolved and their resources released. However, participants in the other partition can take no action and thus this situation is no different from the case where there is only one copy of the coordinator log and it is in another partition. Thus nothing is gained in this case by replicating the log before the commit point. Furthermore, we can no longer blithely change the status of a transaction from UNDEFINED to ABORT when an exceptional condition occurs. (Recall that in the current scheme, upon site restart any transaction whose status is UNDEFINED is simply changed to ABORT.) We need some other way of determining when a transaction whose status is UNDEFINED may be changed to ABORT.

In sum, we cannot change the status of a transaction unless we guarantee that all copies will have unconflicting statuses, i.e., we must avoid a situation where some copies of the log think that the transaction is to be aborted while other copies think that it is to be committed. Thus we see that replicating the coordinator log before the final disposition of the transaction has been determined provides no advantage and can lead to inconsistent results unless some additional measures are taken. For example, Reed proposes a scheme which essentially replicates the coordinator before the fate of the transaction is determined and then employs a voting scheme to decide

whether to commit or abort the transaction [Reed 78]. Completion does not require a unanimous vote, so in any partition containing a majority of the copies of the coordinator, the transaction can be resolved. However, since only one partition can have a majority of the copies, for our purposes, this scheme is little better than the scheme which employs one coordinator. In all fairness to Reed, he proposed this scheme as a solution to a different problem, namely the problem of permanent failure of the coordinator during the two-phase commit. However, while narrowing the window of failure, Reed's proposed algorithm does not eliminate it, as Moss points out in [Moss 81].

If, however, we maintain only one copy of the coordinator log until the fate of the transaction is known and then add replicated copies*, we can realize significant gain in exceptional conditions as will be described below. Using this scheme, we incur little additional overhead to manage the replicated copies of the coordinator log and we make considerable progress towards our goal to release sequestered resources quickly. Note that this does not solve the problem Reed was attempting to solve. Permanent loss of the coordinator between the time the first phase of the protocol is begun and before the commit point is reached causes some participants to remain locked forever. However, through the use of stable storage and the pack porting scheme described earlier, we hope to have made the probability of such an event so small as to be inconsequential in practice.

* Several implementation strategies are possible. A single copy may exist until final outcome of the transaction is known at which time additional copies are created. Alternatively, all copies may be created simultaneously with one flagged as the valid copy until the commit point when this version is propagated to all other copies marking them valid as well. Regardless of the chosen implementation details, it remains true that there can be only one valid copy of the log until the fate of the transaction is determined.

We now have the problem that if there are copies of the coordinator log in different partitions, they can be in different states when the partitions merge due to the removal of completed participants. However, the semantics of coordinator log files is sufficiently simple as to make automatic reconciliation possible. Given several conflicting copies of a coordinator log, they may be resolved using the following simple rules. The number of participant entries in a coordinator log monotonically decreases until no participant entries are left at which time the coordinator log file may be deleted. Once a participant has been removed in any copy of the coordinator log, that participant has been resolved and can be removed in all copies. Hence, when merging several copies of the coordinator log we obtain the set of remaining participants by forming the intersection of the participants in all copies of the log. Given that the recovery system of Locus can detect conflicts between multiple copies of a file and that the semantics of the files which act as coordinator logs is well understood, we can easily build system software which will automatically resolve multiple copies of coordinator logs at partition merge time.

So we see that replication of coordinator logs can be very beneficial in an environment where partitioning is frequent. In particular, more participants can be resolved and hence resources can be freed more quickly if replicated copies of the coordinator log are dispersed throughout the different partitions.

8.4 Participant Querying

In addition to coordinator log replication, we will also add a mechanism by which participants can query the coordinator log file. By employing such a mechanism, we hope to avoid some futile attempts to complete transactions whose unresolved participants are unavailable. Recall that whenever a site restart or partition merge occurs, each site marks all coordinator logs as active and attempts to pro-

cess each log in hopes of completing the corresponding transaction. In an environment where partitioning is frequent and highly variable, there may be many attempts to process a given log before actually succeeding. We intend the querying scheme to act as an advisory mechanism to coordinators. In this way, a coordinating site can queue up logs to be processed in response to an inquiry from one of the participants interested in this log. This may eliminate the processing of logs which have no unresolved participants in the current partition.

In order to make clear the problem (or inefficiency) we are trying to alleviate we present the following example. Suppose a given network consists of three sites which are denoted A, B, and C and that site C becomes partitioned from A and B. Further suppose that there are several transactions at site A and B which are enqueued at their respective sites and which have participants at site C. Whenever a log at site A or B is processed and it is discovered that some of the participants are unreachable, that log will be marked as inactive. Participants which are reachable will be resolved and removed in the log whenever responses indicating that they have completed are received by the coordinator. Now assume that A and B become partitioned from one another and then merge and that all the while site C remains unavailable. Recall that our algorithm will mark all coordinator logs at A and B as active because of the partition merge even though many of them cannot be resolved because they require communication with participants at site C. Clearly if C is down for a long period of time and many short-lived partitions occur between A and B in the interim, the amount of work wasted in vain attempts to resolve these logs can be considerable. Moreover, logs which could be resolved will wait behind the others in the queue of logs to be processed.

To solve the problems illustrated above, we propose to implement a querying scheme which allows participants to play a more active role in the scheduling of logs to be processed. Recall that upon site restart and partition merge, we locate all prepared participants and build a table mapping participants to their participant logs. These participants then wait for commands from their coordinators. Some simple additions to the existing scheme will allow us to add a querying mechanism. One addition is to include the coordinator log name in the participant log for each participant. Then for each prepared participant, we send a query to any site which has a copy of the coordinator log.* To determine which site, if any, in the current partition stores a copy of this log file, we simply use the parts of the file open mechanism already existing in Locus. Recall that in Locus, one opens a file simply by specifying its application level name. One does not have to specify the site at which the file resides and the system will locate and open any copy of the file which is available in the current partition. If a copy of the coordinator log is reachable, the participant site will send a query to the coordinating site. This query will cause the coordinating site to mark this coordinator log as active (if it is currently inactive). This log will then be queued and processed at the coordinator's convenience as in the scheme we have previously described. Note that this could cause multiple copies of a given coordinator log to be processed. However, this is not a problem since all copies agree on the status of the transaction and we have taken care to guarantee that the actions taken by a participant when it is commanded to commit and abort are idempotent.

We propose that queries not explicitly be acknowledged by the coordinator so as to save on message traffic and protocol overhead. If this advice is followed, sites

* Throughout this discussion we intend coordinator or coordinator log to mean valid copy of the coordinator log.

should periodically initiate querying to protect against the deleterious effects of lost queries. Of course, this need only be done if it has been a long time since the last site restart or partition merge which themselves cause querying to be initiated.

Through use of this query mechanism we hope to eliminate much wasted effort in environments where partitioning is frequent and in which both long and short-lived partitions are expected to occur.

CHAPTER 9

RELATED WORK

In this chapter we present a review of work related to ours in an effort to put our work in proper perspective. In particular, we review the result of recent research in the area of transaction management including work on single-level as well as nested transactions. We also review the Argus language which incorporates nested transactions, and the work in remote procedure calls done by Bruce Nelson.

In reviewing other work, we will attempt to compare this work to ours in various respects including functionality, reliability, replication, concurrency control, network transparency, performance, and degree of implementation.

9.1 Single-level Transactions

The transaction concept has long been a common notion and we do not attempt to trace its origins. Gray states that the transaction concept derives from contract law [Gray 81]. The use of transactions in computer systems, for example in electronic funds transfer and airline reservations systems, has become commonplace. Here we will review just some of the work that has been done in the area of single-level transactions. We have attempted to select the work which we believe is representative and we discuss some of the systems which employ transactions as a means to guarantee system consistency and/or reliability. For a good survey of the transaction concept and many frequently used techniques, see [Gray 78] [Gray 81] [Lindsay 79] [Lampson 79].

9.1.1 System R

System R is a relational data management system which employs single-level transactions to allow applications to commit, abort, or partially undo their effects. Its implementation consists of two layers of software which lie above the operating system. The internal layer provides, among other data management features, transaction management and recovery. For simplicity, one may assume that application programs consist of conventional programs which invoke a sequence of calls to this internal layer.

A transaction is defined as a application specified sequence of actions on the objects implemented in System R's internal layer. Examples of such actions include: insert record, fetch record, and create file. An application declares the start of a transaction by issuing a BEGIN action. Thereafter, all data manipulation actions requested by that application are within the scope of that transaction until the application issues a COMMIT or ABORT action. Transactions may also be aborted by the system, for example in cases of authorization violation, deadlock, or system crash. System R also defines the additional notion of *transaction save point*, which is a fire-wall where transaction undo may stop. Application programs declare a save point by issuing a SAVE action. If an exceptional condition occurs during the execution of a transaction, e.g., deadlock or resource limitations, it may be sufficient to back up the transaction to such an intermediate save point rather than undoing all the work of the transaction.

The internal layer of System R is responsible for running concurrent transactions and for assuring that each transaction sees a consistent view of the database. Each transaction is a unit of recovery and System R will recover the data to its most recent consistent state in the event of failure or user request to cancel the transac-

tion. Recovery is accomplished using an undo-redo log technique as described in [Gray 78] and [Gray 79]. All System R data is stored in files and System R defines two types of files: *shadowed* and *non-shadowed*. No automatic recovery is provided for non-shadowed files and these are updated directly. Users are responsible for making redundant copies of such files if they desire such functionality. By contrast, System R maintains two versions of shadowed files: a *shadow version* and a *current version*. In this case, operations affect only the current version of a file. The shadow version is altered only by file save and restore commands. The act of committing thus becomes the act of SAVing the current version of a file as the shadow version. Alternatively, the current version of a file can be RESTOREd to the shadow version thus undoing all updates since the last SAVE or COMMIT of this file. The current and shadow versions are implemented using an intentions list technique, just as in our scheme. The shadow version of a file survives system restart, but the current version does not. At system restart, all shadowed files are reset to their shadow versions. Then the log is used to remove the effects of aborted transactions and to restore the effects of committed transactions. The shadow and current versions are used to recover individual files and the log mechanism is used to coordinate the effects of multi-file transactions.

The transaction and recovery mechanisms of System R are in some ways similar to our implementation. As previously noted, System R employs an intentions list scheme similar to ours for its shadowed files. Recall that Locus employs an intentions list scheme for single-file commits and our scheme makes use of this mechanism by storing the intentions list as a participant log. However, note that our scheme is not tied to the use of intentions lists and could easily be modified to work properly with whatever single-file commit mechanism were made available by the operating system. Also note that System R employs a log for coordinating multi-file transac-

tions and for object state restoration. Recall that we also employ a log to perform our two-phase commit. However there are some differences. System R stores in its single log enough information to undo or redo every transaction which has ever taken place on a given site. In our case, a coordinator log only exists for the duration of the two-phase commit protocol, i.e., from the time the two-phase commit is invoked until the transaction is completed. Thus System R must provide schemes to decide when to move parts of the log to tape archives, since they cannot possibly be stored entirely on direct access devices. Of course, we do not require such mechanisms. However, System R does gain additional functionality by keeping this log, namely that recovery from media failure is possible for logged files. However, part of regular Locus system maintenance is a daily dump routine which dumps files to tape archives. Thus we can always restore files to a version not more than 24 hours out of date. Furthermore, we employ one coordinator log per transaction as opposed to a single log for all transactions at a given site. This will allow our coordinator logs to be replicated and/or to migrate on a per-transaction basis and, as we saw in Chapter 8, future plans include extending our current implementation to include such flexibility.

The transaction scheme employed in System R is also similar to our implementation in its use of locking for synchronization. Both schemes lock objects for the duration of the transaction in which they are involved.

Of course, the transaction mechanism of System R is limited in functionality when compared to our scheme since it is implemented as part of a data management system and not a general-purpose programming environment. Thus the operations which applications may include in their transactions are limited to database operations. Most importantly, the transaction mechanism of System R does not support

nested transactions. As we have argued in previous sections, we believe that these limitations are too severe for a general-purpose programming environment. In addition, System R transactions have no parallelism within a transaction, for example if multiple nodes of a network execute a single transaction, at any given time only one node is executing that transaction. By contrast, we have seen that our scheme allows parallel execution among processes which make up a transaction.

System R's transaction facility provides some functionality which we do not provide, and that is the transaction save point mechanism which allows transactions to replace the shadow version with the current version. We could easily add a similar mechanism to our scheme, in particular we could add a mechanism which would perform intermediate commits of the top-level transaction. However, the since System R keeps a log as well as the shadow version, their transactions can be rolled back beyond transaction save points using the log. If we add intermediate commits to our scheme, we could not roll back past intermediate commits without keeping multiple "old" versions of the files.

System R's transaction management system as described here is fully implemented. Further development is ongoing including a distributed version of system R, known as R*. For more information on System R and System R* the reader is referred to [Gray 79] and [Lindsay 79].

9.1.2 Tandem ENCOMPASS

ENCOMPASS* is a distributed data management system developed by Tandem Computers Incorporated. ENCOMPASS is part of the Tandem system which provides a high degree of reliability by employing redundancy at both hardware and

* ENCOMPASS is a Trademark of Tandem Computers Incorporated.

software levels. At least two paths connect any two components in the system. Each Tandem node consists of from 2 to 16 processors each of which runs a copy of the operating system. The message-based operating system manages all system resources in a decentralized fashion. The message system makes the physical distribution of hardware components transparent to processes. The message-based structure of the Tandem operating system allows easy extension to a local network of Tandem nodes.

The transaction management facility of ENCOMPASS, known as the Transaction Monitoring Facility (TMF), implements single-level transactions. To initiate a transaction, the user issues a BEGIN-TRANSACTION call. This marks the beginning of a sequence of operations which are to be treated as a single transaction. The allowable operations are invocations of application "server" programs which access and update database files. The structure of each application server program is simple and single-threaded: 1) read the transaction request, 2) perform the database function requested, 3) reply. The network location of the application server process and/or all or part of the database is transparent to the user and may reside on remote network nodes. Execution of BEGIN-TRANSACTION causes a unique transaction identifier, *transid*, to be generated. From this transid, it is possible to determine the identity of the processor in which the transaction was initiated, and the identity of the network node of which this processor is a component, called the *home node* for the transaction. All messages sent from this processor on behalf of this transaction, e.g., disk I/O or record locking requests, have this transid appended to them. When the sequence of operations which make up this transaction are complete, the user issues the END-TRANSACTION command to cause the transaction's database updates to become permanent.

TMF uses locking for synchronization. All data items required by the transaction must be locked before they are read or updated and must remain so until the end of the transaction.

The Tandem system provides the notion of a *process-pair*. A process-pair consists of two cooperating processes which run in two different processors. One of these processes is designated the *primary* and the other the *backup*. The primary handles all requests to access the resource it manages, for example an I/O device, and the backup functions as a standby in case of failure of the primary. The primary process sends to the backup process checkpoints which ensure that the backup process has all the information that it requires to assume management of the resource and complete any operation initiated by the primary in the event of failure of the primary. A DISCPROCESS, implemented as an I/O process-pair, controls all accesses to the disk volume it manages and protects the integrity of the files resident on this volume. This scheme represents a type of stable storage. TMF also maintains distributed audit trails of logical database record updates. The DISCPROCESS which manages a disk volume automatically causes *before-images* and *after-images* of database updates by application processes to be written to an audit trail. This audit trail will remind readers of the log kept by System R and is used to recover from total node failures.

TMF uses a two-phase commit protocol similar to those we have already seen. At each network node, there is a process-pair known as the Transaction Monitor Process (TMP). The messages required to perform the well-known distributed two-phase commit protocol are sent from TMP to TMP. Each node which participated in the transaction sends transaction state change messages to the TMP of all nodes for which it was the direct source of transmission of messages relating to this tran-

saction. Upon receipt of such a message, the TMP broadcasts the message to all processors within that node. For phase one to complete successfully, each node to which the home node directly transmitted the transid must be accessible and reply affirmatively. Before replying, the TMP writes the transaction's audit records to disk and guarantees that all nodes to which the transid was further transmitted have done likewise. The remainder of the protocol is similar to that of other systems.

