A Nested Transaction Mechanism for LOCUS

Erik T. Mueller, Johanna D. Moore, and Gerald J. Popek

University of California at Los Angeles

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.

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

1. INTRODUCTION

An *atomic transaction* is a computation consisting of a collection of operations which take place indivisibly in the presence of both failures and concurrent computations. That is, either all of the operations prevail or none of them prevail, and other programs executing concurrently cannot modify or observe intermediate states of the computation. Transactions help to preserve the consistency of a set of shared data objects in the face of failures and concurrent access. While this is true for a centralized system, transactions are even more useful in a distributed environment where it is necessary to deal with additional failure modes such as the partial failure of a distributed computation. Although many transaction implementations currently exist, especially in database management systems [Borr 81] [Gray 81a], these implementations typically have several limitations which make them unsatisfactory for a general programming tool in a distributed environment.

First, transactions cannot be nested. While not significant in a database management system, this restriction prevents users from constructing new transaction applications by composing already existing transactions. The programmer should be able to write transactions as *modules* which may be composed freely, just as procedures and functions may be composed in ordinary programming

languages.*

Second, 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.**

Transactions which are composed of other transactions, or *nested transactions,* have been the subject of much current literature [Moss 81] [Liskov 82] [Reed 78] [Svob 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* [Eswaran 76]. 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 81b] [Moss 82].

An original algorithm for full nested transactions has been developed and an implementation produced for the LOCUS distributed operating system [Popek 81] [Walker 83]. 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 the next section, LOCUS nested transactions provide additional functionality beyond that which is usually proposed. This paper summarizes the work reported in [Mueller 83]. The implementation builds on simple nested transactions, reported in [Moore 82a] and [Moore 82b].

* The System R [Gray 81a] internal implementation contains some support for nested save points, but that functionality is not available to applications.

** Much of the TMF [Borr 81] implementation is within the operating system, although there is no support for nesting.

2. OTHER MODELS

Our model of nested transactions differs in significant ways from traditional views. We discuss these differences in this section.

2.1 Replication

Most 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.

For the sake of availability 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]. 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 replicated data in the context of network partitioning is dealt with extensively in [Faissol 81]. This work breaks the semantics of operations on data objects 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.

2.2 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] [Svob 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. USER MODEL

In this section, we first present an overview of LOCUS and then explain the model of nested transactions which is presented to the user, including transaction invocation, completion, and access to data objects. Finally we discuss the user-visible results of network partitioning.

3.1 Overview of LOCUS

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. 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 personal workstation which connects to the rest of the network via an expensive long-distance telephone line. General partitions are possible in an environment where local networks are connected by gateways. LOCUS employs a sophisticated merge algorithm which allows sites to leave and return to the network gracefully without interrupting service [Walker 83].

* UNIX is a Trademark of Bell Laboratories.

3.2 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.

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

relcall(load-module, args)

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 the input data objects to the transaction. 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, called subtransactions. Subtransactions are frequently referred to merely as transactions. A transaction whose initiator is not a transaction is called a *top-level transaction.* To speak of the related family of transactions, i.e., a top-level transaction and all of its subtransactions, the term *entire transaction* is used. Tree terminology will be used in discussing relationships between transactions. When a transaction calls a subtransaction, the calling transaction will be the *parent* of the subtransaction, and the subtransaction the *child* of the calling transaction. We also speak of *ancestors* and *descendants.* A transaction is an ancestor and descendant of itself. We also use the terms *superior* and *inferior.* A transaction is neither a superior nor an inferior of itself. We sometimes refer to the *transaction invocation tree.*

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

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 complete by issuing the *exit* system call which has an argument indicating the success or failure of the process. In order for a transaction to commit, its top-level process must wait for the other member processes to complete, and then issue an exit call with a successful completion code. If a top-level process issues an exit call with an unsuccessful completion code, the transaction aborts.

