Transactions and Synchronization in a Distributed Operating System*

Matthew J. Weinstein, Thomas W. Page, Jr., Brian K. Livezey, and Gerald J. Popek

University of California, Los Angeles

Abstract

A fully distributed operating system transaction facility with fine-grain record level synchronization is described. Multiple member processes, remote resource access, dynamic process migration, and orderly interaction with concurrent non-transaction activities are all supported. An unusual logging strategy, based on shadow pages but supporting logical level locking, is used. This choice is justified on the basis of ease of implementation and performance analysis.

The design and implementation is done in the context of Locus, a high performance distributed Unix operating system for local area networks.

1 Introduction

It has often been observed that the construction of distributed software in a multi-machine environment can be substantially more difficult than the development of single machine software. Two principal causes are the frequent difference in interfaces between local and remote resources, and the richer set of failure and error modes in the multi-machine case. *Network transparency*, which makes local and remote resource interfaces logically identical, addresses the first issue. It is a well known concept, and degrees of transparency are embodied in a number of network and multiprocessor operating systems [Bartlett78] [Wecker80] [Locus84].

Addressing the second issue requires that additional tools be available to the software developer. Flexible and general support for *transactions*, for example, would be quite useful. Numerous proposals for transactions have been made, both in operating systems and programming languages [Bartlett81] [Mueller83] [Liskov82] [Moss82].

Here a full design and implementation of transactions in a distributed Unix operating system is presented. We assume an environment composed of a substantial number of relatively small machines, with considerable communications bandwidth among them, performing database-oriented operations. In order to perform effectively in comparison to large centralized systems, such systems rely on achieving considerable concurrency of data access and update; hence, fine-grain synchronization is important. Further, since it is considered important to permit workstation class machines to include these tools, a compact transaction implementation is necessary.

The transaction and locking facilities described in this paper utilize several pre-existing mechanisms in the underlying Locus[1] distributed operating system [Popek81] [Walker83], currently operational on both mainframes and workstations. Some of these features were found invaluable in permitting rapid implementation of the work reported here. The base Locus system provides support for atomic file updates via an intentions list mechanism; this mechanism provides the underlying support for the current transaction implementation, supporting both the transaction logging and data file commit mechanisms. Lightweight network protocols provide the basis for high-performance distributed locking, data page transport, and process migration services. The distributed, transparent namespace provided by Locus, enabled the implementors to ignore many difficult problems of distributed file handling and transaction design.

This paper describes the functionality and implementation of the transaction and synchronization facilities provided.

* This research has been supported by the Advanced Research Projects Agency under contract DSS-MDA-903-82-C-0189.

[1] Hereafter, "Locus" refers to the Locus system as extended to include transactions and record level locking. The base system to which these facilities were added contained no transaction mechanism.

Section 2 provides an overview of Locus transaction semantics. Section 3 discusses transaction operation and synchronization. Section 4 describes the implementation of the Locus transaction mechanism. Section 5 discusses the implementation of the distributed record-level locking mechanism. Section 6 presents performance measurements and observations. A comparison with related work and conclusions follow.

2 Transaction Semantics

The transaction facility in Locus provides the usual assurance of serializability of transactions with one another, and atomic behavior in the face of network and nodal failures. It is desired that, from a user and program point of view, the transaction facilities be entirely transparent. In the Locus context, this means that a process in a transaction should be able to migrate around the network during execution, create transaction subprocesses either locally or remotely and interact with them, and access resources transparently, independent of their location. All of these characteristics should operate with high performance, and behave reasonably in the face of failures. While these goals impose significant requirements on a design and implementation, they increase the facility's utility.

The Locus implementation of transactions described here does not allow general nesting of transactions. Previous implementations of transaction mechanisms that have allowed full- nested transactions have been rather expensive, both in terms of implementation and performance [Mueller83]. Version stacks and version trees, the mechanisms for intra-transaction synchronization, and other bookkeeping overhead are unnecessary when full-nested transactions are avoided. Instead, transactions in Locus are *simple-nested*.

The semantics of simple-nested transactions are straightforward. A non-transaction process starts a transaction by issuing a *BeginTrans* call. This call encapsulates all subsequent file operations by that process, and its sub-processes, within that single transaction. Encapsulation ends upon execution of a corresponding *EndTrans* call, which makes the actions of the transaction permanent; or an *AbortTrans* call, which undoes the transaction's actions. Only those resources locked within the *BeginTrans-EndTrans* pair are considered part of that transaction. Resources locked before the start of the transaction may be used within the transaction but are not committed or aborted along with the transaction, as described in section 3.3.