This form of the two-phase commit protocol offers an interesting contrast to the scheme we have implemented. Recall that we kept a single list of participants at the home site of the top-level transaction rather than having each node keep a list of all participants at that node and a list of all nodes on which it invoked transaction related operations. We believe the two schemes to be equivalent in terms of record keeping. However, note that our scheme requires us to inform the home site when a new participant is added to a transaction. This involves a round-trip message delay when this occurs. The Tandem scheme does not incur this delay when new participants are added. However, our scheme allows us to send the two-phase commit messages directly to all participants. Thus we do not have to wait for these messages to trickle down and back up a tree of participants. Depending on the characteristics of the communications links involved, the time spent processing these messages, and the depth of this tree, the performance of the two-phase commit protocol will vary and may degrade considerably using a scheme such as Tandem's.

TMF compares favorably with our scheme in the area of network transparency. Both schemes allow transactions to access resources at remote sites without having to specify the site upon which those resources are stored. Note, however, that this property of network transparency is not a feature of either transaction mechanism, but rather the operating systems upon which these transactions have been built.

It has been our experience that transparency provided at a low-level greatly simplifies programming distributed tasks at higher levels.

As is true of the transaction management facility of System R, TMF is implemented as part of a data management system and hence is not a general-purpose tool. Furthermore, TMF does not support nested transactions. TMF is a working system and is commercially available as part of the ENCOMPASS data management system. Readers interested in a more thorough discussion of TMF are referred to [Borr 81]. Those who desire more information regarding the Tandem Operating System are encouraged to read [Bartlett 78] and [Bartlett 81].

9.1.3 Distributed INGRES

Distributed INGRES is a distributed version of the relational database management system, INGRES. The concurrency control, crash recovery, and multiple copy update algorithms are based on a *primary site* model, much like that presented in [Alsberg 76]. Each object possesses a known primary site to which all updates in the network for that object are first directed. Distinct objects may have distinct primary sites. In this system, an object is a subset of the rows of a relation, called a *fragment*. Each relation is partitioned into fragments, each with a primary site and some number of multiple copies.

A transaction in the distributed INGRES environment may consist of a single query language command which is either a command to retrieve or update data. While the authors claim that the more general case, where a transaction consists of an arbitrary collection of such commands, requires no additional algorithmic complexity, the more general case is not now implemented and future plans do not include its implementation. Crash recovery is decomposed into local crash recovery at

each node with some additional processing. The single-site crash recovery software handles only transactions which consists of a single command. However, if single-site recovery were modified to handle the general case, the authors believe that their algorithms would work for the general case.

Concurrency control is achieved through locking. Each site in the network has a local concurrency controller (CC) which handles control for all local transactions. The lock tables that each CC creates and uses are local to its site and are not present at any other site in the network. Thus, locking is handled in a distributed fashion. Deadlocks may occur. Deadlock detection and resolution is handled by a single, universally known machine called the SNOOP. When a local CC notices that transaction X holds a lock for which transaction Y waits, this information is sent to the SNOOP. This machine then constructs a global "wait for" graph from which it can detect deadlocks.

To control multiple copy consistency and handle crash recovery, each node maintains an *up-list*. This is a list of sites that the given node believes to be operational. This up-list is analogous to the site table discussed in our algorithms. Furthermore, each site maintains the identity of the current SNOOP. If, at any time, the current SNOOP is not in the up-list, there is a globally agreed upon procedure for selecting a new SNOOP. The location of all copies of an object may be determined by examination of some system catalogues, and there is a known linear ordering of all these copies. The primary copy is defined to be that copy which is the lowest in the ordering among those sites in the up-list. When the network is partitioned, a primary copy of an object is defined to exist only if a majority of all copies are at sites in the up-list.

The correct operation of distributed INGRES requires some assumptions concerning the underlying communications subsystem. These are as follows. First, every message sent is reliably received if the recipient is in the up-list and the sender does not crash while sending the message. In the case that the recipient is not in the up-list, the sender can queue messages for later delivery. Lastly, messages arrive in the order in which they originated from the sender. Note that our transaction mechanism requires none of these assumptions to operate correctly. Furthermore, the management of the queues of messages for inaccessible sites seems to us burdensome if sites are down for long periods of time.

A transaction is initiated from a user process at some site. This causes a "master" collection of INGRES processes to be invoked at that site. This master creates "slave" INGRES processes at other sites where processing on behalf of this transaction is to take place and ensures that each slave knows the identity of all other slaves. The master then examines the up-list and calculates the primary site for each object involved in the transaction. If the primary site of any object is inaccessible, the transaction fails. Otherwise, the master supervises the distributed transaction making sure that each slave knows the updates that the slave must make locally. The master then waits for a ready message from each slave. When such a message is received from every slave, the master sets its commit flag. This is the commit point of the transaction. Next, the master sends commit messages to all of the slaves and waits for a done message from each of the slaves. It then notifies the application process that the transaction has completed.

Each slave assists the master in processing the transaction. After a slave has completed its portion of the transaction, it notifies the master that it is ready and then waits for further instructions. If the master sends a commit message, the slave

commits its updates and responds to the master with a done message. If the master sends a reset message, the slave runs a local recovery algorithm.

Above we have briefly reviewed just a portion of the algorithm. There are recovery algorithms which govern the behavior of the master and its slaves if failures occur while a transaction is in progress. The algorithm we have briefly described is part of the performance algorithm. Two versions of the algorithms exist. One promises good performance but can lead to data integrity problems. The other guarantees that no data integrity problems will occur if a certain number of the sites are up, but the performance degradation suffered by this algorithm is severe. The latter algorithm requires a commit to update all copies of an object instead of just the primary copy. See [Stonebraker 79] for a more detailed discussion of the two sets of algorithms.

This brings out one of the fundamental differences between distributed INGRES and Locus. Namely, the builders of Locus decided that, for the sake of reliability, operation must be allowed to continue during network partitions. If the resources for an operation are available in a given partition, that operation must be allowed to proceed even if some of the required resources are replicated in other partitions. Although this policy can easily lead to consistency conflicts at partition merge time, we believe this to be a reasonable policy for the reasons we argued in Chapter 2. The builders of INGRES do not take this philosophy and thus have attempted to avoid consistency problems by using a primary copy strategy which does not allow update in partitions which do not possess the primary copy. The price paid by the INGRES system is reduced availability during network partitions and poor response time if the reliable algorithms are used.

The transaction mechanism of Distributed INGRES, like that of System R, is limited in functionality when compared to our scheme. Again, this is largely due to the fact that it is implemented as part of a data management system. Furthermore, the operations which applications may include in their transactions are limited to database operations and in fact are limited to single query language commands. Of course, the transaction mechanism of distributed INGRES does not allow nesting of transactions. However, we have seen that the transaction mechanism of distributed INGRES does exist in an environment where replication is included in the basic system and hence provides algorithms to manage replicated copies.

One feature provided by distributed INGRES which is not yet implemented in our system is that of promoting a slave to master so that the transaction can complete in some cases if its master becomes unavailable. Their algorithm works in the following manner. If a slave notices that its master has become unavailable, the slave examines its up-list to determine whether or not all other slaves are accessible. If so, and this slave is the lowest numbered in a predetermined order among the slaves, this slave takes on master responsibilities. Otherwise, it waits for another slave to take over. The new master then queries all slaves to see whether or not any had previously received a commit command. If so, this transaction may be committed by the new master, otherwise it is aborted by the new master. In our scheme, functionality of this type will be gained when we implement replicated coordinator logs as discussed in Chapter 8.

9.1.4 Sirius-Delta

Delta is a real-time distributed transaction processing system built within the framework of Project Sirius at INRIA [LeLann 81].

A transaction is a set of elementary actions -- such as READ, WRITE, DELETE, CREATE -- which manipulate data objects grouped into files. Two types of executive processes are used in transaction management: ***producers*** are in charge of controlling the execution of transactions and ***consumers*** are in charge of performing transactions on objects. As in our implementation and that of distributed INGRES, Delta employs a scheme to keep track of which nodes are currently accessible. Each producer is assigned a unique identifier that defines its order in the set of producers which make up a ***virtual ring.*** Every producer is required to periodically exchange messages with its successor to determine whether or not its successor is still available. If several messages go unacknowledged, the producer attempts to contact the next producer in the sequence. This procedure is repeated until a positive response is received or the producer discovers that it is the only active producer. There is also a virtual ring insertion protocol which allow producers to rejoin the network. The set of producers currently in the ring are the accessible nodes.

The virtual ring is also used as a distributed concurrency control mechanism in the following way. A special message called a ***control token*** circulates around the virtual ring. This token carries with it a ***sequencer*** that at any time has an integer value called a *ticket.* A producer obtains a ticket by accessing the sequencer to find its current value and then incrementing the sequencer. Protocols exist to guarantee that a token is circulating at all times and that there is only one such token and that the sequencer maintains the required properties. The sequencer is used to timestamp transactions which are submitted to producers by applications. Each transaction obtains a unique ticket and all actions invoked on behalf of a given transaction carry the ticket value allocated to that transaction. This mechanism provides all the information necessary for a consumer to correctly schedule competing actions. When two transactions have conflicts at several consumers, all such conflicts will be

resolved identically as follows. Whenever an action possessing a lower-numbered ticket is received, a consumer decides immediately either to roll back or to abort any actions possessing a higher-numbered ticket.

The use of timestamps as a concurrency control mechanism is quite different from the locking scheme that our system employs. Our scheme requires us to keep state information in the form of lock tables, while the timestamp scheme requires only timestamps to be kept. However, the timestamp mechanism causes some number of unnecessary rollback and aborts. Lelann argues that in real-time transaction processing systems, transactions do not represent large amounts of processing. Hence rollbacks and aborts should not be considered costly when compared with the price of exchanging state information among nodes in a distributed environment. Since our system is a general-purpose programming environment, we cannot make assumptions about the amount of transaction processing requested by users nor the duration of those transactions. In such an environment, we believe locking to be the superior choice for concurrency control. Furthermore, our implementation is part of the Locus operating system which uses locking as its synchronization mechanism and hence for compatibility reasons locking seemed an obvious choice for our mechanism.

Transactions which write, create, or delete objects update copies of these objects rather than the actual objects. A form of the two-phase commit protocol is used to cause the modifications to be made to the actual data objects. Objects are locked during the two-phase commit protocol. The two-phase commit protocol employed in Delta is similar to the version employed in our scheme with the following modification. Included in the prepare to commit message which the producer sends to all participating consumers is the list of all consumers involved in the execution of the transaction being committed. This enables any consumer noticing that the pro-

ducer has failed, to attempt to contact all other consumers involved in the transaction to determine what the outcome of the transaction should be. If any consumer has aborted, the transaction will be aborted. Similarly, if any consumer has committed, the transaction will be committed. If all are accessible and all have responded positively to the prepare to commit message, the transaction will be committed. The only case in which no action may be taken is the case where none of the accessible consumers has either committed or aborted and the producer as well as some of the consumers are inaccessible. In this case, the transaction cannot be completed until the failed consumers are once again available.

In order to recover from crashes, each node keeps a log containing all information necessary to survive failures. Whenever a write action is issued, the system employs a two-phase commit protocol to atomically update all copies of an object and to make appropriate entries in the log. The log entry includes the list of all consumers involved in this write operation. In this manner, writes are atomic even if some elements crash. Furthermore, consumers periodically cause checkpoints to be written to the log. Thus upon restart, a consumer begins by returning to the state recorded as the most recent checkpoint in the log and then processing all write operations which occurred after this checkpoint as recorded in the log. Upon encountering a write action which was prepared but not committed, this consumer consults other consumers belonging to the associated list in order to determine whether to commit or abort this action. Once this is completed for all write actions in the log, the consumer is once again accessible. Since it is not necessary to wait for a failed producer to complete a transaction, producers may be considered memoryless systems. Only consumers are provided with stable storage. Thus the failure and recovery of producers does not affect data consistency.

This approach to crash recovery is an interesting alternative to our method. Recall that we keep a list of all participants and the state of the transaction in a single centralized coordinator log. By contrast, Delta keeps no centralized information about the state of the transaction (once a producer has failed) and keeps a list of all participants (consumers) with each participant. In the normal case, our approaches are equivalent. It is in cases of failure that our schemes differ. We identify some of the tradeoffs briefly. Delta uses a truly decentralized approach. Some may argue that this provides a greater degree of local autonomy than an approach such as ours since participants may be able to free locked objects sooner. However, if we replicate coordinator logs, a participant need only communicate with one of the many sites storing the replicated coordinator log to determine the outcome of the transaction. In addition, if we so desire, we can approximate the Delta scheme by replicating the coordinator log at each participant site. Furthermore, in some cases, we believe that the Delta scheme may even cause resources to be held longer. To see this, recall that in the case that the producer has failed, no accessible consumer has either committed or aborted, and at least one consumer is inaccessible, Delta requires all consumers to be in communication again before the transaction can complete. Our scheme, with the optimizations described in Chapter 8, can complete a transaction once all participants have responded successfully to the prepare message even if they are never again in the same partition. Furthermore, we believe our approach to be simpler, since the fully decentralized algorithm requires much more sophistication from its consumers (participants) while requiring the same complexity of the producer (coordinator) since it is responsible for committing the transaction in the normal case. Thus, in choosing between the two schemes, a system designer must decide what is appropriate for the particular environment in question.

Another significant difference between our systems is the notion of network partitioning. While Locus deals with this issue in a general manner, Delta assumes that all nodes are either in the virtual ring or inaccessible. While this is a suitable model for a single local area network, this model does not gracefully extend to more sophisticated topologies such as multiple local networks connected by gateways.

While Delta does not support nested transactions, it is a transaction tool for a general-purpose programming environment, unlike the transaction mechanisms we have reviewed up to this point. In addition, Delta provides replication of data objects containing vital information so that the system can survive crashes of storage elements. However, the details of this replicated copy consistency were not available for our review. We have discussed this subject at length in our review of distributed INGRES.

9.2 Nested Transactions

In recent years, the concept of nested transactions has been developed as a general-purpose programming tool for an unreliable environment. The earliest paper describing something resembling nested transactions seems to be [Davies 73]. His term is *spheres of control.* However, this work does not represent a design for a nested transaction mechanism, but rather attempts to define some of the general properties required of any such scheme.

In this section, we review two substantial previous designs for nested transactions. We first consider a design for nested transactions using locking for synchronization proposed by Eliot Moss in his doctoral thesis at MIT [Moss 81]. Next, we present a summary of a nested transaction scheme which uses timestamps for synchronization. This scheme came about as part of a general object naming scheme

proposed by David Reed in his doctoral thesis at MIT [Reed 78]. We compare and contrast their designs with ours.

Also worthy of note is the Eden File System (EFS) which is currently being designed at the University of Washington. EFS employs an object model approach in the design of a transaction-based file system which will be a part of the Eden distributed system. The Eden distributed system, which is currently under development, provides a high degree of network transparency among its many features. In addition, EFS will provide support for nested transactions, file replication, and multiple versions. At the current time, few details about EFS are available. In particular, very little information regarding the issues of transaction management for nested transactions, concurrency control, and recovery management is available. However, some portions of EFS are currently being simulated and we believe that much useful knowledge will come out of this effort. For a brief summary of the Eden Transaction-Based File System, readers are referred to [Jessop 82].

9.2.1 Reed

Reed has designed a mechanism for implementing nested transactions using a timestamp technique for synchronization and recovery. This mechanism is part of the NAMOS (Naming as Applied to Modular Object Synchronization) system which is described in detail in [Reed 78]. The essence of Reed's approach is as follows. Each modifiable object in the system is regarded as a sequence of unchangeable *versions*. Each version is the state of the object after an update is made to the object. The sequence of versions is referred to as the *object history*. In NAMOS, each version has a two-component name consisting of the object name and a *pseudo-time*. The pseudo-time is the name of the system state to which the version belongs. Thus, we can envision each object history as a set of values each of which is valid

over a region of pseudo-time. For example, the object O and its value history might be:

O: <v0,[t0,t1]>, <v1,[t1,t2]>, <v2,[t2,t*]>

meaning that O had the value v0 from pseudo-time t0 to t1, then at time t1, O was assigned the value v1, then at time t2, O was assigned the value v2 and this is the current value. If a program now reads O at time t3, it will see the value of the object at that time. If t3 > t2, then the value v2 will be made valid for the period [t2,t3]. If a program assigns value v3 to O at time t3, one of the following occurs. If t3 > t2, the object history of O becomes:

O: <v0,[t0,t1]>, <v1,[t1,t2]>, <v2,[t2,t3]>, <v3,[t3,*]>

However, if t3 <= t2, then the program is aborted because it is attempting to rewrite history.