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. An example of a transaction program invoking two concurrent subtransactions is given in Appendix A.

3.4 Data Access

LOCUS provides for the manipulation of permanent objects called *files*. 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 the transaction aborts, the updates of the transaction are undone. In order to guarantee serializability of 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 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 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 closes a file, the held lock becomes a retained lock.

3.5 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 *relcall.* 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.

4. BASIC IMPLEMENTATION

This section describes the basic implementation of nested transactions in LOCUS. 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 architecture was greatly simplified by the high degree of network transparency which LOCUS provides and by its partition management algorithm.

* Our implementation does not provide general support for lock waiting or deadlock detection and resolution. If a lock cannot be granted after several retries, the open call fails.

4.1 Underlying Communications System

We assume that the communications system delivers messages to their proper destination and that the stream of data in the message remains unaltered. For the sake of simplicity and brevity, the description of our algorithms in this paper will assume that the communications system does not cause messages to be lost, except in the case of network partitioning, and that the communications system does not deliver duplicate messages, does not delay messages arbitrarily, and delivers messages in the order in which they were sent. In [Mueller 83] it is shown how our algorithms may handle lost, duplicated, delayed, and reordered messages if so desired.

4.2 LOCUS Partition Management

When the network is partitioned, we assume that the collection of sites making up the network is broken up into a number 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. Since this model of network partitioning differs in some respects from what may actually occur, LOCUS employs a partition management algorithm [Walker 83] which enforces the partition model just described.

Whenever any sites join or leave the network, a *topology change procedure* is run. Each site maintains a table of those sites with which it can communicate, called the *site table*. This table is managed by the topology change procedure and may lag behind the actual physical state of the network while topology changes are in progress. The topology change procedure is responsible for maintaining 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 task of the topology change procedure is to invoke *recovery,* which performs reconciliation of replicated data objects.

4.3 Terminology and 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 from a transaction's Tid both the home site of the transaction as well as the Tids of all the transaction's superiors. Processes are uniquely identified in the network by a *process unique identifier* (Pid), from which we assume it is possible to determine the site on which the process is executing.* Processes running as part of a transaction may fork, giving rise to transactions which have more than one member process. Since processes may fork remotely as well as locally, 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.* To simplify the description of our algorithms, we will assume that all processes making up a transaction reside at the same site. Later in the paper we describe the extensions which are necessary in order to handle remote member 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, Status, Pid, Members, Files]

Status = Oneof[UNDEFINED, COMMITTED, ABORTED]
Members = List[Struct[Pid, Subtrans]]
Files = List[Struct[Filename, Site, Mode]]
Subtrans = Oneof[NULL, Tid]
Mode = Oneof[READ, WRITE]
```

A transaction has status 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 the transaction and 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 includes an entry for each process making up the transaction. Each entry consists of the Pid of the process and any active subtransaction of the process. 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 name of the file and a site number 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.

4.4 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]. In this section, we first discuss the protocols for file operations, and then describe the locking and file recovery algorithms in more detail.

---

* We are speaking here of mechanisms internal to the system implementation, where location information is essential. The *application* interface is nevertheless transparent.

4.4.1 File Protocols

For each file which is open from a transaction, one of the sites storing the file in a given partition 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 site of a transaction accessing a file is called the *using site* (US).

When a transaction process invokes the *open* system call,** a message is sent to the TSS for the file.*** Upon receiving the message, the TSS makes a locking decision and takes appropriate actions. 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. It is assumed that a transaction closes all its files before committing or aborting.

* 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. Thus when a file is opened by a transaction, a TSS is assigned and this site plays the role of both CSS and SS until the file is no longer involved with transactions. 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 do not actually send a message. Instead, we directly invoke the appropriate routine.

4.4.2 Locking and State Restoration

The TSS maintains the locking and recovery information for a file involved with a transaction in a volatile data structure called a *t-lock.* A t-lock consists of a list of held and retained locking transactions and file state restoration information for write locks:

```
TLock = Struct[CurrentFileState, Holders, ReadRetainers,
                WriteRetainers]

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

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. This information is kept in the WriteHolder and WriteRetainers fields of the t-lock. The WriteRetainers field is a *version stack,* i.e., a stack of file versions with an entry for each transaction retaining a write lock.*

We now discuss how t-locks are used to manage locking information and perform file state restoration. When a transaction T wishes to open a file F, the following *TssOpen* algorithm is performed:

1. If a t-lock does not exist for F, one is created. CurrentFileState is initialized from the state of F maintained in non-volatile storage.

2. If the open request is for modification, the request is denied if any other transaction holds a lock or there is a retainer of a lock that is not an ancestor of T. If the request is granted, a WriteHolder entry for T containing a copy of CurrentFileState is entered.

3. Otherwise if the open request is for read access, the request 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 T. If the request is granted, an

* We do not require each entry in the version stack to be a complete version of the file. For each entry, enough information is required to be able 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 mechanism is very fast and inexpensive. No I/O is required; little more than an in-core file descriptor copy is involved. For more details see [Mueller 83].

entry for T is added to ReadHolders.

When a transaction T closes a file, the following *TssClose* algorithm is performed:

1. If T holds a read lock, it is removed from ReadHolders. T is added to ReadRetainers, unless an entry for T is already present in ReadRetainers or WriteRetainers.

2. Otherwise if T holds a write lock, an entry for T containing the FileState in the WriteHolder entry is pushed onto WriteRetainers, unless an entry for T is already on top of WriteRetainers.* An entry for T which may be present in ReadRetainers is removed. The WriteHolder entry is removed.**

When a transaction T commits, the *TssCommit* algorithm is performed for each participant file of T:

1. If T is a subtransaction and T retains a read lock, it is removed from ReadRetainers. T's parent is added to ReadRetainers, unless an entry for T's parent is already present in ReadRetainers or WriteRetainers.

2. Otherwise if T is a subtransaction and T retains a write lock, the entry for T on top of WriteRetainers is changed to be an entry for T's parent, unless an entry for T's parent is already present in WriteRetainers in which case the entry for T is merely popped.

3. Otherwise if T is a top-level transaction and T retains a read lock, it is removed from ReadRetainers. If ReadRetainers is empty, the t-lock structure for this file is removed.

4. Otherwise if T is a top-level transaction and T retains a write lock, the entry for T on top of WriteRetainers is popped.

* This would be the case if T had previously opened and closed the file for modification or if a committed child of T had directly or indirectly modified the file.

** Note that this algorithm supports two-phase locking since, when a transaction closes a file that was open for modification, the transaction keeps a retained lock. No other transaction that is not an inferior can access the file until the transaction commits or aborts.

When a transaction T aborts, the *TssAbort* algorithm is performed for each participant file of T:

1. If T retains a read lock, it is removed from ReadRetainers.

2. Otherwise if T retains a write lock, CurrentFileState is restored from the entry on top of WriteRetainers, and the entry is popped.

3. If there are no remaining read or write retainers, the t-lock structure for the file is removed.

As an example, suppose transaction T_1 has invoked transaction T_2, and that both transactions have modified a file F, as shown in Figure 1. 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 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 2a. If T_2 instead aborts, the version for T_2, i.e., F_1, which is on top of the version stack is popped and replaces the current version as shown in Figure 2b.