In order to permit the composition of existing transaction code, the transaction implementation pairs *BeginTrans-EndTrans* calls. Each process contains a count of the current transaction nesting level. When a *BeginTrans* operation is encountered, the level of transaction nesting increases; an *EndTrans* decreases the nesting level. When the nesting level returns to 0, via an *EndTrans* call, the transaction has completed successfully.

As an example of the usefulness of transaction nesting, consider an application program which, within a transaction, calls on a database subsystem to perform an update. This database subsystem may call *BeginTrans* and *EndTrans* internally, to cause its critical section to execute atomically. This kind of nested call must be permitted, and in fact may occur unintentionally. Clearly, the *EndTrans* call generated by the database subsystem must not terminate the entire transaction.

3 Synchronization Functionality

There are a number of issues which must be addressed in transaction synchronization. The following section discusses the solutions implemented in Locus to such issues as locking schedule and lock granularity, system interface, and the interaction of transactions and non-transactions accessing shared data.

3.1 Locking Schedule and Granularity

The Locus transaction facility provides record-level locking, as well as a transaction envelope. Locks may be acquired explicitly (via a system request) or implicitly (at the time of record access).

The locking schedule employed here is the conventional two-phase locking discipline, as described by [Eswaran 76]; all processes created from within a transaction are part of that transaction (for synchronization purposes), independent of their location in the network. If a process, while executing as a transaction, creates a child process, and either of them locks a record for exclusive access, the other may do so as well. This policy is consistent with Unix process semantics, where child processes inherit file access from their parents, with an identical set of access rights.

Any resource acquired by a process which is part of a transaction is locked, either explicitly before access, or implicitly when accessed, as required by the two-phase locking discipline. A transaction may choose to explicitly unlock a resource after use; the lock is *retained*, and may later be reacquired by any process within the transaction; unlocked resources are not made available to processes outside the transaction until the transaction commits or aborts.

The locking facility may be used by any process, whether or not the process is part of a transaction. Alternately, a non-transaction process may choose not to lock resources before use, in the conventional Unix manner; thus, compatibility with conventional Unix sharing is preserved, while at the same time it is possible to synchronize access as required by transaction clients.

	Unix	Shared	Exclusive
Unix	r/w	read	no
Shared	read	read	no
Exclusive	no	no	no

Figure 1: Transaction Synchronization Rules

The compatibility rules for the various locking modes provided by the transaction mechanism are shown in Figure 1.

Note that Locus record locks are enforced, not advisory, as in the /usr/group standard, or Berkeley 4.2 Unix. Enforced locks are necessary to ensure that the two-phase locking discipline is maintained. Because enforced locks can introduce integrity problems (in the form of access denial), the current policy requires that a process have write access to a file in order to issue locking requests for that file.

Another ramification of enforced locks is the possibility of deadlock. The Locus kernel does not detect deadlock. Instead, an interface to operating system data is provided, permitting a system process to detect deadlock by constructing a wait-for graph, using conventional techniques [Coffman 71]. In this manner, a variety of deadlock resolution and redo strategies may be implemented.

3.2 Record Locking Interface

The locking mechanism permits byte granularity synchronization of access to data files, in a manner quite similar to common record locking proposals for Unix. Enforced locks are used to ensure that the two-phase locking protocol is satisfied.

Before record locking can take place, the file to be locked must be *opened*. In Locus, this resource name-mapping (performed by the *open* call) is separate from resource locking. Thus, a program may perform name mapping, a relatively expensive operation in a distributed system, once, then lock and unlock records within the file, without repeatedly incurring the expense of name resolution.

After a file has been opened, ranges of bytes in that file may be locked in several modes, including shared read or exclusive read/write. Locked ranges may be extended or contracted, and locking modes may be upgraded or downgraded, through subsequent locking requests. Locking a range of bytes is accomplished by positioning the current file pointer to the first byte to be locked, and issuing the lock request:

Lock(file,length,mode)

The *file* argument refers to the channel number returned by the *open* call, *length* specifies the number of bytes to lock, and *mode* indicates whether the requested lock is a shared lock request, an exclusive locking request, or an unlock request. If a lock request conflicts with an existing lock, the requestor will receive an indication of the conflict, or alternatively will be queued until the lock can be granted.