In this context, synchronization is a mechanism for naming versions to be read and for defining where in the object history a new version resulting from some update belongs. Synchronization of access to multiple objects is achieved by giving programs control over the pseudo-time in which an access is made. To accomplish this, Reed introduces the notion of a *pseudo-temporal environment* which is simply a region of pseudo-time. The pseudo-temporal environment allows a program to ensure that the objects it accesses change only as a result of actions performed by this program. That is, the pseudo-temporal environment provides a means to reserve a range of pseudo-times for the exclusive use of the program making the reservation. We can think of pseudo-temporal environments in the following way. There is a root pseudo-temporal environment which is the set of all pseudo-times. This root may be subdivided into subranges that begin and end in pseudo-times that are

system-wide consistent states. One of these subranges that corresponds to the execution of a sequential program is further broken up into subranges that begin and end in pseudo-times that correspond to the states in between execution of operations that are separate modules of this program. All of these ranges are pseudo-temporal environments.

Reed observes that pseudo-time must be correlated with real time. In order to generate pseudo-times that are ordered in correspondence with real time, one must have at each node a way of creating a pseudo-time value that exceeds all previous values created at that node. The pseudo-time value created must also exceed all values created at other nodes at significantly earlier times. Reed discusses how the correlation with real time may be achieved using approximately synchronized clocks [Reed 78].

Reed further introduces the concept of a *possibility*. A possibility is a group of tentative versions created by updates that can simultaneously be installed as real versions. A possibility contains a boolean value. This value is set to true when the associated set of updates complete. The possibility mechanism also includes a timeout after which the boolean value cannot possibly be set to true. This allows the system to bound the time that a group of updates may be in progress. Thus there are three important states of a possibility. Before its value is determined, a possibility is said to be in the *wait* state. An attempt to access the value at this time will be forced to wait until the value is set or until the timeout goes off. If the group of changes have been completed, the possibility enters the *complete* state. If an error occurs or the timeout goes off while the possibility is still in the wait state, the possibility enters the *abort* state. All of the changes made under the control of a particular possibility form an atomic action, since any computations attempting to

access the objects which are part of this possibility will either see all of the changes (if the possibility eventually completes) or see none of the changes (if the possibility eventually aborts.) Reed extends this concept to include dependent possibilities whose completion also depends on the completion of their parent possibility.

The concepts of pseudo-temporal environments and possibilities can be used to implement nested transactions in the following way. Any computation that manipulates shared objects can be executed as a transaction by: 1) executing it in a unique pseudo-temporal environment; and 2) making all of the updates conditional on a possibility that is completed if no errors occur that prevent the computation from finishing. The first of these insures that the transaction reads and writes consistent data and that the only changes made to the state of the system during the range of pseudo-times composing the transaction are those that the transaction itself initiates. The second guarantees that no other computations see any of the intermediate states that the transaction creates as it proceeds.

Within a transaction, accesses to objects must occur in a certain order, namely the order of the statements making up the transaction. To guarantee this, the rules for getting a pseudo-time for a pseudo-temporal environment are as follows. If a pseudo-time is needed for a read operation, use the pseudo-time of the last write operation. If a pseudo-time is needed for a write operation, it must be strictly later than any pseudo-time used for a previous read or write operation. This ensures that changes made by earlier statements are seen by later statements. Reed also provides an operation for constructing parallel streams of pseudo-time. See [Reed 78] for details.

A subtransaction is executed in a pseudo-temporal environment which is a unique subregion of the transaction which invoked it. If a transaction invokes several subtransactions, the ordering of the subregions must correspond to the ordering of the statements in the transaction code which invoked the subtransactions. In addition, the pseudo-temporal environments of the subtransactions must be ordered with respect to the pseudo-times used for object accesses in the invoking transaction. More specifically, all accesses to objects within a subtransaction must deal with versions later in pseudo-time than those accessed by statements preceding the invocation of the subtransaction. In addition, all accesses following completion of the subtransaction must deal with versions later in pseudo-time than any of the versions accessed by the subtransaction. To execute subtransactions in parallel, a subregion is further divided into non-overlapping regions.

Associated with each subtransaction is a stably stored *commit record* which contains the state of the subtransaction: waiting, complete, or aborted, as well as a reference to the transaction which invoked this subtransaction. Each time an object is updated, a new stable version of the object, called a *token* is created without destroying the old one version. This token contains a reference to the updating transaction's commit record. While the commit record corresponding to a token is in the waiting state, only the transaction which created this token may access it. Once a transaction completes, all tokens created by this transaction become versions, thus making valid all updates performed on behalf of this transaction. This is accomplished by setting the state of the transaction's commit record to complete and broadcasting the transaction outcome. To change a commit record, the system must locally lock that commit record. The object history of each object involved then incorporates the updates of this completed transaction. Then all descendants of the transaction which invoked the completed transaction may see the updates caused by

the completed transaction. If a transaction aborts, the tokens created by this transaction are discarded, thus invalidating all of its updates. This is accomplished by setting the state of the transaction's commit record to aborted and broadcasting this information.

Querying is used to check the state of a commit record not stored locally. When a transaction requests access to an object in a pseudo-time for which a token currently exists, the site storing the object will query the commit record to determine the state of the transaction which created this token. Suppose the commit record is in the wait state. If the requester is part of the transaction which created the token, the token may be accessed by the requester. However, if the requester is not part of this transaction, the requester must wait until the commit record leaves the wait state. The mechanism is actually more complicated than presented here when dependent possibilities are involved. For more details on determining the right to access a token, especially in the case of dependent possibilities, see [Reed 78]. If the commit record is aborted, the token will be discarded and the previous version may be accessed. If the commit record is in the complete state, then the token will eventually become a version and the requester will be able to access this new version.

Reed uses a timestamp scheme as opposed to our locking mechanism. Hence Reed's scheme allows applications the full power of time-domain addressing. For example, if old versions are retained, Reed's scheme can easily produce answers to queries about the state of the system at some previous time, e.g., the inventory as of the end of last year. However, keeping all the known history of an object would require the system to provide an ever increasing amount of memory. Thus the system can maintain only a subset of the known history of each object. A mechanism must be devised for choosing the subset of versions to be maintained. Reed notes that a

system which does not retain old versions requires that a very large read-only transaction, e.g., one involving many sites and consequently much delay, lock out all updates to any objects being read by this transaction for a relatively long period of time. He points out that systems based upon locking suffer from this problem as well. His use of timestamps for ensuring correct synchronization makes it possible to eliminate this form of lockout if sufficient old history of objects likely to be involved in read-only transactions is retained.

However, as we pointed out in our discussion of Delta, the use of timestamps for synchronization produces some of its own problems. We previously noted that use of a timestamp mechanism for concurrency control causes some number of unnecessary rollback and aborts. In addition to this drawback, reads of an object often cause extending the time period for which an particular value of an object is valid, as in our example above. Thus reads must update the object history, which may increase I/O activity. Furthermore, a long-running transaction which performs many updates may encounter starvation, never finishing because it is repeatedly aborted by new transactions accessing some of the same objects. Another problem is dynamic deadlock, where several transactions cause each other to be mutually aborted and, upon each restart, the timing of the transactions causes mutual aborting to recur. Reed proposes a reservation scheme which can be used to reduce the likelihood of starvation and dynamic deadlock where needed, at the cost of requiring that a transaction "reserve" its resources in advance. This scheme is useful if the transaction can predict the resources it will need. To us, such "reservations" sound very much like locks.

There are other complications in Reed's scheme. For example, commit records are very special entities and the protocols for deleting them and reusing

them, although not presented here, are quite complex. Secondly, for performance reasons, Reed suggests encached commit records, again requiring mechanisms to keep encached versions consistent with the real version.

In summary, several problems remain to be worked out before an implementation of such a scheme for nested transactions can be implemented. However, many of these issues may be settled when actual implementation is attempted. Reed and his colleagues are currently developing such a system [Svobodova 80].

9.2.2 Moss

Our work is heavily influenced by the model of nested transactions developed by Eliot Moss and the Distributed Systems Group of MIT's Laboratory for Computer Science. Moss [Moss 81] develops the nested transaction concept in detail, including the locking rules which are necessary to ensure serializability. A transaction manager design which implements nested transactions and a deadlock detection algorithm to guarantee transaction progress are presented. In this section, we review Moss's design for nested transactions and compare and contrast it with ours.

Each site in the network runs a *transaction manager*, and all the transaction managers follow the same algorithm. The transaction manager handles transaction-related processing requests at a site. It is informed when transactions are created, when they are to be committed or aborted, and so on. The transaction managers communicate privately among themselves according to the protocols which we will explain shortly. In the following discussion, when we speak of the transaction manager of some transaction, we mean the transaction manager of the transaction's home site.

Transactions are assumed to run at a single site. That is, a transaction is not permitted to be distributed over several sites. In order to accomplish work at another site, a separate subtransaction must be invoked at the other site. Moss imposes this restriction for simplicity. This does not compromise the functionality of the system. By imposing this restriction the bookkeeping for each transaction is localized to the transaction's home site. In contrast, our model does not impose this restriction, in keeping with the principle of network transparency. We are able to restrict most of the transaction bookkeeping to the transaction home site in any case. The remote site informs the home site whenever the bookkeeping information must be updated, e.g., if the remote process opens a file, forks, terminates, or starts a subtransaction. Our topology change procedure simply aborts a transaction having member processes which cannot communicate.

Moss also assumes that a transaction may modify objects only on the site on which the transaction is executing. Although a transaction may modify objects only at the transaction's home site, it may modify objects at another site by invoking a subtransaction at the other site. We do not impose this restriction, since it violates object location transparency.

When a new transaction is encountered by a transaction manager, an entry for the transaction is created. Entries for all the transaction's superiors are also created if they do not already exist at that site. When a transaction is first started, it is in the *running* state. When the transaction is done performing its work and wishes to commit, it enters the *finished* state. Before a transaction may commit, all the transaction's children must be *resolved*, i.e., either committed or aborted. However, a transaction may abort at any time. Once the transaction's children have all been resolved, the transaction manager moves the transaction from *finished* to *com-*

mitted.

Although transactions cannot modify objects at other sites, they may exist at other sites in the sense that they may retain locks to objects at other sites. Each transaction manager maintains object locking and state restoration information in volatile storage. When a site's transaction manager is informed of the commit or abort of a transaction, it updates locking and state restoration information associated with that transaction at the site.

A transaction has *visited* a site if the site is the home of the transaction or one of the transaction's committed inferiors. The transaction manager keeps a list of sites visited by each of its transactions. When a transaction commits, the list of visited sites specifies which other transaction managers must be informed of the commitment, via a *commit notice*. The parent's transaction manager must also be notified. The list of visited sites is sent to the parent's transaction manager, which adds the received list to its list. Thus once all the children of a transaction have committed, the transaction's transaction manager has an accurate list of sites visited by the transaction. The visited sites must be informed when a transaction commits, so that they may update locking information. Note that in Moss's model, the only locking information which must be updated when a transaction commits is that which is located at the home sites of the committing transaction's descendants. Once a transaction has committed and the necessary adjustments to other tables have been completed, the entry for the transaction may be discarded at that site, since the necessary information has been passed to the parent.

A transaction may abort at any time without waiting for its children. When a transaction aborts, all the inferior's transaction managers are informed. This is accomplished by sending an *abort notice* to all children, who in turn send abort notices

to their children, and so on. Each transaction manager receiving the abort notice appropriately updates locking and state restoration information associated with that transaction.

Lost messages are handled in Moss's design as follows. If a transaction manager does not hear about a transaction it is interested in for some length of time, it queries the transaction's transaction manager. Such queries are repeated until a response is received, or is no longer required. Out-of-order, duplicate, and delayed messages are also handled in the scheme.

The first kind of querying is called *parent* querying. This type of querying is performed when a transaction that wishes to commit queries its unresolved remote children. This will handle the loss of commit notices from a child to parent as well as a loss of the original message to cause the child transaction to be started.

The second type of querying is called *participant* querying. This kind of querying handles the loss of a commit notice from a transaction to its visited sites. The visited sites query the transaction's transaction manager. This kind of querying is also performed when resources held by a remote transaction are requested by another transaction.

If a site on which a transaction has been run crashes before the top-level transaction commits, work performed by the transaction will be lost since the updates to objects are stored in volatile memory. Thus a transaction which has committed may be aborted by a crash. To handle this, the top-level commit is accomplished using a two-phase commit protocol. In the prepare phase, each participant, or committed inferior of the top-level transaction, checks that crashes have not destroyed transactions thought to have committed. The top-level transaction may complete only if

every transaction which is in the transaction's list of committed inferiors is still committed at each participant, that is, the transaction has not aborted as a result of a crash. In addition, the participant resolves any unresolved inferiors of the transaction using a list of committed inferiors passed to it in the prepare message. A transaction which is in the list and unresolved at the participant should be committed; all other inferiors which are unresolved at the participant should be aborted. Also, there may be sites which ran unsuccessful inferiors, which are not included in the top-level transaction's list of committed inferiors. These sites will eventually discover that the transactions were unsuccessful through participant querying.

Since Moss's algorithms require all of a transaction's children to be resolved before the transaction may commit, a transaction must wait until all children sites become accessible before it may commit. In order to solve this problem, each transaction wishing to invoke a remote subtransaction first invokes a local subtransaction. This local child subtransaction then invokes the desired remote transaction. Thus each remote subtransaction is really a grandchild of the invoking transaction. Suppose the remote site crashes. In this event, the local child subtransaction may be aborted, effectively aborting the remote subtransaction as well. By aborting the local child, the invoker may commit if it wishes. Now, since the remote subtransaction has crashed, inferiors of the remote subtransaction will not necessarily receive the abort notice from the aborted intervening transaction. Thus some inferiors of the remote subtransaction are orphans--they continue to run and hold resources. Participant querying will eventually resolve orphans; however, in Moss's algorithm, participants only query the most deeply nested transactions. Since a site which is down cannot respond to queries, orphans cannot be resolved until the crashed site comes up. Thus alternate transactions which wish to access resources at orphan sites before the site comes up will not succeed.

Since Moss seems to assume that both site and communications failures are of short duration, this is not a problem.* We, on the other hand, have taken a different approach in our design. In particular, we acknowledge the possibility of network partitions: the situation in which a set of sites will be isolated from other sets of sites for an unknown length of time. We assume the existence of a *partition manager*, which assures us that within a partition all sites can communicate with one another. We have used the concept of *topology change* to drive our recovery algorithms, which assure prompt orphan removal. In the event that a transaction requests a resource before the orphan removal has completed, querying is employed.

Moss also designed a deadlock detection algorithm which finds all deadlock cycles, and does not report many "phantom" deadlocks. The algorithm is initiated whenever a transaction begins to wait for a lock which is held in a conflicting mode by another transaction. The algorithm follows edges of the transaction/resource waits-for graph in real time, looking for cycles. When a cycle is found, an appropriate transaction is aborted to break the deadlock.

We have not provided such a deadlock detection algorithm in our design. In our system, when a transaction requests a lock on a file, if the lock cannot be granted after trying a certain number of times, the transaction is refused the lock. The transaction has the option of trying to request the lock again, to accomplish the work in some other way, such as by requesting a lock on an alternate resource, or the transaction may simply abort. Thus instead of incorporating a deadlock prevention or detection algorithm into our transaction mechanism, we rely on the application programmer to resolve deadlocks. Transactions can avoid deadlocks simply by aborting if a lock cannot be obtained after trying a certain number of times. Simi-

* In any case, this problem may be fixed by querying superiors as well.

larly, a subtransaction which aborts as a result of not being able to obtain a lock, may be reinvoked by its caller. In this case, the calling transaction should also abort after a certain number of retries. Thus our system makes no guarantee that progress will be made. Cyclic restart and starvation effects are possible, but may be avoidable through careful programming. Additional experience is needed in this area.