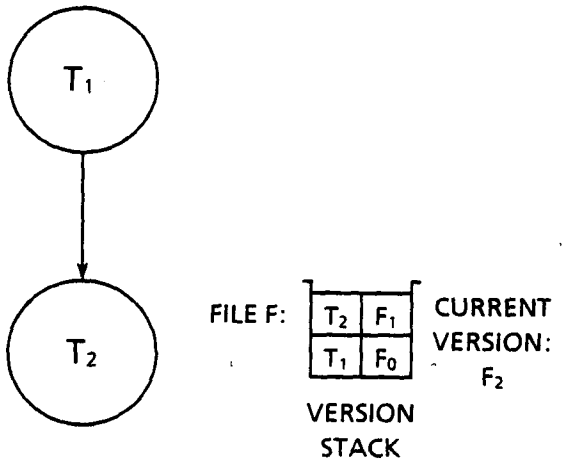

Figure 1

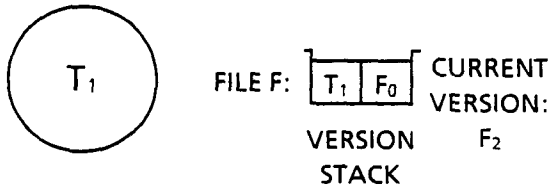

Figure 2a

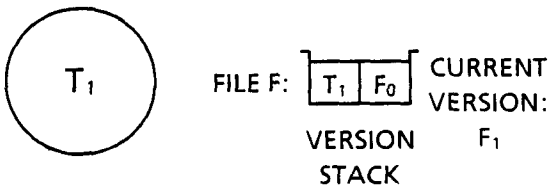

Figure 2b

Recall that a transaction may lock a file in the current partition even if there are copies of the file in other partitions. Given this policy, there is the possibility that when partitions merge there will be more than one TSS maintaining t-lock information for a particular file. This situation is called a *t-lock conflict*. Methods for handling such conflicts are discussed elsewhere; see [Edwards 82], and [Rudisin 80]. Essentially, the two conflicting t-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.

4.5 Transaction Invocation and Completion

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. The following algorithm is employed:

1. At the site of the calling process P, a Tid T_1 is generated for the new transaction.

2. If P is running on behalf of a transaction T_0, T_1 is entered in the Subtrans field in the entry for P in T_0's member process list.

3. If the user (or context mechanism) wishes the transaction to be invoked at a remote site, it is necessary to pass to the remote site the name of the load module to be executed, the command-line arguments, the Tid of the new transaction, and the Pid of the calling process.*

4. At the home site for the new transaction T_1, a transaction structure is created and the fields are filled in appropriately.

5. Execution of a new transaction process is begun.

6. The completion of T_1 is awaited by calling process P.

Whenever a transaction process performs a fork operation, an entry for the newly created process is added to the transaction's member process list. Each time a member process of a transaction completes, its entry is removed from T's member process list. 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. We will return to the details of committing and aborting a transaction in the following sections. After the called transaction has committed or aborted, the following is performed:

1. If T_1 is a subtransaction, the Subtrans field in the entry for calling process P in its transaction's member process list is reset to NULL.

2. Control is returned to calling process P.

4.6 Transaction Committing

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

4.6.1 Subtransaction Commit

In order to commit a subtransaction, the TssCommit algorithm must be performed for each file in the subtransaction's participant file list. The *subtransaction commit* algorithm is as follows:

1. At the home site of the committing subtransaction T, which we refer to as *the child,* if the TSSs for all participant files are not accessible to the child, T is aborted (to be described) and this algorithm exited.

2. A REQCOMMIT message containing T's participant file list is sent to T's parent.

* The details of forking a remote process are dealt with elsewhere [Jagau 82].

3. The home site of the parent transaction, which we refer to as *the parent,* adds the files in the received message to its participant file list.* A GRTCOMMIT message is sent to the child.

4. The child receiving the message** sends TSSCOMMIT messages to each of the TSSs for the participant files. Each message contains a list of files for which the TssCommit algorithm must be performed.

5. Each TSS receiving the message performs the TssCommit algorithm for each file and returns a RTSSCOMMIT response message along with a success code.

6. If the child receives a RTSSCOMMIT message from all TSSs and all TSSs have succeeded, T has successfully been committed and a SUBCOMMIT message is sent to the parent.