The ability to extend a file and simultaneously lock the newly allocated area is useful when updating shared log files[2]. This operation is performed by placing the file to be updated in *append* mode. Future lock requests are interpreted as being relative to the end of file, so processes can lock and extend a file atomically.

[2] Without such a mechanism, remote processes attempting to extend a heavily used log could repeatedly be intercepted between the time the end of a file was located, and the time a lock was placed on the file, and thus suffer from livelock.

3.3 Interaction With Non-Transaction Programs

Implementing transactions in a general purpose operating system presents many advantages: efficacy of a single implementation, and the ability to easily compose multiple independently prepared programs into a single transaction, for example. However, one must also deal with the reality that clients of the transaction facility will not always wish to be executing in transaction mode, and that there will be non-transaction programs present in the network.

These issues arise immediately when one constructs applications such as database management systems. Database managers execute queries as transactions, but may simultaneously require access to resources on a non-transaction basis. Accounting and performance records are obvious examples. Consequently, when a transaction begins, one must consider how to treat resources such as open files already acquired by the process, and data which has been modified but not yet committed.

Consider the following programs, which access related data items in a file:

```
open x;                          open x;
  .                              writelock x[1];
  .                              x[1] := C;
  .                              unlock x[1];
BeginTrans;                        .
  readlock x[1];                   .
  t := x[1];                       .
  writelock x[2];                  .
  x[2] := t;                       .
EndTrans;                          .
                                 abort x[1];
```

Figure 2: Serialization Problem

In the fragments shown, a non-transaction part of one program updates record x[1] and then unlocks it without committing the record; this is possible because the synchronization mechanism cannot require a non-transaction to retain locks. Subsequently, another program starts a transaction, which writes x[2] based on the uncommitted value read from x[1]. Unfortunately, the transaction mechanism does not guarantee that record x[1] will not be rolled back or lost due to a system crash; the result in this case is that x[2] is not equal to x[1], potentially violating consistency constraints.

It is desirable to guarantee that transaction serializability is preserved regardless of what non-transactions do. The above example shows a correctly composed transaction that is made non-serializable by the actions of an unrelated program. Locus solves this problem by enforcing the following rules when applying a lock to a file or record:

1. Any lock obtained by a transaction is retained until the transaction commits or aborts. This is a standard two-phase locking requirement.

2. If a modified but uncommitted record is locked by a transaction, in *any* mode, the lock is retained until the

transaction commits or aborts; the record is committed upon successful transaction completion regardless of whether it was modified by the transaction. This allows access to unstable data without violating two-phase locking.

By imposing these requirements, updates survive, atomic behavior can be assured, and transactions cannot unintentionally be made non-serializable by interactions with non-transactions.

3.4 Exceptions to Serializability

There are instances in which a transaction must have the ability to read or update records without those actions becoming part of that transaction [Stonebraker 81]. Critical data structures such as system catalogs for a database, or directories in a filesystem, should not remain locked for the duration of a transaction. There are also some actions which should be explicitly visible during transaction execution. Concurrent execution of two transactions which attempt to create a file by the same name is one example. One transaction must fail even though neither has reached a commit point that would make their updates visible. Allowing transactions to *selectively* violate two-phase locking is a functionality which must be provided.

There are two ways in which a process is permitted to intentionally violate the default two-phase locking discipline. The first method is through an additional locking mode called a *non-transaction lock*. A non-transaction lock obeys the same locking rules as locks acquired by a transaction, as shown in Figure 1; however, the two-phase locking protocol is not enforced on these locks by the transaction mechanism.

The second way to violate serializability is to acquire a lock before a process becomes a transaction; these locks are *not* converted to transaction locks at the *BeginTrans* point. This approach is clearly less general than the first, because the locks that will be required may not be known before the start of the transaction.

In both of these cases, the system does not force the lock to be retained until the outcome of the transaction is determined; this can improve concurrency in some circumstances, at the cost of potentially lost serializability. As always, careful design of algorithms is necessary.

4 Transaction Implementation

Transactions were surprisingly straightforward to implement in the transparent distributed environment provided by Locus. A number of mechanisms within the underlying Locus kernel were utilized, including the shadow-page based file system, distributed name-mapping services, light-weight network message protocols, and support for network failure detection.