Unlike Moss, we have incorporated replication into our transaction mechanism because we believe it is essential for data replication to be managed by the system. Update of different copies of the same object is permitted during partitioning. For those data types which the system understands, automatic reconciliation is performed. Otherwise, the problem is reported to the application level. Although in this report we have not considered the problems of consistency of data objects in the event of updates in multiple partitions, this problem has been dealt with extensively in other work.

In order to gain confidence in his protocols, Moss performed a centralized simulation of his design. We have produced an actual distributed implementation which has been integrated into the Locus operating system, and have solved many implementation issues in the process. We are now well equipped to evaluate the utility of a nested transaction mechanism, since we have a functioning system.

9.3 Argus

In this report we have developed a design for a mechanism to provide for the construction of reliable programs in a distributed system, i.e., the nested transaction mechanism. We have provided the *relcall* system call which invokes a program as a subtransaction of its invoker, or as a top-level transaction if its invoker is not a transaction. We do not address the general issue of how a nested transaction mechanism

should be incorporated into a distributed programming language. Much work has been done in this area by Barbara Liskov's group at MIT in developing the Argus language [Liskov 82].

Argus is a programming language and system which is designed to organize and maintain distributed programs. The language is based on the CLU language [Liskov 81], and MIT's model of nested transactions [Moss 81]. In Argus, a program is constructed from a collection of modules called *guardians*. A guardian is an abstraction of a physical node of the underlying network. In constructing a distributed program, one thinks of the program as a collection of abstract nodes, each of which performs a meaningful task for its application.

A guardian is used to control access to one or more resources such as a hardware device or a database. The interface to a guardian is through a set of operations called *handlers*. Each guardian consists of a set of data objects and processes. Processes are spawned for each handler invocation, and perform background tasks as well. Every handler is executed as a subtransaction of its caller.

A guardian's state consists of both volatile and stable objects. The stable objects are written to stable storage only when the top-level transaction commits. The stable state of a guardian may be saved incrementally, that is, only those objects that were modified by the committing transaction need be written to stable storage; see for example [Mueller 81].

Processes within a guardian can share objects directly, however, sharing between guardians is not permitted--guardians may communicate only by calling handlers. In this way, each guardian retains control of its objects. Guardians are created dynamically by specifying the site at which a guardian is to be created. A

guardian may be used without knowing its location.

The objects which obey the indivisibility properties of atomic transactions are called *atomic objects*. Since Argus is based on CLU, a language which supports the use of abstract datatypes, Argus provides built-in atomic datatypes as well as user-defined atomic datatypes, through the *atomic cluster* mechanism.

Argus allows the arguments and results of a handler invocation to be of arbitrary type. Since different guardians may have different internal representations for the same abstract object, a scheme had to be devised for communicating abstract values in messages. The scheme is described in [Herlihy 82]. For each transmissible abstract type, one must define a canonical representation, or *external representation* which is used to communicate the values of objects of the type. The external representation may be any transmissible type. *Encode* and *decode* operations convert between the abstract type and the external representation. When it is desired to transmit a value, the system calls the encode operation to obtain an object of the type of the external representation. If the external representation is an abstract type, then the encode operation for *this* type is called, and this process is repeated until the original object is translated into a built-in type. The system knows how to communicate objects of built-in type. The reverse process occurs using the decode operation when a message containing the value is received.

So far, only a preliminary, centralized implementation of Argus has been produced.

9.4 Remote Procedure Call

Bruce Nelson defines remote procedure call (RPC) as "the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel" [Nelson 81]. Nelson attempts to provide both *syntactic* and *semantic* transparency for RPC. That is, the syntax for a remote procedure call must be the same as for a local procedure call. The semantics of a remote transactions should also be made the same as the semantics of a local procedure call. In order to do so, several possible semantics for procedure calls are examined.

In *exactly-once* semantics, a procedure is executed exactly once after which control is returned to the caller. This assumes in both the local and remote cases that the hardware is reliable. In *at-least-once* semantics for a remote transaction call, the procedure may be executed one or more times. This semantics results from a protocol in which the caller retransmits an invocation message periodically until the return message is received. In this case the called site may repeat the execution. When the caller receives a response, it is assured that the procedure was executed at least once. In *at-most-once* semantics, proposed in [Liskov 79], a remote procedure call is executed as a subtransaction of its invoker. The system guarantees that the call was acted upon exactly once (commit), or not all all (abort).

Last-one semantics is the semantics of a local procedure call with the possibility of crashes. That is, a procedure computation may be cut short. If the machine is rebooted and the program restarted, the procedure will be repeated. Since crashes can occur repeatedly, the call results are of the *last call* that executes. The results of intermediate executions--partial or total--are abandoned, although the side effects of intermediate calls can influence the last one. Thus last-one semantics is several par-

tial or complete executions of the procedure call followed by one complete execution of the call. In this case, the programmer must perform crash recovery for a remote procedure call, just as one must in a local procedure call if one is not using a transaction mechanism.

Nelson then devises suitable algorithms to achieve last-one semantics for remote procedure calls. This semantics is provided as follows. After a crash of the calling site, and before restarting any programs, all outstanding activity (called *orphans)* must cease. Several algorithms for dealing with orphans are developed. The most simple of these algorithms is extermination. In essence, when a crashed site S comes up, it contacts all the sites that it called, informing them to exterminate all its calls. Each of these sites in turn exterminates the children calls of the calls from S (by contacting other sites if necessary). Once having achieved last-one semantics, remote procedure calls have the same semantics as local procedure calls.

Liskov [Liskov 79] argues for at-most-once semantics for remote procedure calls in Argus, which addresses robust applications such as airline reservation systems, bank accounting systems, and so on. Nelson's work covers a broader spectrum of applications and argues for last-one semantics, on the grounds that the expense of transactions is not justified for many applications, e.g., mail systems, non-database type applications, and so forth. Additionally, he argues that remote procedures and atomicity are basically independent notions that require more investigation and experience before being tied together. We take Liskov's position, since it has been our experience that top-level transactions are at most twice as expensive as non-transaction programs. The expense is mostly in the top-level commit, which requires use of the two-phase commit protocol. Furthermore, as the transaction itself performs more operations, the expense of two-phase commit becomes less significant.

Executing a program as a subtransaction has even less overhead than running a non-transaction program, since the final committing of data to stable storage is deferred until top-level transaction commit. We feel that the additional expense of running a program as a transaction is well worth the reliability gained. See Appendix B for comparisons of execution times of top-level transactions, subtransactions, and non-transaction programs, as well as detailed measurements of the two-phase commit protocol.

CHAPTER 10

CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

In this chapter, we present some suggestions for further research and conclusions.

10.1 Checkpointing

As the duration of a transaction increases, it becomes more likely that the site on which the transaction is executing will crash during its lifetime and the transaction will not succeed. A mechanism to checkpoint the state of a transaction may be desirable in such cases. If the site executing the transaction crashes, the transaction may be restarted from the last checkpoint when the site comes up again.

Our nested transaction mechanism allows a subtransaction to fail, as a result of a partition or a crash, and for its caller to select an alternate transaction to accomplish a task similar to that of the failed subtransaction. If we were to checkpoint the subtransaction, and the invoker desired to employ the checkpointing, then the invoker would have to wait until the subtransaction's site recovered before it could complete. An alternative would be to send checkpoint information to another site, and to move the transaction to the other site in the event that the original site crashes or becomes inaccessible. However, the mechanisms required to accomplish such transaction migration can get quite complicated in a nested transaction mechanism. Thus a checkpointing mechanism may not fit well into our mechanism as it currently stands. However, simple checkpointing of top-level transactions could

be useful.

10.2 Abstract Data Types

In this report, the data objects of interest have been Locus files. The granularity of locking was also the file. In the future, the file system should be expanded to allow each data object to be of some *abstract type*, with associated access and partition recovery/merge operations. Each type should be able to have its own locking rules. Every access operation would obey the locking rules for the type of object being accessed. For example, the mailbox type would allow many transactions to concurrently append elements to a user's mailbox. In addition to there being predefined object types, abstract object types should be definable by the programmer.

Our design for nested transactions has assumed that the granularity of locking was the file. We saw that there was much information associated with each file locked by a transaction. This same amount of information must be kept with each lockable granule of an abstract data object. Schemes will have to be devised to reduce the overhead required for each component of an object.

10.3 Inter-Process Communication

It is not clear what the relationship is between transactions and inter-process communication. Suppose we wished to relax the restriction that running transactions may not communicate in any way, through shared objects or inter-process communication. If we allow such communication, indivisibility of the communicating transactions with respect to one another is clearly lost, since intermediate states of an object being modified by one transaction may be viewed by another transaction. Thus the communicating transactions cannot really be distinct transactions, since they disobey the properties of transactions.

Hence, when several transactions communicate, the transactions must somehow *merge* to form a single transaction. Now, all transactions participating in the merge commit or abort together. In a nested transaction scheme, transaction merge means that the tree structure of transaction invocation becomes an acyclic directed graph. Much additional mechanism may be required to manage transaction merge. The locking algorithms would have to be enhanced to deal with this more complex structure, e.g., the superiors of a transaction no longer form a list.

It is not clear whether there are any advantages to the concept of transaction merge. It could be argued that if two transactions wish to communicate, they should not be considered as two transactions. Rather, the two transactions should be combined into a single transaction (consisting of possibly many processes) and invoked that way. Further research is needed in this area.

10.4 Conclusions

Programming in a distributed environment is complicated by the additional failure modes of that environment. The transaction concept is an effective approach for coping with failures in a distributed system. The extension of transactions to nested transactions allows programmers to compose transaction programs freely, just as subroutines can be composed. Nested transactions also allow the programmer to perform two supposedly independent tasks simultaneously. By running the two tasks as subtransactions, the programmer is assured of serializable results.

A distributed implementation of nested transactions has been designed, implemented, and tested on the Locus operating system. Preliminary performance results indicate that transactions are not that expensive. The major expense lies in the two-phase commit protocol, used to commit a top-level transaction. The additional relia-

bility gained is well worth the added cost. Future work includes completing remote member process support,* taking extensive performance measurements, and incorporating appropriate optimizations.

The real benefits of nested transactions will probably not be known until they are widely used by programmers. Now that an operational environment for nested transactions exists, we look forward to considerable actual experience with real problems to evaluate their utility.

* The current implementation does not contain all of the mechanism necessary to support remote member processes.

REFERENCES

[Alsberg 76] Alsberg, P.A., and Day, J.D., "A Principle for Resilient Sharing of Distributed Resources", *Proceedings of 2nd International Conference on Software Engineering,* October 1976.

[Bartlett 81] Bartlett, J. P., "A Nonstop Kernel", *Proceedings of the Eighth Symposium on Operating Systems Principles,* Pacific Grove, California, December 1981.

[Bartlett 78] Bartlett, J. P., "A NonStop Operating System", *Eleventh Hawaii International Conference on System Sciences,* 1978.

[Bernstein 80] Bernstein, Philip A., and Goodman, Nathan, "Fundamental Algorithms for Concurrency Control in Distributed Database Systems", Computer Corporation of America Technical Report CCA-80-05, February 1982.

[Borr 81] Borr, A. J., "Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing", *Proceedings of 7th International Conference on Very Large Data Bases,* Cannes, France, September 9-11, 1981, pp. 155-165.

[Davies 73] Davies, C. T., "Recovery Semantics for a DB/DC System", *Proceedings ACM National Conference 28,* 1973, pp. 136-141.

[Edwards 82] Edwards, D. A., "Implementation of Replication in LOCUS: A Highly Reliable Distributed Operating System", Masters Thesis, Computer Science Department, University of California, Los Angeles, 1982.

[English 83] English, Robert M., and Popek, Gerald J., "Dynamic Reconfiguration of a Distributed Operating System", unpublished paper, Center for Experimental Computer Science, University of California, Los Angeles, January 1983.

[Eswaran 76] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM,* Vol. 19, No. 11, November 1976, pp. 624-633.

[Faissol 83] Faissol, Sergio Z., and Popek, Gerald J., "Partitioned Operation of Distributed Databases", submitted for publication.

[Faissol 81] Faissol, S., "Availability and Reliability Issues in Distributed Databases", Ph. D. Dissertation, Computer Science Department, University of California, Los Angeles, 1981.

[Goldberg 83] Goldberg, A., "Performance Evaluation of Local Area Distributed Systems", Master's Thesis (forthcoming), Computer Science Department, University of California, Los Angeles, 1983.

[Gray 81] Gray, J. N., "The Transaction Concept: Virtues and Limitations", *Proceedings of the Seventh International Conference on Very Large Data Bases,* Cannes, France, September 9-11,1981, pp. 144-154.

[Gray 79] Gray, J. N., McJones, P., Blasgen, M. W., Lorie, R. A., Price, T. G., Putzulu, G. F., and Traiger, I. L., "The Recovery Manager of a Data Management System", IBM Research Report RJ2623, August 1979.

[Gray 78] Gray, J. N., "Notes on Data Base Operating Systems", Operating Systems An Advanced Course, *Lecture Notes in Computer Science 60,* Springer-Verlag, 1978, pp. 393-481.

[Herlihy 82] Herlihy, M., and Liskov, B., "A Value Transmission Method for Abstract Data Types", *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 4, Oct. 1982, pp. 527-551.

[Jagau 83] Jagau, August-Wilhelm, "File Access Tokens", LOCUS Internal Memorandum 13, Center for Experimental Computer Science, University of California, Los Angeles, March 2, 1983.

[Jagau 82] Jagau, August-Wilhelm, "Process Management Under LOCUS", LOCUS Internal Memorandum 11, Center for Experimental Computer Science, University of California, Los Angeles, December 16, 1982.

[Jessop 82] Jessop, W. H., et al., "The Eden Transaction-Based File System", *Proceedings of the Second Symposium on Reliability in Distributed Software in Database Systems,* Pittsburgh, Pennsylvania, July 19-21, 1982, pp. 163-169.

[Kernighan 78] Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language,* Prentice Hall, Englewood Cliffs, NJ, 1978.

[Lampson 79] Lampson, B. W. and Sturgis, H. E., "Crash Recovery in a Distributed Data Storage System", XEROX Palo Alto Research Center, April 1979.

[LeLann 81] LeLann, Gerard, "A Distributed System for Real-Time Transaction Processing", *IEEE Computer Magazine,* Vol. 14, No. 2, February 1981, pp. 43-48.

[Lindsay 79] Lindsay, B. G., Selinger, P. G., Galtieri, C., Gray, J. N., Lorie, R. A., Price, T. G., Putzolu, F., Traiger, I. L., and Wade, B. W., "Notes on Distributed Databases", IBM Research Report RJ2571(33471), IBM Research Laboratory, San Jose, CA, July 14, 1979, pp. 44-50.

[Liskov 82] Liskov, Barbara, and Scheifler, Robert, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages,* Albuquerque, NM, January 1982, pp. 7-19.

[Liskov 81] Liskov, B. et al., "CLU Reference Manual", *Lecture Notes in Computer Science 114,* Goos and Hartmanis editors, Springer-Verlag, Berlin, 1981.

[Liskov 79] Liskov, Barbara, "Primitives for Distributed Computing", *Operating Systems Review,* Vol. 13, No. 5, pp. 33-42, December 1979.

[Menasce 77] Menasce, D.A., Popek, G.J., and Muntz, R.R., "A Locking Protocol for Resource Coordination in Distributed Systems," Technical Report UCLA-ENG-7808, Department of Computer Science, UCLA, October 1977.

[Menasce 80] Menasce, D. A., Popek, G. J., and Muntz, R. R., "A Locking Protocol for Resource Coordination in Distributed Databases", ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980, pp. 103-108.

[Metcalfe 76] Metcalfe, R.M. and Boggs D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM,* Vol. 19, No.7, July 1976, pp. 395-404.