7. Otherwise if the child has been partitioned from a TSS or a site was unsuccessful at performing the algorithm, a SUBCMTFAIL message is sent to the parent.

8. T's transaction structure is removed by the child.

9. If the parent receives the SUBCMTFAIL message or if it is partitioned away from the child, the parent must abort itself in order to recover properly. Otherwise, a SUBCOMMIT message is received.

The reasons it is possible to recover in case of commit failure simply by aborting the parent of the subtransaction are as follows. 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 for which the TssCommit algorithm has been performed. Thus the only transactions which may view some of the subtransaction's committed files are those which will be aborted.

* 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.

** If the child is partitioned away from the parent before receiving the GRTCOMMIT message, the orphan removal algorithm to be described properly aborts the child.

Second, it is possible to recover the files' t-locks properly. If the TssCommit algorithm was not completed for a particular file, 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 WriteRetainers entry for the subtransaction that was attempting to commit, or in an entry for the parent if the parent modified the file. If, on the other hand, the TssCommit algorithm was completed, then either 1) the version to restore was given to the parent if the parent did not already have an entry in WriteRetainers, or 2) the parent already had an entry in WriteRetainers. In both cases, we have the proper version to restore. Our TssAbort algorithm must be enhanced to handle commit failure of course; how to do this will be shown in the section on handling network partitions.

4.6.2 Top-Level Transaction Commit

Top-level transaction commit is accomplished as follows. TSSCOMMIT messages are sent to each of the TSSs for the participant files. This will cause all files that were opened only for read by the entire transaction to have their t-lock structure released at the TSS (unless transactions which are outside this entire transaction also retain or hold read locks). If any TssCommit fails or a TSS is partitioned away, the top-level transaction must be aborted, and this is accomplished by sending TSSABORT messages to each of the TSSs for participant files.* If the entire transaction has modified a particular file, then after the top-level TssCommit, WriteRetainers will be empty and CurrentFileState is the file state that we wish to commit to non-volatile storage. We invoke a distributed two-phase commit protocol to accomplish these updates atomically and then remove the transaction structure. The participant file list minus the files that were only opened for read becomes the participant list for the two-phase commit protocol. The two-phase commit protocol used for committing a top-level transaction in LOCUS is described in detail in [Moore 82a] and is summarized in [Moore 82b]. The TssCommit algorithm may be incorporated into the first phase of the two-phase commit protocol, but this is ignored here for simplicity. During the second phase of the protocol, the t-lock structure for the file is removed.

Once the first phase of the protocol is complete, a TOPCOMMIT message may be sent to the site of the calling process. If the protocol fails or if the top-level transaction is aborted for some other reason, a TOPABORT message is sent to the site

* 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 in non-volatile storage. The TssAbort algorithm must be able to handle this case.

of the calling process.

4.7 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 T and each of its running descendants, the following *transaction abort* algorithm is employed:

1. Each of T's member processes is destroyed.

2. A FORCEABT message is sent to the sites of each of T's running child transactions and RFORCEABT responses are awaited.

3. After all responses have been received, a TSSABORT message is sent to each site having a participant file of the aborting transaction, in order to perform the TssAbort algorithm,* and RTSSABORT responses are awaited.

4. A child receiving a FORCEABT message in turn follows this abort algorithm, destroying *its* member processes, aborting *its* child subtransactions by sending FORCEABT messages and waiting for responses, sending out TSSABORT messages and waiting for responses, and finally returning a RFORCEABT response.

5. A SUBABORT or TOPABORT message, depending on whether this is a subtransaction or a top-level transaction, is sent to the site of the invoking process.

6. T's transaction structure is removed.

In the absence of partitions, this algorithm will abort all descendants of the aborting transaction and cause the TssAbort algorithm to be invoked in the proper order for each file. Handling network partitions in this case will be discussed in detail in the next section.

5. HANDLING NETWORK PARTITIONS

This section extends our algorithms to cope fully 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 a transaction home site.