The basic Locus operating system contains a single-file commit mechanism which is implemented by intentions lists, and is used as part of normal filesystem operation. The list consists of a set of page pointers for the file; in Unix that list is contained in the file's descriptor block (inode), although there may be indirection present. Files are committed by forcing dirty file pages to disk, and atomically overwriting the inode on disk with new data, freeing up the old data pages. Little additional I/O over conventional (unsafe) Unix filesystem behavior results, so this is the default operating mode.

The two-phase transaction commit mechanism was easily built, using the record-level shadow-paging facility as a base. Logs built by the transaction mechanism consist of sets of *intentions lists*, as well as associated locking information. The transaction mechanism relies only on the functionality of the record commit mechanism, and not on the specific implementation. The intentions list mechanism used to commit records within individual files could be replaced with a logging mechanism, without affecting the multi-file transaction commit facility.

Salient aspects of the transaction mechanism are outlined below. The first section describes transaction initiation, followed by descriptions of transaction commit, abort, and system failure recovery. The underlying shadow page mechanism for committing variable length records within files is described in detail in section 5.

4.1 Initiation and Operation

A transaction is initiated when a *BeginTrans* call is issued by a non-transaction process. This causes the generation of a temporally unique identifier, which names the newly formed transaction. This identifier is used internally to identify files and records accessed by the transaction. All processes created as part of the transaction inherit this transaction identifier.

For each process within a transaction, the kernel maintains a *file-list*, which enumerates all files used by that process. This file-list is used by the two-phase commit protocol, to assure that all files used by a transaction are correctly committed. Up to that point, the list is kept in a decentralized fashion, at the same site as the process to which they refer, in order to allow efficient access. If a process migrates to another site in the network, its file-list migrates as well.

At the beginning of the two-phase commit protocol, the entire file-list for a transaction must be known in order to determine which files will participate in the commit process. To accomplish this, as each child process completes, its file-list merges with that of the top-level process of the transaction[3]. When all child processes in a transaction have completed, the file-list for the top-level process of the transaction contains all of the files which have been accessed by the entire transaction.

[3] The child may be running on a different site from the top level process, consequently requiring that its file-list be sent to another site in a network message. Similarly, the top level process may have migrated to another network node, with its current file-list.

The protocols must guarantee that the file-list eventually gets to the correct site, even if the top-level process migrates several times. There is a potential race condition in which a process begins migrating to a new site, and simultaneously a list of files sent from one of its child processes arrives. If the parent process's file-list has already migrated to the new site, but the process itself has not completed migration, the child process's file-list would not be recorded.

This race condition is avoided by marking the migrating top-level processes with an *in-transit* indication. When a message containing a list of files arrives, the system verifies that the target process still resides at that site, and is not in the process of migrating. If the process is no longer at that site, or is already migrating, the system returns a failure message to the child's site, which must retry the operation. However, if the top-level process is resident at that site, the system locks the process from migrating, for a short duration, until the operation has been completed; since processes migrate infrequently, this does not adversely impact process performance[4].

4.2 Transaction Commit

When a top-level process reaches the transaction endpoint, and all of its subprocesses have completed their processing, transaction commit begins. Transaction commit is accomplished using the two-phase commit protocol [Gray78] [Lindsay79]. The top-level process's current site becomes the commit *coordinator site*. The coordinator site directs the transaction commit process, acting as coordinator in the two-phase commit protocol, and maintaining the transaction log. The list of all files used in the transaction is accessible to the coordinator site, and is used to direct transaction commit. The storage sites of each of these files are required to be *participant sites* in the two-phase commit protocol.

Transaction commit is performed in several steps, involving three levels of logs. The first log written is the transaction *coordinator log*. This log contains the transaction identifier of the transaction being committed, a list of all files containing records which were used by the transaction, along with their corresponding storage sites, and a status marker, indicating the outcome of the transaction (initially *unknown*). After successfully writing this log, the coordinator sends *prepare* messages to each of the participant sites.

The second level of logging takes place at the participant sites. Upon receiving a prepare message, each participant site flushes modified records and writes its *prepare log*, storing enough of the intentions lists and lock lists for each file to guarantee that the files can be committed when the transaction reaches the second phase of the commit protocol, regardless of local failures. When these logs have been stored, each participant site replies to the coordinator site with a *prepare completed* message. Upon receipt of all prepare completed messages, the coordinator changes the status marker in its log to *committed*; this determines the transaction commit point.