[Moore 82a] Moore, J. D. "Simple Nested Transactions in LOCUS: A Distributed Operating System", Master's Thesis, Computer Science Department, University of California, Los Angeles, 1982.

[Moore 82b] Moore, Johanna D., Mueller, Erik T., and Popek, Gerald J., "Nested Transactions and Locus", unpublished paper, Center for Experimental Computer Science, University of California, Los Angeles, October 1982.

[Moss 82] Moss, J. Eliot B., "Nested Transactions and Reliable Computing", *Proceedings of the Second IEEE Symposium on Reliability in Distributed Software and Database Systems,* Pittsburgh, PA, July 1982.

[Moss 81] Moss, J. Eliot B., "Nested Transactions: An Approach to Reliable Distributed Computing", Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, M.I.T., 1981.

[Mueller 83a] Mueller, Erik T., Moore, Johanna D., Popek, Gerald J., "A Nested Transaction Mechanism for LOCUS", *Proceedings of the Ninth Symposium on Operating Systems Principles,* Bretton Woods, NH, October 10-13, 1983.

[Mueller 83b] Mueller, Erik T., "Implementation of Nested Transactions in a Distributed System", Master's Thesis, Computer Science Department, University of California, Los Angeles, 1983.

[Mueller 81] Mueller, Erik T., "An Incremental Stable Storage System for Guardians", S.B. Thesis, M.I.T. Department of Electrical Engineering and Computer Science, May 1981.

[Nelson 81] Nelson, Bruce J., "Remote Procedure Call", Report Number CSL-81-9, XEROX Palo Alto Research Center, May 1981.

[Parker 82] Parker, D.S., and Ramos, R., "A Distributed File System Architecture Supporting High Availability", *Proceedings of the Sixth Berkeley Conference on Distributed Data Management and Computer Networks,* Asilomar, California, February 1982.

[Parker 83] Parker, D. Stott, Popek, Gerald J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C., "Detection of Mutual Inconsistency in Distributed Systems", IEEE Transactions on Software Engineering, May 1983, pp. 240-247.

[Popek 83a] Popek, Gerald J., and Walker, Bruce J., "Network Transparency and its Limits in a Distributed Operating System", unpublished paper, Center for Experimental Computer Science, University of California, Los Angeles, January 1983.

[Popek 83b] Popek, Gerald J., Thiel, Greg, and Kline, Charles S., "Recovery of Replicated Storage in Distributed Systems", unpublished paper, Center for Experimental Computer Science, University of California, Los Angeles, January 1983.

[Popek 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G., "LOCUS: A Network Transparent, High Reliability Distributed System", *Proceedings of the Eighth Symposium on Operating Systems Principles,* Pacific Grove, California, December 1981.

[Randell 75] Randell, B., "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering,* Vol. 1, No. 2, 1975.

[Reed 78] Reed, D. P., "Naming and Synchronization in a Decentralized Computer System", Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, M.I.T., 1978.

[Reiher 82] Reiher, P., "Pack Porting Users Manual", LOCUS Internal Memorandum 4, Center for Experimental Computer Science, University of California, Los Angeles, September 1982.

[Ritchie 78] Ritchie, D. and Thompson, K., "The UNIX Timesharing System", *Bell System Technical Journal,* Vol. 57, No. 6, Part 2, July - August 1978, pp. 1905-1930.

[Rudisin 80] Rudisin, G., "Architectural Issues in a Reliable Distributed File System", Master's Thesis, Computer Science Department, University of California, Los Angeles, 1980.

[Saltzer 81] Saltzer, J. H., Reed, D. P., and Clark, D. D., "End-to-End Arguments in System Design", *Proceedings of the 2nd International Conference on Distributed Computing Systems,* Versailles, France, April 1981.

[Stonebraker 79] Stonebraker, Michael, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 3, May 1979, pp. 188-194.

[Svobodova 81] Svobodova, L., "Recovery in Distributed Processing Systems", unpublished paper, INRIA, Rocquencourt, France, July 1981, revised version to appear in IEEE Transactions on Software Engineering.

[Svobodova 80] Svobodova, L., "Management of Object Histories in the Swallow Repository", Technical Report MIT/LCS/TR-243, Laboratory for Computer Science, M.I.T., 1980.

[Thomas 78] Thomas, R.F., "A Solution to the Concurrency Control Problem for Multiple Copy Data Bases," *Proceedings Spring COMPCON,* Feb 28 - Mar 3, 1978.

[Walker 83] Walker, Bruce J., Popek, Gerald J., English, R. M., Kline, C., and Thiel, G., "The LOCUS Distributed Operating System", *Proceedings of the Ninth Symposium on Operating Systems Principles,* Bretton Woods, NH, October 10-13, 1983.

APPENDIX A

NESTED TRANSACTION ALGORITHM

This appendix contains a detailed description of the nested transaction algorithm developed in Chapters 4 and 5. The description omits the implementation of remote member processes for brevity. That is, member processes of a particular transaction are required to execute at a single site. This of course does not disallow the invocation of subtransactions at remote sites. Our description also omits the mechanism required to cope with lost, duplicate, delayed and out-of-order messages, although we do handle network partitions. For a description of the modifications which are necessary both to support remote member processes and to cope with an unreliable communications system, see Chapter 6. Our description also does not incorporate the two-phase commit protocol which is required to commit final updates when a top-level transaction commits. The two-phase commit protocol is described in detail in Chapters 7 and 8.

In the first section of this appendix, we specify the language that will be used to describe the nested transaction algorithm. In the second section, we present the datatypes and operations which will be needed in the third section, which is the detailed description of our nested transaction algorithm. A shortened presentation of our nested transaction algorithm may be found in [Mueller 83a].

A.1 Description Language

Our description language is similar to CLU [Liskov 81], except that we have relaxed its strong typing for simplicity. The data entities of the language are called *objects*. Each object is a member of a particular *datatype*. Numbers such as 1, 2, 3, etc., are objects of the Int datatype. The objects TRUE and FALSE are the sole members of the Bool datatype. One may refer to objects using *variables*. A variable is made to refer to an object with an *assignment* statement such as i := 4.

In our description language, one may define a new type simply by equating it to some other type. For example, we might define a new type Site as follows:

Site = Int

Several *constructors* allow the creation of new types from other types. The *oneof* constructor allows a new class of objects to be created. Each object in this new class may be one of many types. For example,

BoolOrInt = Oneof[Bool, Int]

defines a new type BoolOrInt. Each object of this type may actually be a Bool or an Int. The Type{object} operation allows the construction of a Oneof of the specified Type from an object of that type. For example, in order to create an object of type BoolOrInt, we might use:

truth := Bool{TRUE}

One may determine which of the possible types an object actually is by using the Is-Type operation. For example, IsBool(truth) would return TRUE, and IsInt(truth) would return FALSE. Note that our language is not strongly typed, as is CLU. We allow an object to be of many types, although most of the time it will be obvious which type is intended. For example, once we have determined that the above Oneof

object truth is actually boolean, we may use it as if it were of type Bool, although, strictly speaking, it is an object of type Oneof[Bool, Int].

The *structure* constructor, called Struct, allows a tuple to be created which consists of a fixed number of fields, each of which may contain an object of a particular type. For example,

Tid = Struct[Site, Int]

defines a structure Tid which contains two fields: the first which contains an object of type Site, and the second which contains an object of type Int. In order to create an object of the above type, the following statement might be used:

tid1 := Struct{site, num}

This statement causes the variable tid1 to refer to a new structure containing a site identifier and an integer. Another variable tid2 may be made to refer to the same structure as follows:

tid2 := tid1

In order to select a field of a structure, a period followed by the type of the field is placed after the structure. For example,

number := tid1.Int

causes the number variable to refer to the object contained within the Int field of the structure, that is, to set number to the integer in the structure's Int field. Note that the type of the field is used to uniquely identify the field. This will cause no problem for our purposes, since each field of every structure we use will be of a different type.

In our language, fields of a structure may be altered with an assignment statement such as:

tid1.Int := newnumber

If tid1 and tid2 both refer to the same object, as in our example, then tid1.Int and tid2.Int are now both equal to newnumber.

Our last constructor, the *list* constructor, allows the creation of a variable-length tuple of elements of a particular type. For example,

Ancest = List[Tid]

defines a new type Ancest as a list of Tids. Such a list containing two elements would be created as follows:

list := List{tid1, tid2}

It is also possible to create a list of zero elements:

empty := List{}

There are several operations which may be performed on lists. The first one adds an element to the head of a list:*

List$AddHead(Type, *List[Type])

This operation takes an object of some type and a list of that type, and adds the object to the head of the list. Thus the length of the list is increased by one. For example, after executing

IntList = List[Int]

list := List{1, 2, 3}
List$AddHead(0, list)

* We will use an asterisk before an argument to indicate that the operation modifies one of its arguments.

the list becomes {0, 1, 2, 3}. There are several other operations on lists which will be required in our description:

```
List$Length(List) => Int
; Return the length of a list.

List$Empty(List) => Bool
; Return TRUE if the length of the list is zero, otherwise return
; FALSE.

List$Remove(Type, *List[Type])
; Remove the first element in the list which is equal to the first
; argument.

List$Member(Type, List[Type]) => Bool
; Return TRUE if the first argument is equal to some element in the list,
; otherwise return FALSE.

List$Add(Type, List[Type]) => List[Type]
; Add an element to the head of a list and returns a new list.
; Its list argument is unchanged.

List$Head(List[Type]) => Type
; Return the first element of a list.

List$Tail(List[Type]) => List[Type]
; Return a list with the first element removed.

List$RestFrom(List[Type], type) => List[Type]
; Return a list consisting of the first element of the argument list that is
; equal to the element argument, and all elements thereafter in the list.

List$Reverse(List[Type]) => List[Type]
; Return a list containing the elements of the argument list in reverse order.
```

An *iterator* construction allows one to loop through the elements of a list:

```
tids := List{tid1, tid2}

for tid in tids do
        <body>
endfor
```

The above code causes body to be executed with tid set to tid1 the first time around, and then tid2 the second time around. It is permissible to remove elements from a list that is being iterated through in the body of the iterator.

The syntax which we will use in our language for sending a message and receiving a response is:

```
send MESSAGE(<args>) to Site
   receive RESPONSE1(<params>):
       <action1>
   receive RESPONSE2(<params>):
       <action2>
         .
         .
         .
   partition:
       <actionn>
endsend
```

This causes MESSAGE to be sent to Site, and a response awaited from the site. For each of the possible RESPONSEs, a different action may be specified. In addition, an action may be specified should Site be partitioned away. Only one of the actions is executed depending on which of the possible responses is received.

When a message is sent to a site, a message handler of the form:

```
MESSAGE(<params>)
begin
   <body>
end
```

handles the message. It begins execution with the arguments contained in the message. Within the body of a message handler, a *respond* statement allows the message handler to respond to the message which caused it to begin execution:

```
respond MESSAGE(<args>)
```

We assume that the response is properly directed by the system to the sender of the message.

A respond statement may optionally request yet another response to the response as follows:

```
respond MESSAGE(<args>) response
   receive RESPONSE1(<params>):
       <action1>
   receive RESPONSE2(<params>):
       <action2>
          .
          .
          .
   partition:
       <actionn>
endrespond
```

For simplicity, we employ the message construct whether or not the receiver is remote. If the receiver is local, then the network need not be employed. In this case we assume the system directs the message to the proper local message handler.

Our language provides parallelism via the *cofor* iterator:

```
cofor elem in list do
       <body>
endfor
```

The cofor iterator is similar to the for iterator, except that the body of the iterator is executed for each element of the list and these executions are performed in parallel. This construct is used in our algorithm to send a collection of messages out and wait for the corresponding collection of responses.

In our description, we follow the following conventions. Variables are in lower case. We use mixed upper and lower case for procedures and datatypes. Data structures which are global to a site are in all upper case; constants are in all upper case; and message names and message handlers are also in upper case. Comments are preceded by a semicolon.

A.2 Datatypes and Operations

This section describes the datatypes and operations which will be used in our description of the implementation of nested transactions. In addition to the datatypes described in the previous section, we assume the existence of the following types. A Site datatype is used to identify sites in the network. A process unique identifier (Pid) uniquely identifies processes in the network. A transaction unique identifier (Tid) type is used to uniquely identify a transaction, and we assume the following operations:

```
Tid$Generate(Site) => Tid
; Generate a Tid for a transaction whose home site is
; the specified site.

Tid$Home(Tid) => Site
; Return the home site of a transaction, given its Tid.

Tid$Equal(Tid, Tid) => Bool
; Check if two Tids identify the same transaction.
```

The FileName datatype is used to uniquely identify a file. The Args datatypes is used to contain the character-string arguments to a transaction program.

Now we move on to the data structures which are required in our design. Associated with each transaction, be it a top-level transaction or a subtransaction, is a volatile data structure called the *transaction structure* which resides at the transaction home site. Each site in the network has a global list of such transaction structures:

```
TRANS       =   List[Trans]

Trans       =   Struct[Tid, Super, Status, Pid, Members, Files, SilAbt, ToAbort]
Super       =   List[Tid]
Status      =   Oneof[UNDEFINED, COMMITTED, ABORTED]
Members     =   List[Member]
Member      =   Struct[Pid, Subtrans]
Files       =   List[File]
File        =   Struct[FileName, Site, Mode]
```

```
Subtrans    =    Oneof[Null, Tid]
Mode        =    Oneof[READ, MOD]
SilAbt      =    Bool
ToAbort     =    List[Tid]
```

We assume several operations exist which allow us to retrieve, add, and remove transaction structures from a site's list of transaction structures:

```
GetTrans(Tid) => Oneof[Trans, Null]
AddTrans(Tid, Trans)
RemoveTrans(Tid)
```

Associated with each process is a *process structure,* which contains the following information: the Pid of the process, a reference to the transaction structure of the transaction it is running as a part of, if any, and a flag indicating whether or not this process is the top-level process of a transaction:

```
Process     =    Struct[Pid, Type, TopProc]
Type        =    Oneof[Trans, NonTrans]
TopProc     =    Bool
NonTrans    =    Null
```

We assume several routines exist to create, start, and destroy processes:

```
CreateProcess(FileName, Args) => Process
StartProcess(Process)
DestroyProcess(Process)
```

In each partition, one of the sites storing a particular file is designated the *transaction synchronization site* (TSS) for the file. This site manages synchronization for the file and provides data access. Our algorithms assume the existence of a *topology change procedure* [English 83] which enforces the partition model described in Chapter 2 and maintains certain system data structures. In particular, it is assumed that this mechanism decides on a TSS for each file in each partition, provides a mapping mechanism which allows us to determine the TSS in each partition, elects

new TSSs when the network topology changes and handles any lock conflicts. Thus we assume the following routines:

```
Inaccessible(Site) => Bool
; Returns TRUE if the specified Site is accessible.

GetTss(FileName) => Site
; Returns the TSS for a file in the caller's partition.
```

Each site maintains a collection of files in stable storage. Several operations allow us to create a new stable file, retrieve the state of an existing stable file, atomically replace the state of a stable file, and remove a stable file:

```
stable FILES = List[Struct[FileName, FileState]]

GetStableState(FileName) => Oneof[FileState, Null]
CreateStableState(FileName, FileState)
ReplaceStableState(FileName, FileState)
RemoveStableState(FileName)
```

We assume that the system is responsible for maintaining replicated file copies. When the ReplaceStableState operation is called, the new state of the file must be propagated to the other sites storing the file. The Locus file update propagation and recovery mechanisms are described in [Edwards 82].