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 performs the TssAbort algorithm on its participant files, some of those files may be locked by the inaccessible child and its inferiors. Thus we must modify our TssAbort algorithm to close any files which are open from inferiors and then effectively perform the old TssAbort algorithm for all descendants of the aborting transaction. In addition, since any of the inferiors of such an inaccessible child may also attempt to abort themselves, the TssAbort algorithm must be idempotent. This is required in case the aborting transaction completes the TssAbort algorithm after which an inferior invokes the algorithm. The TssAbort algorithm must also handle the case in which subtransaction commit fails and it is necessary to abort the parent of the subtransaction that was being committed. In this case the aborting transaction may not necessarily be on the stack.

Our revised *TssAbort* algorithm for a transaction T is as follows:

1. For each element of ReadHolders having T as a superior, the TssClose algorithm is invoked.

2. The TssClose algorithm is invoked if there is a WriteHolder.

3. Any entries in ReadRetainers for a transaction having T as an ancestor are removed.

4. Entries are popped from WriteRetainers until it is empty or the top element is for a superior of T. The CurrentFileState is restored from the last entry popped.

5. If there are no remaining read or write retainers, the t-lock structure for this file is removed.

Note in addition that we must now handle read, write, and close messages from inferiors effectively aborted by this algorithm. Since each such message contains the Tid of the requesting transaction, read and write messages may be denied, and close message ignored, by first checking if the requesting transaction holds a lock on the file.

* Recall that this algorithm, which is invoked for each local and remote file, aborts the file updates, clears locks, and performs other cleanup.

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 effectively perform the TssAbort algorithm on any files for which they hold or retain locks.

The orphan removal algorithm is driven by both the transaction home sites and the TSSs. As part of the topology change procedure at a site S, the following *transaction-site-driven orphan removal* algorithm is invoked for each transaction T whose home site is S:

* If a superior of T is inaccessible to S, a *silent abort* of T is performed. This consists of destroying all of the transaction's member processes and removing the transaction structure, thus aborting the transaction without performing the TssAbort algorithm on participant files or forcing child subtransactions to abort.

The abort of an inferior transaction of T is effected when the topology change procedure detects the same condition for the inferior, i.e., one of its superiors is inaccessible.

As part of the topology change procedure at a site S, for each t-lock at S, the following *file-site-driven orphan removal* algorithm is invoked:

1. For each element of ReadHolders having an ancestor inaccessible to S, the TssClose algorithm is invoked.

2. If the WriteHolder has an ancestor inaccessible to S, the TssClose algorithm is invoked.

3. For each element of ReadRetainers having an ancestor which is inaccessible to S, the entry is removed from the list.

4. If WriteRetainers is not empty, the entry bottommost in the stack which is for a transaction that is inaccessible to S is located. Entries from the stack are popped until a superior of the located inaccessible transaction is on top or the stack is empty. CurrentFileState is restored from the last entry popped.

5. The t-lock is removed if there are no remaining write or read retainers or holders.

The orphan removal strategy is correct because if a transaction is inaccessible to the TSS, then the transaction will be aborted since one of its files is inaccessible, as will be described in the following section. If one of the transaction's superiors is inaccessible to the TSS but the transaction *is* accessible, then a superior of the transaction must be inaccessible to the transaction and so the transaction will silently abort.