The third level of logs used in the commit protocol are the per-file shadow pages stored at the participant sites. After the transaction commit point, a kernel process at the coordinator site asynchronously sends *transaction commit* messages to each of the participant sites. The participant sites complete the update of the files involved in the transaction by using the single-file commit mechanism, and releasing all corresponding retained locks.

4.3 Transaction Abort

When any process within a transaction fails, or issues an *AbortTrans* call, the entire transaction must abort. The transaction mechanism aborts all changes to files made by the top-level process, and all those made by member processes which had completed[5]. Transaction abort is initiated by sending an abort message to the site at which the top-level process of the transaction resides. The system first discards any changes made to files by the top-level process, then locates all children of the top-level process, and sends abort messages to each of them. The children roll back their modified records, release all locks, and signal their children in turn. In this way, the abort cascades down the process tree.

If a transaction is being aborted because a member process has been lost due to failure of the node on which it has been executing, its open files will be closed and changes aborted by the underlying system protocols when they detect the failure. Similarly, when an active storage site crashes, if the open files on it had not yet become involved in two phase commit, they will be aborted upon system restart.

When the transaction mechanism is informed of a change in the topology of the network (e.g. a site crashes or becomes inaccessible), it aborts all ongoing transactions involving sites no longer in the current partition. Failures that occur before a site has prepared to commit are treated as aborts.

Once a transaction has entered two-phase commit processing, it consists of only a single top-level process. In this case, transaction abort is accomplished by changing the status of the transaction in the coordinator log to *aborted*, and sending abort messages to the participant sites, which are responsible for rolling back their own files.

4.4 Transaction Recovery

The transaction recovery mechanism deals with several types of failures. The failure of a communications link may separate one or more of the sites involved in a transaction from the rest of the transaction. The coordinator site or any participant site may crash. Any software module of the transaction

[4] There are other circumstances involving process migration which can result in race conditions. This technique makes process migration appear to be an atomic operation.

[5] Aborts consists simply of discarding the shadow pages and intentions lists unless other processes have record locks on the same pages. The case of aborting updates to pages with multiple locks is discussed in the section 5.

may fail or issue an *AbortTrans* call. Once two-phase commit has begun, it is the responsibility of the transaction coordinator site to recover from failures.

When a site reboots after a crash, before transactions are permitted to run, the transaction recovery mechanism is started. To ensure correct transaction recovery, coordinator logs are retained until all commit or abort processing has successfully completed for the corresponding transaction.

The recovery mechanism examines each existing coordinator log at its site. If the transaction in the log had not reached its commit point, or was marked as aborted, it is queued for abort processing. If the transaction in the log had reached the commit point (indicated by the presence of a commit mark in the log), the transaction is queued for the second phase of the two-phase commit protocol. During this process, the transaction recovery mechanism may send duplicate commit or abort messages to other sites. However, since transaction-id's are temporally unique, duplicate commit or abort messages cannot produce unintentional failures.

In a transparent distributed filesystem, files may be stored at any site in the network and may be transported on removable media. Therefore, it is important to assure that logs are stored on the same medium as the files to which they refer; otherwise, logs might not be present at the time that recovery actions are required. If this were to happen, it would not be possible to decide which data pages were to be freed and which were to be kept, since pointers to such pages are in the log file. Hence, the Locus transaction mechanism maintains a separate log per logical volume (filesystem).

5 Record Locking Implementation

In Locus, record locking facilities are available to both transaction and non-transaction processes. There are marked differences in the treatment of locks granted to transaction and non-transaction processes, however.

When a non-transaction is granted a lock, it need not obey the two-phase locking protocol; thus, data may be modified, and locks relinquished, without committing or aborting the modified data. These uncommitted changes are generally visible and may be used and committed by any transaction or non-transaction (within policy constraints).

When a transaction is granted a lock, however, the two-phase locking protocol must be followed, in order to preserve serializability. This requires locks to be *retained* until transaction commit.

5.1 Locking Implementation

The implementation of record locking is reasonably straightforward. When a file is opened, a copy of the file descriptor (*inode*) is brought into kernel memory at the file's storage site. As lock requests are processed, a list of lock descriptors indicating process-id of the process holding the lock, the locking mode, and the range of bytes locked are attached to the file descriptor; if the process is part of a transaction, the transaction identifier is also placed in the lock structure. The lock list structure is shown in figure 3.