The locking and recovery information maintained at the TSS for a file involved with a transaction is called a *t-lock*. Each site maintains a list of volatile t-locks:

```
TLOCKS      =    List[Struct[FileName,Tlock]]
Tlock       =    Struct[FileState, Holders, ReadRetainers, WriteRetainers]

Holders          =    Oneof[Null, ReadHolders, WriteHolder]
ReadHolders      =    List[ReadHolder]
ReadHolder       =    Struct[Tid, Super]
WriteHolder      =    Struct[Tid, Super, FileState]
ReadRetainers    =    List[ReadRetainer]
ReadRetainer     =    Struct[Tid, Super]
WriteRetainers   =    Struct[VersionStack, Ancest]
VersionStack     =    List[Version]
```

```
Version        =    Struct[Tid, FileState]
UsingSites     =    List[Site]
Ancest         =    List[Tid]
```

Several operations allow us to retrieve a t-lock, add a t-lock to the site's list of t-locks, and remove a t-lock from the site's list of t-locks:

```
GetTlock(FileName) => Oneof[Tlock, Null]
AddTlock(FileName, Tlock)
RemoveTlock(FileName)
```

We make the assumption that operations on a particular t-lock are serialized, although this is not specified explicitly in our description. That is, only one handler may be performing an operation on a t-lock (such as TssOpen, TssClose, TssCommit, to be discussed) at a given time, and each operation runs to completion before the next operation is allowed to proceed.

Finally, there are several other operations which will be needed in our routines:

```
Members$Find(Members, Pid) => Oneof[Member, Null]
; Return the entry for the specified process in the member process list.

Members$EnterSub(*Members, Pid, Tid)
; Enter a subtransaction for the specified process in the member process
; list.

Members$RemoveSub(*Members, Pid)
; Remove the subtransaction entry for the specified process in the member
; process list.

Ancest$GetTopTid(Ancest) => Tid
; Retrieve the top-level Tid from the list of ancestors.

Ancest$GetParent(Ancest) => Oneof[Tid, Null]
; Retrieve the parent Tid from the list of ancestors.

ReadRetainers$Find(ReadRetainers, Tid) => Oneof[ReadRetainer, Null]
; Return the entry for the specified read retainer.

ReadRetainers$Remove(*ReadRetainers, Tid)
; Remove the entry for the specified read retainer.
```

```
ReadHolders$Find(ReadHolders, Tid) => Oneof[ReadHolder, Null]
; Return the entry for the specified read holder.

ReadHolders$Remove(*ReadHolders, Tid)
; Remove the entry for the specified read holder.

Files$Find(Files, FileName) => Oneof[File, Null]
; Return the entry for the specified file in the participant file list.

; Definition of return codes.
Status = Oneof[SYSABORT, Int]
Success = Oneof[SUCCESS, FAILURE, USERABORT, PARTITIONED]
```