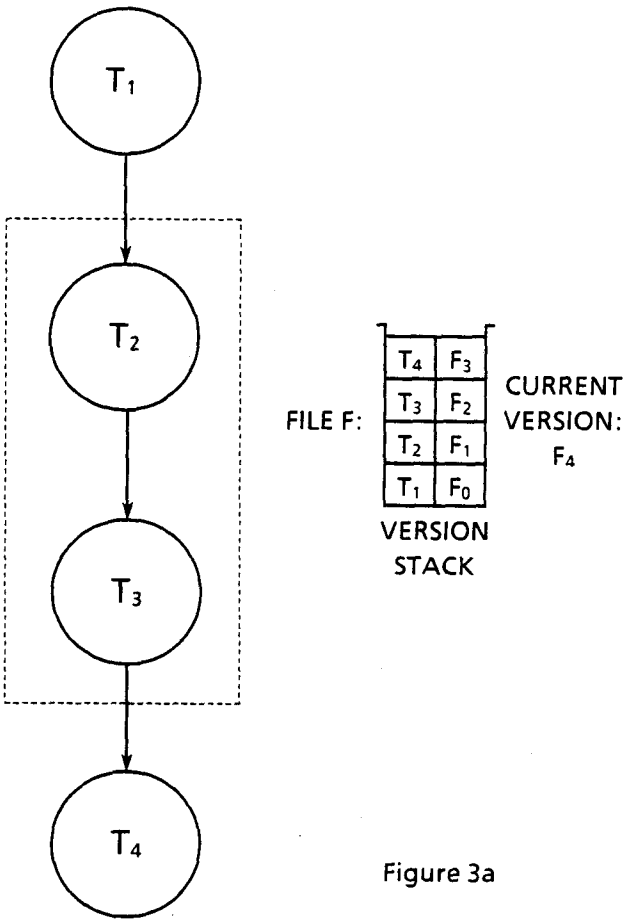

Figure 3a

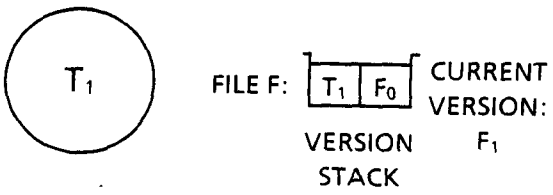

Figure 3b

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 3a. 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 freed 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 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 perform the TssAbort algorithm for T_2. Now the only remaining retainer of a write lock for the file is T_1, as shown in Figure 3b.

Note that this orphan removal algorithm does not interact with the two-phase commit protocol used to commit a top-level transaction. Before two-phase commit begins, participant files are only locked by the top-level transaction. Thus orphan removal is not concerned with these files. However, other files may be locked by aborted orphans of the top-level transaction. These files are handled by orphan removal.

If orphan removal 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. A process which called a top-level transaction also receives a special completion code, however it is unknown whether the transaction committed or aborted.*

In the scheme that we have described, there are two outstanding issues 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 TSS. 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.

The second problem is that when a transaction wishes to commit and the TssCommit algorithm is invoked, it may encounter locks that are held or retained by aborted inferior transactions whose aborts have not yet completed. This situation can be handled simply by effectively performing the TssAbort algorithm for any inferiors of the committing transaction before performing the usual TssCommit algorithm. 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. Thus our revised *TssCommit* algorithm is:

1. Steps 1 and 2 of the revised TssAbort algorithm are invoked to close any files which are open from inferiors.

2. Any entries in ReadRetainers having T as a superior are removed.

3. Entries are popped from WriteRetainers until it is empty or the top element is for a transaction which is an ancestor of T.

4. Continue with Step 1 of the original TssCommit algorithm.

* 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.

5.3 Inaccessible Storage Sites

Whenever a site becomes partitioned away, all transactions having a participant file for which that site is the TSS will be unable to commit. Thus our topology change procedure aborts a transaction if any of its participant files is inaccessible. This action, while correct, is inefficient. Since some of the transactions having inaccessible files have superiors who also have inaccessible files, many simultaneous aborts will be performed, generating unnecessary processing and network traffic. For example, suppose transaction T_1 invoked T_2 which invoked T_3, and all three transactions have file F as a participant. If F is separated from T_1, T_2, and T_3, we will abort T_1, T_2, and T_3, when we actually need only abort T_1.* Thus what we would like to do is to abort simply the *topmost* transaction involved with the inaccessible file. A method is presented in [Mueller 83] for determining the topmost involved transaction.