When a request to lock a record in a file is issued, that request is processed at the file's storage site. If the requestor is not at the storage site, the lock list will not be locally available; a light-weight network message is sent to the storage site, and a response awaited. At the storage site, the kernel examines the list of existing locks for the file and, if the lock is compatible with the existing locks on the file, an entry for that lock is added to the lock list. A success response is returned to the requesting process. Otherwise, a failure response is returned, which may be used to notify the requesting process, or may initiate queueing for a later locking attempt.

When a requesting site receives a successful response to a locking request, it caches this response in its local lock list. This permits the kernel to quickly validate each process's

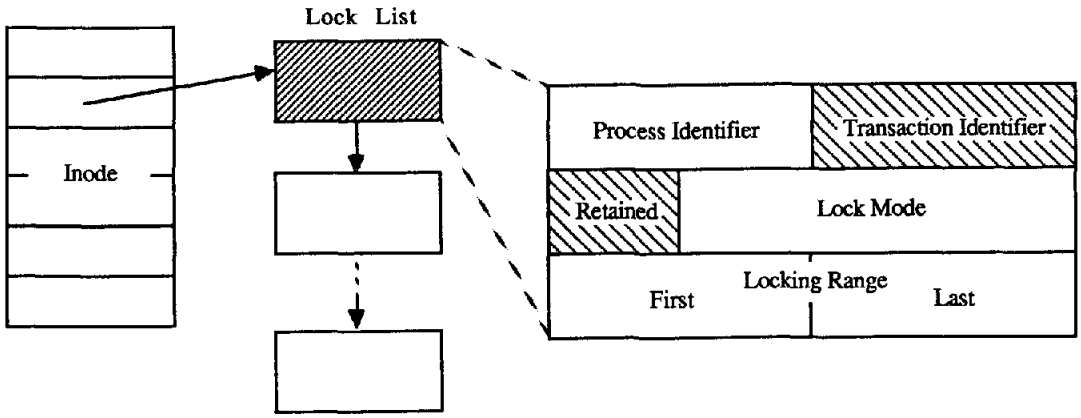

Figure 3: Lock List Structure

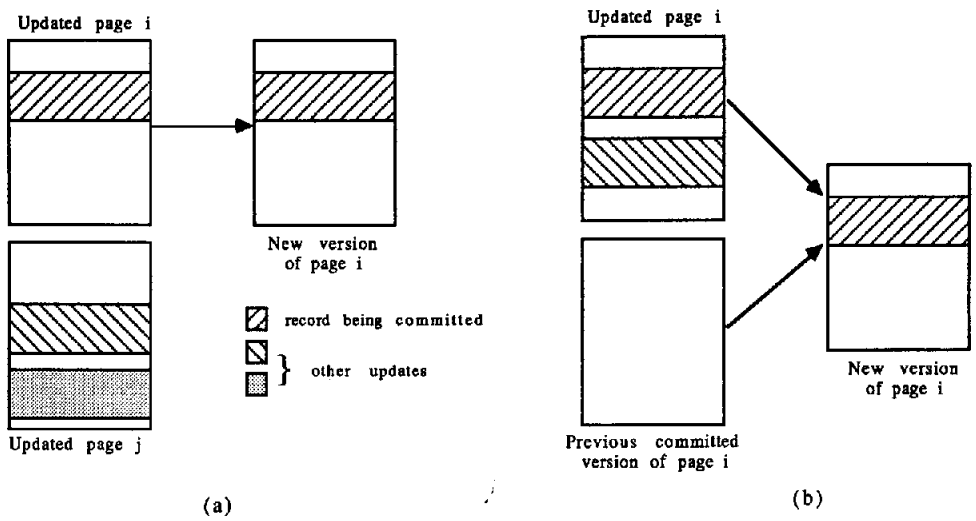

Figure 4: Record Commit Mechanism

read and write requests.

5.2 Record Commits and Aborts

Record-level commits and aborts are performed using a modified intentions-list mechanism which logs changes to data at the physical (page) level, rather than at the logical (record) level. However, it is desirable to allow more than one process to modify disjoint (logical) records on a *single* (physical) page to improve performance, and more importantly, reduce the possibility of false deadlock.

Even though unrelated updates are a statistically infrequent occurence, the commit mechanism must be able to deal with them correctly. In Locus, the standard intentions-list strategy is augmented by a *page differencing* method, to provide correct record-level commit operation. Differential update is not required in the statistically dominant case, where records to be committed reside on pages that contain no updates by other transactions or processes. In this case, it is not necessary to examine the data contained on each page, and the pages can be written to non-volatile storage directly. This case is shown in Figure 4(a).