A.3 Description of Algorithm

Here is the description of our nested transaction algorithm.

```
;*****************************************************************************
;
; N E S T E D   T R A N S A C T I O N   M E C H A N I S M
;
; The following routines provide the interface to the nested transaction
; mechanism in the form of system calls:
;
; FILE RELATED:
;       Open        --      Request a lock on a file
;       Read        --      Read the state of a locked file
;       Write       --      Modify the state of a locked file
;       Close       --      Release a held lock on a file
;
; CONTROL RELATED:
;       RelCall     --      Invoke a transaction
;       Fork        --      Fork another transaction member process
;       Exit        --      Terminate a transaction member process
;
;*****************************************************************************

Open(file: FileName, mode: Mode, proc: Process) => Success
begin
   trans := proc.Type
   send TSSOPEN(file, trans.Tid, trans.Super, mode) to GetTss(file)
      receive RTSSOPEN(success: Success):
         if success = SUCCESS then
            AddFile(trans, file, GetTss(file), mode)
         endif
         return(success)
      partition:
         Abort(trans, SYSABORT)
   endsend
end

Close(file: FileName, proc: Process)
begin
   trans := proc.Type
   send TSSCLOSE(file, trans.Tid) to GetTss(file)
      receive RTSSCLOSE:
         return
      partition:
         Abort(trans, SYSABORT)
   endsend
end

Read(file: FileName, proc: Process) => Oneof[FileState, FAILURE]
begin
   trans := proc.Type
   send TSSREAD(file, trans.Tid) to GetTss(file)
      receive RTSSREAD(filestate: FileState):
         return(filestate)
      partition:
```

```
        Abort(trans, SYSABORT)
    endsend
end

Write(file: FileName, filestate: FileState, proc: Process) => Success
begin
    trans := proc.Type
    send TSSWRITE(file, trans.Tid, filestate) to GetTss(file)
        receive RTSSWRITE(success: Success):
            return(success)
        partition:
            Abort(trans, SYSABORT)
    endsend
end

;
; RelCall -- This routine is invoked when a process wishes to invoke
;            a transaction using the relcall system call. The specified
;            file is invoked as a transaction with the specified arguments
;            at the specified site.
;

Abort = Struct[Status, Success]

RelCall(file: FileName, args: Args, site: Site, proc: Process) =>
        Oneof[COMMIT, Abort, INACC, PARTITIONED]
begin
    topflag := IsTrans(proc.Type)
    pid := proc.Pid

    if topflag then
        trans := proc.Type
        super := List$Add(trans.Tid, trans.Super)
    else
        super := List{}
    endif

    if Inaccessible(site) return(INACC) endif

    ; Generate a tid for the new transaction.
    tid := Tid$Generate(site)
    ; If calling process is running as a transaction, enter subtransaction
    ; in the calling process's entry in the member process list.
    if !topflag then
        Members$EnterSub(trans.Members, pid, tid)
    endif

    ; Now start the transaction and wait for completion message or
    ; partition.
    send STARTTRANS(file, args, tid, super, pid) to site
      receive TOPCOMMIT:
        ; Called top-level transaction committed.
```

```
  return(COMMIT)

receive TOPABORT(status: Status, success: Success):
  ; Called top-level transaction aborted, return exit status to
  ; caller of system call.
  return(Abort{Struct{status, success}})

receive REQCOMMIT(files: Files):
  ; Called subtransaction wishes to commit.

  ; First add child's list of files to our list.
  for file in files do
     AddFile(trans, file.FileName, file.Site, file.Mode)
  endfor

  ; Send message granting the commit and wait for results or
  ; partition.
  send GRTCOMMIT to site
     receive SUBCOMMIT:
        ; Commit of subtransaction successful.

        ; Remove subtransaction entry for this process in the member
        ; process list.
        Members$RemoveSub(trans.Members, pid)

        ; Called subtransaction committed.
        return(COMMIT)

     receive SUBCMTFAIL(success: Success):
     partition:
        ; The commit of a subtransaction failed, so abort the caller.

        ; Remove subtransaction entry for this process in the member
        ; process list.
        Members$RemoveSub(trans.Members, pid)

        ; Abort self and exit (do not return from RelCall system call)
        Abort(trans.Tid, SYSABORT)
        Exit(SYSABORT, proc)
  endsend

receive SUBABORT(status: Status, success: Success):
  ; Remove subtransaction entry for this process in the member
  ; process list.
  Members$RemoveSub(trans.Members, pid)
  return(Abort{Struct{status, success}})

partition:
  if topflag return(PARTITIONED)
  else return(Abort{Struct{SYSABORT, PARTITIONED}})
  endif
```

```
   endsend
end

;
; Fork -- This routine must be executed whenever a transaction process
;         forks, so that the new child process can be added to the
;         transaction's member process list.
;

Fork(parent: Process, child: Process)
begin
   trans := parent.Type
   List$AddHead(Struct{child.Pid, Null{}}, trans.Members)
   child.TopProc := FALSE
end

;
; Exit -- This routine is executed whenever a transaction process
;         terminates using the exit system call. The process passes
;         along an exit status to this routine.
;

Exit(status: Status, proc: Process)
begin
   trans := proc.Type
   pid := proc.Pid
   member := Members$Find(trans.Members, pid)

   ; Add an active subtransaction of the exiting process to the transaction's
   ; list of subtransactions to abort.
   if IsTid(member.Subtrans) then
      List$AddHead(member.Subtrans, trans.ToAbort)
   endif

   ; Remove exiting process from member process list.
   Members$Remove(trans.Members, pid)
   if (!proc.TopProc) and (trans.Status != ABORTED) then
   ; If the top-level transaction process is not exiting and an abort
   ; has not been initiated, just destroy the exiting process and we are
   ; done.
      DestroyProcess(proc)
      return
   endif

   ; Determine if there are any remaining member processes.
   empty := List$Empty(trans.Members)

   ; Determine if there are any inaccessible participant files.
   inaccp := FALSE
   for file, site, mode in trans.Files do
      if Inaccessible(site) then
         inaccp := TRUE
```

```
      break
    endif
endfor

if (trans.Status = ABORTED) or (status != COMMIT) or (!empty) or inaccp then
; If an abort has already been initiated, or if the top-level transaction
; process exits with non-zero status, or if there are any member
; processes remaining, or if there are any inaccessible participant
; files, we must perform abort processing.

    ; Initiate abort processing if it has not already been initiated.
    if trans.Status != ABORTED then
        Abort(trans, status)
    endif

    if empty then
    ; If the member process list is empty, the last member process
    ; is exiting, so we complete the abort processing by
    ; aborting descendants, updating locks, and returning control
    ; to the caller (unless this is a silent abort).

        if trans.Silabt = FALSE then
            AbortDesc(trans);
            AbortTlocks(trans);
        endif

        ; Return appropriate information to caller.
        if List$Empty(trans.Super) then
        ; We are the top-level transaction.
            respond TOPABORT(status, USERABORT)
        else
        ; We are not the top-level transaction.
            respond SUBABORT(status, USERABORT)
        endif

        ; Remove transaction from the site's list of transactions.
        RemoveTrans(trans.Tid)
        ; Destroy the exiting process and we are done.
        DestroyProcess(proc)
        return
    endif

else
; We are the top process and the member process list is empty and the exit
; status is zero: transaction wishes to commit.

    if List$Empty(trans.Super) then
    ; We are a top-level transaction.

        ; First we commit-lock-update our files. In an actual
        ; implementation, this final update for top-level transaction
        ; commit would be accomplished using a two-phase commit protocol.
```

```
        ; This is ignored here for simplicity.
        success := CommitTlocks(trans)

        if success != SUCCESS then
            AbortTlocks(trans)
            respond TOPABORT(SYSABORT, success)
        else
            respond TOPCOMMIT
        endif

        ; Remove transaction from the site's list of transactions.
        RemoveTrans(trans.Tid)

        ; Destroy the exiting process and we are done.
        DestroyProcess(proc)
        return

    else
    ; We are not a top-level transaction.
        respond REQCOMMIT(trans.Files) response
            receive GRTCOMMIT:
                ; First commit-lock-update our files.
                success := CommitTlocks(trans)

                if success != SUCCESS then
                    respond SUBCMTFAIL(success)
                else
                    respond SUBCOMMIT
                endif

            partition:
            ; No need to abort-lock-update files. Other mechanisms accomplish
            ; this: topology change in our partition, commit failure recovery
            ; in our parent's partition, and topology change in a partition
            ; containing neither us nor our parent.

                ; Remove transaction from the site's list of transactions.
                RemoveTrans(trans.Tid)
                    ; Destroy the exiting process and we are done.
                DestroyProcess(proc)
                return
        endrespond

        ; Remove transaction from the site's list of transactions.
        RemoveTrans(trans.Tid)

        ; Destroy the exiting process and we are done.
        DestroyProcess(proc)
        return

    endif
endif
```

```
end

; Abort -- Cause a transaction to abort by forcing all the transaction's
;         member processes to exit. The aborting of descendants and
;         abort lock updating takes place when the last member process
;         exits.

Abort(trans: Trans, status: Status)
begin
   ; Simply return if transaction has already begun aborting.
   if trans.Status = ABORTED then return endif

   ; Set status of transaction to ABORTED.
   trans.Status := ABORTED

   ; Force exit of all member processes.
   for proc, subtrans in trans.Members do
      Exit(status, proc)
   endfor
end

; AbortDesc -- Abort all descendants of an aborting transaction by sending
;              FORCEABT messages to all accessible sites in the ToAbort list
;              and waiting for responses or a partition.

AbortDesc(trans: Trans)
begin
   cofor tid in trans.ToAbort do
      if !Inaccessible(Tid$Home(tid)) then
         send FORCEABT(tid) to Tid$Home(tid)
          receive RFORCEABT:
            continue
          partition:
            continue
         endsend
      endif
   endcofor
end

; AbortTlocks -- Abort-lock-update t-locks for transaction by sending TSSABT
;                for each participant file and waiting for responses
;                or a partition.

AbortTlocks(trans)
begin
   tid := trans.Tid
   ancest := List$Add(tid, trans.Super)
   cofor file, site, mode in trans.Files do
      if !Inaccessible(site) then
         send TSSABT(file, tid, ancest) to site
          receive RTSSABT:
            continue
```

```
            partition:
              continue
            endsend
        endif
    endcofor
end

; CommitTlocks -- Commit-lock-update t-locks for all files involved with
;                 transaction by sending TSSCMT messages and waiting for
;                 responses or a partition. Returns with error code if
;                 the commit was unsuccessful, in which case the caller,
;                 which is the parent, must abort itself.

CommitTlocks(trans) => Success
begin
    tid := trans.Tid
    ancest := List$Add(tid, trans.Super)
    success := TRUE

    ; Attempt to perform all commit-lock-updates. Fail if any update fails,
    ; or if there is a partition. An optimization would be to discontinue
    ; processing once the first failure is encountered.
    cofor file, site, mode in trans.Files do
        if Inaccessible(site)
            success := FALSE
            continue
        else
            send TSSCMT(file, tid, ancest) to site
              receive RTSSCMT(code: Success):
                success := success and (code = SUCCESS)
                continue
              partition:
                success := FALSE
                continue
            endsend
        endif
    endcofor
    if success then return(SUCCESS) else return(FAILURE) endif
end

;
; AddFile -- This routine must be invoked when a transaction
;            opens a file in order to the file to be added to
;            the transaction's participant list.
;

AddFile(trans: Trans, file: FileName, site: Site, mode: Mode)
begin
    existing := Files$Find(trans.Files, file)

    if IsNull(existing) then
    ; File not in our list. Add it.
```

```
        List$AddHead(Struct{file, site, mode}, trans.Files)
    else
    ; File already in our list. Upgrade mode if necessary.
        if mode = MOD then
            existing.Mode = MOD
        endif
    endif
end

;
; AnyInacc -- Determine if any transaction in a list is inaccessible.
;

AnyInacc(tids: List[Tid]) => Bool
begin
    for tid in tids do
        if Inaccessible(Tid$Home(tid)) then return(TRUE) endif
    endfor
    return(FALSE)
end

;
; FORCEABT -- This message handler forces the specified transaction to abort.
;

FORCEABT(tid: Tid)
begin
    trans := GetTrans(tid)
    if !IsNull(trans) then
        Abort(trans, SYSABORT)
    endif
    respond RFORCEABT
end

;
; STARTTRANS -- This message handler causes a new transaction to be started
; at the site where the message is received. The messages TOPABORT, TOPCOMMIT,
; REQCOMMIT, SUBABORT, which are sent from the Exit routine, will be
; directed to the process which sent the STARTTRANS message.
;

STARTTRANS(file: FileName, args: Args, tid: Tid, super: Super, caller: Pid)
begin
    ; First create a new process which will execute instructions in the
    ; given file and with the given arguments.
    newproc := CreateProcess(file, args)

    ; Set up a transaction structure for the new transaction
    trans := Struct{tid, super, UNDEFINED, caller,
                List{Struct{newproc.Pid, Null{}}},
                List{}, List{}, FALSE})
```

```
   ; Set up appropriate fields in the process structure of the new process.
   newproc.Type := trans
   newproc.TopProc := TRUE

   ; Add the new transaction to the site's list of transactions.
   AddTrans(tid, trans)

   ; Finally, start the execution of the new process, passing on
   ; to the Exit routine the responsibility of responding to
   ; the STARTTRANS message.
   StartProcess(newproc)
end

;*****************************************************************************
;
; TransTopchg -- This routine is invoked when there is a network topology
;                change by the topology change procedure in order to perform
;                those topology change functions that are required in the
;                nested transaction algorithm.
;
;*****************************************************************************

TransTopchg()
begin
   ; Abort any transactions having an inaccessible superior.
   for trans in TRANS do
      if AnyInacc(trans.Super)
      ; A superior is inaccessible.
         trans.SilAbt := TRUE
         Abort(trans, SYSABORT)
      endif
   endfor

   ; Perform recovery for all t-locks at this site.
   for tlock in TLOCKS do
      TopchgTssCleanup(tlock)
   endfor
end

;*****************************************************************
;
; T-LOCK MESSAGE HANDLERS:  These message handlers form the interface
;                           to the transaction locking and recovery
;                           mechanism.
;
;*****************************************************************

; TSSOPEN -- This message handler allows a US to request a lock on
;            a file for which this site is the TSS.

TSSOPEN(file: FileName, tid: Tid, super: Super, mode: Mode)
begin
```

```
    success := TssOpen(file, tid, super, mode)
    respond RTSSOPEN(success)
end

; TSSCLOSE -- This message handler allows a US to release a held lock.

TSSCLOSE(file: FileName, tid: Tid)
begin
    success := TssClose(file, tid)
    respond RTSSCLOSE
end

; TSSREAD -- This message handler allows a US to read the state of a file
;           for which it holds a lock.

TSSREAD(file: FileName, tid: Tid)
begin
    filestate := TssRead(file, tid)
    respond RTSSREAD(filestate)
end

; TSSWRITE -- This message handler allows a US to modify the state of a file
;           for which it holds a write lock.

TSSWRITE(file: FileName, tid: Tid, filestate: FileState)
begin
    success := TssWrite(file, tid, filestate)
    respond RTSSWRITE(success)
end

; TSSCMT -- This message handler accomplishes a commit-lock-update for the
;           specified file and transaction.

TSSCMT(file: File, tid: Tid, ancest: Ancest)
begin
    success := TssCommit(file, tid, ancest)
    respond RTSSCMT(success)
end

; TSSABT -- This message hander accomplishes an abort-lock-update for the
;           specified file and transaction.

TSSABT(file: File, tid: Tid, ancest: Ancest)
begin
    TssAbort(file, tid, ancest)
    respond RTSSABT
end

;
; TssOpen -- This routine is invoked at the TSS for a file in order
;           for a transaction to request a lock on the file.
;
```

```
TssOpen(file: FileName, tid: Tid, super: Super, mode: Mode) => Success
begin
   ancest := List$Add(tid, super);
   tlock := GetTlock(file)

   if IsNull(tlock) then
   ; There is no t-lock for this file at this site. This is the
   ; first open of the file.

      ; Create a new t-lock for the file.
      tlock := Struct{GetStableState(file), Null{}, List{},
                  Struct{List{}, List{}}};
      switch mode
            case READ:
             tlock.Holders := ReadHolders{List{Struct{tid, super}}}
            case MOD:
             tlock.Holders := WriteHolder{Struct{tid, super, tlock.FileState}}
      endswitch

      ; Add t-lock to the site's list of t-locks.
      AddTlock(file, tlock)
      return(SUCCESS)
   else
   ; File already open.
      switch mode
          case READ:

         if IsWriteHolder(tlock.Holders) then
         ; Someone holds a write lock. It may be tid, however, the
         ; ability for a transaction to possess both READ and MOD access
         ; to a file is not implemented here.
            return(FAILURE)

         elseif IsReadHolders(tlock.Holders) then
         ; Someone holds a read lock.
            reader := ReadHolders$Find(tlock.Holders.ReadHolders, tid)
            if !IsNull(reader) then
            ; Read lock already held by tid.
               ; Multiple opens will never reach the TSS.
               return(FAILURE)
            endif
            ; Tid does not already hold read lock, fall through.
         else
         ; No locks are held.
            ; Set up an empty read holder list.
            tlock.Holders := ReadHolders{List{}}
         endif

         if !CheckWretns(tlock, ancest) then
         ; There are write retainers that are not ancestors,
         ; so deny request.
            if List$Empty(tlock.Holders.ReadHolders) then
```

```
            tlock.Holders := Null{}
          endif
          return(FAILURE)
        endif

        ; Grant a new held read lock.
        ; Add read holder to list of holds.
        List$AddHead(Struct{tid, super}, tlock.Holders.ReadHolders)
        return(SUCCESS)
         case MOD:

        if IsWriteHolder(tlock.Holders) then
           ; Someone holds a write lock.
           if tlock.Holders.WriteHolder.Tid = tid then
           ; Write lock already held by tid.
              ; Multiple opens will never reach the TSS.
              return(FAILURE)
           else
           ; Write lock held by another tid, so deny request.
              return(FAILURE)
           endif

         elseif IsReadHolders(tlock.Holders) then
         ; Someone holds a read lock. It may be tid, however, lock
         ; upgrading is not implemented here.
            return(FAILURE)

         endif

         ; No held locks.
         if !CheckRretns(tlock, ancest) or !CheckWretns(tlock, ancest) then
         ; There are retainers that are not ancestors, so deny request.
            return(FAILURE)
         endif

         ; Grant new held write lock.
         tlock.Holders := WriteHolder{Struct{tid, super, tlock.FileState}}
         return(SUCCESS)
      endswitch
   endif
end

;
; TssRead -- This routine is invoked at the TSS for a file in order
;            for a transaction to read the state of a file for which it
;            holds a lock.
;

TssRead(file: FileName, tid: Tid) => Oneof[FileState, FAILURE]
begin
   tlock := GetTlock(file)
   if CheckRead(tlock, tid) then
```

```
        return(tlock.FileState)
    else return(FAILURE)
    endif
end

;
; TssWrite -- This routine is invoked at the TSS for a file in order
;             for a transaction to modify the state of a file for which it
;             holds a write lock.
;

TssWrite(file: FileName, tid: Tid, filestate: FileState) => Success
begin
    tlock := GetTlock(file)
    if CheckWrite(tlock, tid) then
        tlock.FileState := filestate
        return(SUCCESS)
    else return(FAILURE)
    endif
end

;
; CheckRead -- This routine verifies that a transaction holds a read or
;              write lock for a file.
;

CheckRead(tlock: Tlock, tid: Tid)
begin
    if IsWriteHolder(tlock.Holders) then
        return(tid = tlock.Holders.WriteHolder.Tid)
    elseif IsReadHolders(tlock.Holders) then
        return(ReadHolders$Member(tid, tlock.Holders.ReadHolders))
    else return(FALSE)
    endif
end

;
; CheckWrite -- This routine verifies that a transaction holds a write
;               lock for a file.
;

CheckWrite(tlock: Tlock, tid: Tid)
begin
    if IsWriteHolder(tlock.Holders) then
        return(tid = tlock.Holders.WriteHolder.Tid)
    else return(FALSE)
    endif
end

;
; TssClose -- This routine is called by a transaction in order for its
;             held lock to become a retained lock.
```

```
;

TssClose(file: FileName, tid: Tid)
begin
   tlock := GetTlock(file)
   if !CheckRead(tlock, tid) then return endif
   TssClose1(tlock, tid, mode)
end

TssClose1(tlock: Tlock, tid: Tid)
begin
   if IsWriteHolder(tlock.Holders) then
   ; Tid holds write lock.

      if !TopVerstkIsTid(tlock, tid) then
      ; Tid does not already retain a write lock, so add a new retainer.

         retainer := ReadRetainers$Find(tlock.ReadRetainers, tid)
         if !IsNull(retainer) then
         ; Remove a retained read by tid.
            ReadRetainers$Remove(tlock.ReadRetainers, tid)
         endif
         Tlock$VerStkPush(tlock, tid,
                          tlock.Holders.WriteHolder.Super,
                          tlock.Holders.WriteHolder.FileState)
      endif

      ; Remove the held lock.
      tlock.Holders := Null{}

   else if IsReadHolders(tlock.Holders) then
   ; Tid holds read lock.

      reader := ReadHolders$Find(tlock.Holders.ReadHolders, tid)

      retainer := ReadRetainers$Find(tlock.ReadRetainers, tid)
      if !TopVerstkIsTid(tlock, tid) and IsNull(retainer) then
      ; Transaction does not already retain a lock, so add a retained
      ; read lock.
         List$AddHead(Struct{tid, reader.super}, tlock.ReadRetainers)
      endif

      ; Remove held read lock.
      ReadHolders$Remove(tlock.Holders.ReadHolders, tid)
      if List$Empty(tlock.Holders) then
         tlock.Holders := Null{}
      endif
   endif
end

;
; TssCleanHelds -- TssClose any write lock holders and any read lock
```

```
;                holders of which we are a superior.
;

TssCleanHelds(tlock: Tlock, tid: Tid)
begin
   if IsWriteHolder(tlock.Holders) then
   ; Close a write lock holder.
      TssClose1(tlock, tlock.Holders.WriteHolder.Tid)

   elseif IsReadHolders(tlock.Holders) then
   ; Close all read lock holders of which we are a superior.
      for readholder in tlock.Holders.ReadHolders do
         if List$Member(tid, readholder.Super) then
         ; We are a superior of the holder, so close the holder.
            TssClose1(tlock, readholder.Tid)
         endif

   endif
end

;
; TssCommit -- Perform the commit-lock-update operation on a file.
;

TssCommit(file: FileName, tid: Tid, ancest: Ancest) => Success
begin
   tlock := GetTlock(file)
   toptid := Ancest$GetTopTid(ancest)
   topflag := Tid$Equal(tid, toptid)
   if topflag = FALSE then
      parent := Ancest$GetParent(ancest)
   endif

   ; Clean up any held locks.
   TssCleanHelds(tlock, tid)

   ; Clean up any write retainers which are inferiors: pop and replace until
   ; top element is an ancestor or bottom of stack is reached.
   while !List$Empty(tlock.WriteRetainers.VersionStack) and
        !List$Member(List$Head(tlock.WriteRetainers.VersionStack).Tid, ancest)
   do
      Tlock$VerStkRepl(tlock)
   endwhile

   ; Clean up any read retainers: remove any read retainers of which we
   ; are a superior.
   for retainer in tlock.ReadRetainers do
      if List$Member(tid, retainer.Super) then
            ReadRetainers$Remove(tlock.ReadRetainers, tid)
      endif
   endfor
   if !List$Empty(tlock.WriteRetainers.VersionStack) and
```

```
  Tid$Equal(tid, List$Head(tlock.WriteRetainer.VersionStack).Tid) then
; Tid retains a write lock.
   if topflag then
   ; Perform commit-lock-update for a top-level transaction.
      Tlock$VerStkPop(tlock)
      TopCommit(filename)
   else
   ; Perform commit-lock-update for a subtransaction.
      if !List$Empty(List$Tail(tlock.WriteRetainer.VersionStack)) and
        Tid$Equal(List$Head(List$Tail(tlock.WriteRetainer.VersionStack)).Tid,
                  parent)
         ; Parent of tid already retains a write lock, so simply
         ; discard top of version stack.
         Tlock$VerStkPop(tlock)
      else
         ; Parent inherits held write lock.
         List$Head(tlock.WriteRetainer.VersionStack).Tid := parent
         tlock.WriteRetainers.Ancest := List$Tail(ancest)
      endif
   endif

else
; Tid retains a read lock.
   retainer := ReadRetainers$Find(tlock.ReadRetainers, tid)

   if IsNull(retainer) then
   ; Tid doesn't retain any lock; the commit fails. This may happen
   ; if we do not force the abort of a transaction having an
   ; inaccessible participant file.
      return(FAILURE)
   endif

   if topflag then
      ReadRetainers$Remove(tlock.ReadRetainers, tid)

      ; If this is the last read retainer (there can be no
      ; write retainers), then we can remove the t-lock.
      if List$Empty(tlock.ReadRetainers) then
         TopAbort(file)
      endif
   else
      if !List$Empty(tlock.WriteRetainer.VersionStack) and
        Tid$Equal(List$Head(tlock.WriteRetainer.VersionStack).Tid,
                  parent)
      ; Parent of tid retains a write lock.
         ReadRetainers$Remove(tlock.ReadRetainers, tid)
      else
         retainer := ReadRetainers$Find(tlock.ReadRetainers, parent)
         if !IsNull(retainer) then
         ; Parent of tid retains a read lock.
               ReadRetainers$Remove(tlock.ReadRetainers, tid)
         else
```

```
            ; Parent of tid inherits retained read.
                retainer.Tid := parent
                retainer.Super := List$Tail(retainer.Super)
            endif
        endif
    endif
    return(SUCCESS)
end

;
; TopCommit -- This routine accomplishes the final top-level commit of
;           a file to stable storage. In an actual implementation
;           these updates would have to be accomplished using a
;           two-phase commit protocol.
;

TopCommit(file: FileName)
begin
    tlock := GetTlock(file)
    ReplaceStableState(file, tlock.FileState)
    RemoveTlock(file)
end

;
; TopAbort -- This routine accomplishes the final abort of a t-lock, simply
;           by removing the t-lock from the site's list of t-locks, and
;           not performing any updates to stable storage.
;

TopAbort(file: FileName)
begin
    RemoveTlock(file)
end

;
; TssAbort -- This routine performs the abort-lock-update operation for a file.
;

TssAbort(file: FileName, tid: Tid, ancest: Ancest)
begin
    tlock := GetTlock(file)
    toptid := Ancest$GetTopTid(ancest)
    topflag := Tid$Equal(tid, toptid)
    if topflag = FALSE then
        super := List$Tail(ancest)
    endif

    ; Clean up any held locks.
    TssCleanHelds(tlock, tid)

    ; Clean up any write retainers which are inferiors: pop and replace until
```

```
  ; top element is a superior or bottom of stack is reached.
  while !List$Empty(tlock.WriteRetainers.VersionStack) and
       !List$Member(List$Head(tlock.WriteRetainers.VersionStack).Tid,
                    super) do
     Tlock$VerStkRepl(tlock)
  endwhile

  ; Clean up any read retainers: remove any read retainers of which we
  ; are an ancestor.
  for retainer in tlock.ReadRetainers do
     if List$Member(tid, retainer.Super) or
       Tid$Equal(tid, retainer:Tid) then
          ReadRetainers$Remove(tlock.ReadRetainers, tid)
     endif
  endfor

  ; Remove Tlock if no remaining read or write retainers.
  if List$Empty(tlock.WriteRetainers.VersionStack) and
    List$Empty(tlock.ReadRetainers) then
     TopAbort(filename)
  endif
end

;
; CheckRretns -- This routine returns TRUE if all read retainers are ancestors.
;

CheckRretns(tlock: Tlock, ancest: Ancest)
begin
   for retainer in tlock.ReadRetainers do
      if !Tid$Member(retainer.Tid, ancest) then return(FALSE) endif
   endfor
   return(TRUE)
end

;
; CheckWretns -- This routine returns TRUE if all write retainers are ancestors.
;

CheckWretns(tlock: Tlock, ancest: Ancest)
begin
   if List$Empty(tlock.WriteRetainers.VersionStack) return(TRUE)
   return(Tid$Member(Tid$Head(tlock.WriteRetainers.VersionStack).Tid,
                    ancest))
end

;
; Version Stack Routines -- These routines push an entry onto the version
;                          stack, pop an entry an replace the version
;                          on top of the version stack into the current
;                          volatile file state (abort action), and pop
;                          and discard an entry on top of the version
;
```

```
;                       stack (commit action).
;

Tlock$VerStkPush(tlock: Tlock, tid: Tid, super: Super, filestate: FileState)
begin
   List$AddHead(Struct{tid, filestate}, tlock.WriteRetainers.VersionStack)
   tlock.WriteRetainers.Ancest := List$Add(tid, super)
end

Tlock$VerStkRepl(tlock)
begin
   tlock.FileState := List$Head(tlock.WriteRetainers.VersionStack).FileState
   tlock.WriteRetainers.VersionStack :=
      List$Tail(tlock.WriteRetainers.VersionStack)

   if List$Empty(tlock.WriteRetainers.VersionStack) then
      tlock.WriteRetainers.Ancest := List{}
   else
      tidontop := List$Head(tlock.WriteRetainers.VersionStack).Tid
      tlock.WriteRetainers.Ancest :=
         List$RestFrom(tlock.WriteRetainers.Ancest, tidontop)
   endif
end

Tlock$VerStkPop(tlock)
begin
   tlock.WriteRetainers.VersionStack :=
      List$Tail(tlock.WriteRetainers.VersionStack)

   if List$Empty(tlock.WriteRetainers.VersionStack) then
      tlock.WriteRetainers.Ancest := List{}
   else
      tidontop := List$Head(tlock.WriteRetainers.VersionStack).Tid
      tlock.WriteRetainers.Ancest :=
         List$RestFrom(tlock.WriteRetainers.Ancest, tidontop)
   endif
end

;
; TopVerstkIsTid -- This routine determines if the top element of the version
;                   stack is for the specified Tid.
;

TopVerstkIsTid(tlock: Tlock, tid: Tid) => Bool
begin
   if List$Empty(tlock.WriteRetainers.VersionStack) then return(FALSE) endif
   return(Tid$Equal(Tid$Head(tlock.WriteRetainers.VersionStack).Tid, tid))
end

;*****************************************************************************
;
; TopchgTssCleanup -- This routine is invoked by TransTopchg to accomplish
```

```
;                the necessary recovery on a file when there is a
;                network topology change.
;
;*****************************************************************************

TopchgTssCleanup(tlock: Tlock)
begin
   ; Clean up any locks held by inaccessible transactions.
   if IsWriteHolder(tlock.Holders) then
      site := Tid$Home(tlock.Holders.WriteHolder.Tid)
      super := tlock.Holders.WriteHolder.Super
      if Inaccessible(site) or AnyInacc(super) then
         TssClose1(tlock, tlock.Holders.WriteHolder.Tid, site)
      endif
   elseif IsReadHolders(tlock.Holders) then
      for readholder in tlock.Holders.ReadHolders do
         site := Tid$Home(readholder.Tid)
         super := readholder.Super
         if Inaccessible(site) or AnyInacc(super) then
            TssClose1(tlock, readholder.Tid, site)
         endif
      endfor
   endif
   ; Remove any read retainers having inaccessible ancestors.
   for retainer in tlock.ReadRetainers do
      if AnyInacc(List$Add(retainer.Tid, retainer.Super)) then
      ; An ancestor is inaccessible, so remove read retainer.
      ReadRetainer$Remove(tlock.ReadRetainers, retainer.Tid)
   endfor

   ; Remove any inaccessible write retainers and their descendants.
   wretns := tlock.WriteRetainers
   if !List$Empty(wretns.VersionStack) then

      ; Search for topmost inaccessible write retainer.
      for tid in List$Reverse(wretns.Ancest) do
         if Inaccessible(Tid$Home(tid)) then
         ; There is an inaccessible ancestor. Abort it.

            super := List$Tail(List$RestFrom(wretns.Ancest, tid))

            ; Clean up any write retainers which are inferiors:
            ; pop and replace until top element is a superior or
            ; bottom of stack is reached.
               while !List$Empty(wretns.VersionStack) and
                     !List$Member(List$Head(wretns.VersionStack).Tid,
                                  super) do
                  Tlock$VerStkRepl(tlock)
               endwhile
            break

         endif
```

```
        endfor

    endif
    ; Remove Tlock if no remaining read or write retainers.
    if List$Empty(tlock.WriteRetainers.VersionStack) and
       List$Empty(tlock.ReadRetainers) then
          TopAbort(filename)
    endif
endif
```