6. REMOTE MEMBER PROCESSES

In order to extend our algorithms to allow remote member processes, we use the transaction home site as a centralized coordinator for all the transaction's processes. In this way, we limit the impact on other parts of our algorithm. The required extensions are as follows. First, a message exchange is required between the remote site and the transaction home site whenever an action is initiated at the remote site which calls for the transaction's member process list or participant file list to be updated. Second, the algorithm for aborting a transaction must be modified to send messages to destroy any remote member processes. Third, the topology change procedure must be modified to destroy a remote member process which becomes partitioned away from its home site, and abort a transaction having any remote member process which is inaccessible to the home site. Last, our t-lock structure must be extended to contain a list of USs for each transaction holding a read or write lock, and the topology change procedure must be extended to invoke the TssAbort algorithm for any file having an inaccessible US. Details of remote member process management may be found in [Mueller 83].

* We cannot perform a silent abort of the transactions upon detecting an inaccessible TSS, because this condition does not cause all transactions in a branch to be aborted. It only causes those with inaccessible TSSs to be aborted.

7. 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. The implementation consists of 7208 lines of C code,* which is a little more than twice that required to implement simple nested transactions [Moore 82a] [Moore 82b]. Preliminary performance results shown in Appendix B 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 reliability gained is well worth the added cost.

Future work includes completing remote member process support, taking extensive performance measurements, and incorporating appropriate optimizations. 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

[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.

[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.

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

[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.

[Gray 81a] 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", Computing Surveys, Vol. 13, No. 12., June 1981, pp. 223-242.

[Gray 81b] 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.

[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.

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

[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.

[Moore 82a] Moore, Johanna 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, UCLA Center for Experimental Computer Science, October 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.

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

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

[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 83] Popek, Gerald J., and Walker, Bruce J., "Network Transparency and its Limits in a Distributed Operating System", unpublished paper, UCLA Center for Experimental Computer Science, 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.

[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.

[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.

[Svob 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.

[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: Example of Invoking Concurrent Subtransactions

The following transaction program fragment written in the C language 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 program, 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 complete 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. A preprocessor may be used to provide a more natural syntax for such an operation, but is not considered in this paper. The general problem of a distributed programming language incorporating nested transactions and abstract datatypes is being dealt with in work on the Argus language [Liskov 82].

Appendix B: Preliminary 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. 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 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. 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.

Extensive measurements of the two-phase commit protocol in our system are given in [Moore 82a]. There it is reported that for a moderate amount of files, say 6, the two-phase commit protocol is worse than simple file closing by a factor of 4 in the local case, and 2.1 in the remote case.

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 processing performed by the entire transaction increases, transactions are not that much more expensive than non-transaction programs.

ELAPSED TIME (in seconds) *All Files Local*			
Number of Files	Non-Transaction	Top-Level Transaction	Subtransaction
0	.233	.183	.216
1	.350	.783	.267
2	.450	.916	.334
4	.700	1.233	.483
6	.966	1.483	.566
8	1.183	1.700	.733
10	1.384	2.100	.783

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
1	.550	1.050	.533
2	.917	1.450	.833
4	1.650	2.267	1.467
6	2.350	3.016	2.067
8	3.000	3.833	2.717
10	3.700	4.616	3.334

Table 2: Elapsed Time -- All Files Remote

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.

[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.

[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.

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

[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.

[Gray 81a] 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", Computing Surveys, Vol. 13, No. 12., June 1981, pp. 223-242.

[Gray 81b] 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.

[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.

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

[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.

[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.

[Moore 82a] Moore, Johanna 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, UCLA Center for Experimental Computer Science, October 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.

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

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

[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 83] Popek, Gerald J., and Walker, Bruce J., "Network Transparency and its Limits in a Distributed Operating System", unpublished paper, UCLA Center for Experimental Computer Science, 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.

[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.

[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.

[Svob 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.

[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.