When disjoint records on the same physical page *have* been modified by more than one transaction or process, the commit mechanism cannot directly commit the modified page, since other updates would incorrectly be committed[6]. To perform the commit, a copy of the *previous version* of the page is re-read from non-volatile storage, the record(s) of interest are transferred to that page, and this new (merged) page is written back to non-volatile storage[7]. This operation is shown in Figure 4(b).

Record abort works similarly to record commit. When an abort occurs, each file page is examined; if there are no conflicting modifications on the page, that page is directly rolled back to its old version. However, if conflicting modification have occured, these cannot be rolled back. In this case, the old version of the data page is re-read, and the record(s) to be aborted are overwritten by their original contents.

The implementation discussion up to this point has ignored the possibility that files of interest might be replicated, as supported by the Locus kernel; this situation can be handled quite gracefully. When record locking is requested for a file, the file is internally marked as being *open for update*. Under normal conditions, when a file has been opened for update, Locus uses a single storage site strategy for updates. Thus, even though there may be a number of storage sites able to process open requests for a file, only one site at a time is designated as the primary update site[8]. The record lock list is kept

[6] Records written on the same physical page by different transactions *must* be disjoint, due to the enforcement of mutually exclusive writes by the policy mechanism.

[7] Clearly, re-reading and differencing two pages is somewhat expensive. Fortunately, this case occurs infrequently, so re-reading pages is rarely necessary. To improve performance, clean copies of frequently used pages could be kept in the buffer pool; the buffer-aging algorithm can be adjusted as necessary.

[8] In order for this strategy to work, it is also necessary to deal with the case where a file may be open for read by multiple processes at different sites. These processes are typically served by the closest available storage site. Therefore, when an open for update (or record locking) occurs, storage site service must be *migrated* to the primary update site. The mechanisms to accomplish this were already present in Locus.

at this primary site, and everything operates as previously described.

There are further opportunities for performance optimization in the distributed environment. When a lock is requested, the page(s) containing the byte range can be *prefetched*, in anticipation of their subsequent use. Another optimization would allow the storage site to *temporarily* transfer its ability to manage a group of locks to another site. This could reduce overhead in the event that a group of co-located processes were making heavy use of locking facilities. Control of these locks, and current locking information, would migrate if the locking patterns changed.

6 Performance Considerations

Logging mechanisms are generally viewed as superior to intentions list strategies, due both to fewer I/O's required for commit, and because physical contiguity of data files is maintained [Stonebraker]. However, some investigators have indicated that the methods are competitive, and for many usage characteristics, intentions lists may not generate substantially higher I/O costs than logging.

A discussion of performance considerations in logging techniques is complex, involving issues of disk scheduling and contention, physical contiguity and allocation schemes, and client usage patterns. A brief analysis of the relative costs of shadow paging and commit logs, based on assumptions regarding file access in a distributed environment, was made in [Weinstein85]. This analysis, which modeled the costs of typical file operations via an operation counting method, indicates that the relative performance of shadow paging and commit log mechanisms is highly dependent on the nature of the access strings generated by applications. In addition, results reported in [Elhardt84] and [Kent85] indicate that paging mechanisms can potentially outperform logging mechanisms with physical page placement controls and hardware support.

Although logging may significantly outperform shadow paging in some circumstances, for many combinations of record size and placement, implementations of shadow paging can provide comparable performance.

6.1 Transaction Performance

Locus uses structured logging to accomplish transaction atomicity. Writing to these logs entails overhead which affects the performance of the transaction mechanism. Below, we enumerate the additional I/O operations that the transaction mechanism requires.

A simple transaction which updates a single page of a single file requires three I/O operations beyond normal file activity (see Figure 5). First, the operating system writes the transaction structure, containing the list of resources participating in the transaction, to the coordinator log. Then, in order to prepare the updated file for committing, it flushes the modified data page to non-volatile storage. Next, it must write the intentions list for the modified file to the prepare log. After these three writes, the file is prepared to commit. The transaction commit occurs when the commit mark is subsequently written to the coordinator log, a fourth I/O. These writes occur before the transaction completes. Some time later, the kernel process which performs the second phase of two-phase commit must perform one more write to actually replace data pointers on disk with the intentions-list contents [9].