APPENDIX B

SUMMARY STATISTICS

Person Hours Spent in Implementation

The implementation of full nested transactions as described in this report, including detailed design, coding, and debugging took approximately eight person-months.

Number of Lines of Code in Implementation

There are approximately 7208 lines of code in the implementation, which is a little more than twice that required to implement simple nested transactions [Moore 82a] [Moore 82b]. This figure includes both newly added code and modifications which had to be made to existing Locus code to support nested transactions. The current implementation does not contain all of the mechanism necessary to support remote member processes.

The additional functionality gained over simple nested transactions is synchronization of transactions within an entire transaction and the ability to limit the effects of subtransaction abort. With full nested transactions, only the aborting subtransaction's work is undone, while in simple nested transactions, the entire transaction must be aborted. Much of the additional mechanism is devoted to recovering from the wealth of failures which may occur.

The breakdown of code according to the major tasks is as follows:

Module	Lines of Code
Data Structure Definitions	432
Locus Data Structure Modifications	159
Transaction Control	1991
Transaction File Locking	1577
Two-Phase Commit Protocol.	1601
Utility Code	824
Locus Modifications	624
Total	7208

Performance Measurements

In an attempt to estimate the performance overhead incurred by nested transactions, we compare the difference in elapsed time between simply running a program, and running that program as both a top-level transaction and as a subtransaction of some other transaction. We have performed these measurements on Locus, executing on VAX 11/750s using RK07 disks for file storage and a 10 Mbps ring network. The activity being measured was the only user activity taking place in the system at the time the measurements were taken.

The measurements are of a program which modifies data in several files. In each case, the second page of a two page file is updated (page size = 1024 bytes). The files were initialized before each run. The program was run as a non-transaction, as a top-level transaction, and as a subtransaction of a top-level transaction. For each of the cases, a program was run which modifies 0, 1, 2, 4, 6, 8, and 10 files. Each test was run three times. Tests were run where all the files were local, and all the files were remote. All programs were run locally and the copy of the load module to be executed was stored locally. The additional time required to invoke and

return from a remote transaction is comparable to that required for a remote non-transaction process, and thus was not measured.

For transactions, the measurement is of the elapsed time from the time *relcall* was invoked, until it returned. For the non-transaction program, the time is measured from just before the child process is forked to run the program, until the parent process, which waits for the child process to complete, is awakened. The measurements were taken with the *ftime* system call.

The measurements are shown in Tables 1 and 2. The first observation is that the time required to simply invoke and return from a program which performs no file modifications is approximately the same whether the program is run as a non-transaction, top-level transaction, or subtransaction.

In Table 1, we give the measurements for a program which modifies all local files. We can see that running the program as a top-level transaction takes less than twice as long as running the program as a non-transaction. Running a program as a subtransaction is substantially faster than running it as a non-transaction, almost twice the speed.

We can explain these results as follows. Much of the time required for running a non-transaction is taken by the file close operations, which write the file modifications to disk. Much of the time required for running a top-level transaction is taken by the two-phase commit operation, which is used to atomically commit a group of files, and requires more disk writes than simple closes. However, running a program as a subtransaction does not cause any file modifications to be written out to disk. This is because the modifications performed by a subtransaction are only written out when the top-level transaction commits. Thus the time required to run

the program as subtransaction is less than the time required to run it as a non-transaction program. Of course, the subtransaction must update locking information for each of the files before it may commit, as must a top-level transaction. But we can see that these operations do not contribute much to the overall time.

In Table 2, which gives measurements for a program which modifies all remote files, we see that the times for running the program as a subtransaction and as a non-transaction become closer. This is because the time to send messages over the network starts to dominate. The times for a top-level transaction become closer to non-transactions for the same reason, although the two-phase commit protocol requires twice as many messages as would be required for non-atomic commit.

ELAPSED TIME (In seconds) *All Files Local*			
Number of Files	Non-Transaction	Top-Level Transaction	Subtransaction
0	.233	.183	.216
0	.200	.233	.184
0	.183	.217	.183
1	.350	.783	.267
1	.450	.717	.283
1	.350	.800	.250
2	.450	.916	.334
2	.450	.900	.350
2	.483	.883	.300
4	.700	1.233	.483
4	.700	1.083	.416
4	.717	1.200	.417
6	.966	1.483	.566
6	.933	1.483	.584
6	.917	1.450	.617
8	1.183	1.700	.733
8	1.166	1.683	.700
8	1.100	1.733	.683
10	1.384	2.100	.783
10	1.350	2.067	.866
10	1.417	2.133	.767

Table 1: Elapsed Time – All Files Local

ELAPSED TIME (in seconds) *All Files Remote*			
Number of Files	Non-Transaction	Top-Level Transaction	Subtransaction
0	.250	.200	.216
0	.217	.216	.183
0	.216	.200	.183
1	.550	1.050	.533
1	.584	1.050	.517
1	.583	1.050	.500
2	.917	1.450	.833
2	.967	1.450	.834
2	.916	1.484	.833
4	1.650	2.267	1.467
4	1.650	2.216	1.433
4	1.683	2.250	1.450
6	2.350	3.016	2.067
6	2.367	3.066	2.066
6	2.384	3.034	2.017
8	3.000	3.833	2.717
8	2.966	3.900	2.717
8	3.000	3.783	2.650
10	3.700	4.616	3.334
10	3.633	4.684	3.300
10	3.667	4.750	3.333

Table 2: Elapsed Time – All Files Remote

Performance Measurements of Two-Phase Commit

Because the two-phase commit protocol appears to be the primary factor of additional delay in our nested transaction mechanism, we have taken detailed measurements of the protocol. These measurements were made using a tool for measuring activity in local area distributed systems which is currently under development as part of the Locus research effort. More information about this measurement tool may be found in [Goldberg 83].

The performance of the two-phase commit protocol is measured in various conditions. In particular, we show the difference between the time taken to perform a two-phase commit and the time required to simply close the participant files. We

present measurements for three different situations and for varying numbers of files in each case.

As in the last section, the files were all two logical pages long and the second page was updated by the transaction. In measuring the CPU time for an operation involving both a local and a remote site, we have measured the processing time at each site individually and then summed these individual measurements to get a measurement of total CPU time.

The tables summarizing our measurement data may be found on the following pages. These tables are organized as follows. For each case, there are two tables, one reporting measurements for CPU time and the other reporting measurements for elapsed time. Each table contains measurements for file closing, phase 1 of the two-phase commit protocol, phase 2 of the two-phase commit protocol, and a total time for the two-phase commit protocol. Tables 3a and 3b give the measurements for the case when all participant files are local. Tables 4a and 4b show the measurements for the case when half of the participant files are local and the other half are remote. Finally, Tables 5a and 5b present the measurements for the case when all participant files are remote. In each case, we give results for both CPU and elapsed time when the number of files is 2, 4, 6, 10, and 16.

In Table 3a, we see that for 2 files, the CPU time required for two-phase commit is about 10 times as great as that required for file closing. For 16 local files, two-phase commit requires about 5 times as much CPU time as that for file closing. In Table 3b, we see that the elapsed time for 2 local files is about 7.5 times greater for two-phase commit than for file closing, and for 16 local files is about 3 times as great.

Table 4a shows that for one local file and one remote file the CPU time required to perform a two-phase commit of the participant files is about 5.5 times greater than the CPU time required to close these files. For 8 local files and 8 remote files this factor is reduced to about 2.5. The elapsed time figures in Table 4b show that two-phase commit is slower by a factor of 5 than file closing when there is one local and one remote participant file. For 8 local and 8 remote participant files, this factor is reduced to about 1.25.

In Table 5a, we see that the CPU time required to perform a two-phase commit of 2 remote files is about 3 times as great as that for closing the files. For 16 remote files, only twice as much CPU time is required by two-phase commit. As shown in Table 5b, the elapsed time figures indicate that the overhead incurred by two-phase commit is about 3.5 times as great as that incurred by file closing for 2 remote participant files, and about 1.5 times as great for 16 remote participant files.

We note that in all cases, the CPU and elapsed time for both file closing and two-phase commit are approximately linear on the number of files being updated. This makes sense since we are performing the same updates, i.e., modifying a single page, on all our files. Furthermore, we note that two-phase commit performs much worse than simple file closing when all files are local, but not significantly worse than file closing when some files are remote. And when all files are remote two-phase commit only increases CPU usage by a factor of two and elapsed time by a factor of 1.5 with 16 participant files. This is because remote close operations require message exchanges approximately equivalent to the second phase of the two-phase commit protocol.

Comparing the tables, we see that the best elapsed time measurements result from the case when half of the files are remote and half of the files are local. In particular, when the number of participant files is larger than two, the elapsed time of two-phase commit measured in the mixed case was less than the elapsed time in either the all local or the all remote case. In the mixed case, we see some of the advantages of parallel processing since our two-phase commit mechanism exploits such parallelism to the degree that is feasible as discussed in Chapter 7.

We believe that these measurements are encouraging. In cases where a moderate number of participant files, say 6, are involved in the two-phase commit, the protocol is worse than file closing by factors of 4, 2.4, and 2.1 in the all local, mixed and remote cases, respectively. This does not seem a severe penalty for the reliability gained through the use of a two-phase commit protocol. In addition, note that these measurements were taken using the current implementation of simple nested transactions which incorporates none of the optimizations we proposed in Chapter 8.

In conclusion, it appears that the greatest cost in running transactions is in the two-phase commit protocol and that there is little additional cost in the maintenance of locking information. Since the cost of the two-phase commit protocol becomes less significant as the amount of work performed by the entire transaction increases, transactions are not that much more expensive than non-transaction programs.

CPU TIME (in seconds) *All Files Local*				
Number of Files	File Closing	Two-Phase Commit		
		Phase 1	Phase 2	Total
2	.0292	.187	.104	.291
2	.0260	.189	.104	.293
4	.0599	.240	.157	.397
4	.0527	.240	.149	.389
6	.0833	.310	.205	.515
6	.0817	.287	.210	.497
10	.140	.414	.309	.723
10	.137	.392	.314	.706
16	.222	.561	.488	1.049
16	.206	.562	.467	1.029

Table 3a: CPU Time – All Files Local

ELAPSED TIME (in seconds) *All Files Local*				
Number of Files	File Closing	Two-Phase Commit		
		Phase 1	Phase 2	Total
2	.109	.563	.267	.800
2	.107	.559	.239	.798
4	.170	.645	.351	.996
4	.192	.694	.337	1.031
6	.311	.894	.449	1.343
6	.317	.857	.464	1.321
10	.553	1.244	.671	1.915
10	.570	1.333	.626	1.959
16	.862	1.687	.958	2.645
16	.835	1.741	.944	2.685

Table 3b: Elapsed Time – All Files Local

CPU TIME (in seconds) *n/ 2 Files Local, n/ 2 Files Remote*				
Number of Files (n)	File Closing	Two-Phase Commit		
		Phase 1	Phase 2	Total
2	.0697	.222	.113	.335
2	.0549	.218	.118	.336
4	.110	.271	.190	.461
4	.133	.289	.187	.476
6	.190	.336	.257	.593
6	.207	.365	.295	.660
10	.330	.465	.399	.864
10	.326	.476	.377	.853
16	.489	.700	.586	1.286
16	.490	.637	.590	1.227

Table 4a: CPU Time – n/2 Files Local, n/2 Files Remote

ELAPSED TIME (In seconds) *n/ 2 Files Local, n/ 2 Files Remote*				
Number of Files (n)	File Closing	Two-Phase Commit		
		Phase 1	Phase 2	Total
2	.185	.593	.267	.860
2	.157	.609	.260	.869
4	.350	.628	.358	.986
4	.304	.680	.348	1.028
6	.501	.791	.405	1.196
6	.503	.746	.465	1.211
10	.878	.977	.551	1.528
10	.875	.971	.557	1.528
16	1.321	1.066	.589	1.655
16	1.340	1.032	.560	1.592

Table 4b: Elapsed Time -- n/2 Files Local, n/2 Files Remote

CPU TIME (in seconds) *All Files Remote*				
Number of Files	File Closing	Two-Phase Commit		
		Phase 1	Phase 2	Total
2	.124	.230	.142	.371
2	.103	.239	.144	.383
4	.225	.334	.216	.550
4	.205	.311	.210	.521
6	.274	.447	.310	.757
6	.342	.465	.319	.784
10	.479	.603	.479	1.082
10	.498	.607	.472	1.079
16	.737	.859	.797	1.656
16	.734	.884	.806	1.690

Table 5a: CPU Time -- All Files Remote

ELAPSED TIME (in seconds) *All Files Remote*				
Number of Files	File Closing	Two-Phase Commit		
		Phase 1	Phase 2	Total
2	.247	.517	.296	.813
2	.217	.539	.344	.883
4	.479	.730	.361	1.091
4	.433	.735	.398	1.133
6	.659	.867	.493	1.360
6	.642	.943	.455	1.398
10	1.146	1.329	.910	2.239
10	1.127	1.269	.876	2.145
16	1.742	1.850	1.036	2.886
16	1.786	1.669	1.046	2.715

Table 5b: Elapsed Time -- All Files Remote