An overhead of three I/O's per transaction is significant when the work of performed by that transaction is minimal (e.g. involves only a single update). However, real transactions commonly generate a substantial I/O load, so that the commit overhead in our design is small by comparison. Also, in the common case when records on multiple pages in a single file are updated in one transaction, no additional overhead results for the additional records. Only the intrinsically necessary I/O (step 2 in Figure 5) is repeated.

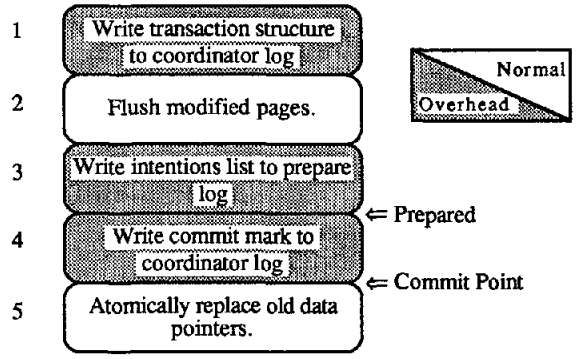

Figure 5: Transaction I/O Overhead

Committing multi-file transactions incurs slightly more overhead. In particular, since it is desirable that each disk be able to be recovered independently, there is one prepare log per media device. Consequently, step 3 in Figure 5 must be repeated for each logical volume containing modified records[10].

6.2 Record Locking Performance

In order to assess locking performance and to measure the performance penalty incurred for non-local operation, the performance of the record locking mechanism was measured with all processes and files located at the same site, as well as with files and processes distributed at different sites. These measurements were conducted on a network of Vax 11/750

[9] The current transaction implementation does not achieve simple transaction commit in five I/O operations. Locus currently requires two writes to add an entry to a log instead of one; one for the log's data page and one for its inode. Hence steps 1 and 3 in Figure 5 each require two I/O operations. This problem is being corrected.

[10] Our current implementation uses one prepare log per file per transaction rather than one per medium or filesystem.

computers connected by a 10Mb Ethernet with Interlan network interfaces.

When a record locking request for a file is issued by a process running at that file's active storage site, the record locking request is processed completely at that site. Processing these locks involves referencing and updating local kernel data structures, as well as some system call overhead.

Local record locking performance was measured using standard timing functions present in the Locus kernel. The measurements were obtained by repeatedly locking ascending groups of bytes in a file. Excluding system call overhead, the cost of obtaining a single lock is approximately 750 instructions (1.5 ms) per lock. This cost is small compared to the cost of initiating a data page read, or sending a network message. The time required to obtain a lock is also a fraction of typical disk I/O latencies, and is even less significant when compared to the time involved in requesting remote disk pages. These results indicate that setting and releasing record locks is a relatively a low cost operation in this environment.

Locking records in a file that is stored at a remote site is somewhat more complex and introduces additional costs. At the requesting site, a message is sent to obtain the appropriate lock from the storage site. The storage site must process the lock request, verify that the lock is allowable under the current policy, and return a reply. In this process, a variety of network-related latencies are introduced.

As noted above, when requestor and data storage site are co-located, performance is quite good, a side-effect of centralization of locking information at the storage site. In the remote case, locking costs were found to be indistinguishable from inherent round-trip message exchange costs. Measurements indicated that locking latencies had increased from approximately 2 ms in the local case, to about 18 ms per lock in the remote case. These delays were comparable to expected network delays, which are a function of the underlying communication network, and could be reduced further in other environments.

6.3 Record Commit Performance

The record commit mechanism underpins the operation of the distributed record locking mechanism. This mechanism provides the ability to selectively update individual bytes on a data page during a commit operation by efficiently *differencing* each data page with its shadow, when necessary, before performing the commit operation.

The cost of a commit operation on a set of records has been measured in two cases: when updates from different users overlap on the same data page, and when they do not. The results of several trials measuring performance differences indicate that there is only a moderate additional cost inherent in this mechanism. Figure 6 shows the results of the measurements; service time reflects the amount of CPU consumed by the record commit operation at the local site and latency reflects the elapsed time. These results are relatively insensitive to the number of overlapping records on the page[11].

	Local Commits	
	Service time	Latency
Non-overlap	21 ms (9450 inst)	73 ms
Overlap	24 ms (10800 inst)	100 ms

	Remote Commits	
	Service time	Latency
Non-overlap	16 ms (7200 inst)	131 ms
Overlap	16 ms (7200 inst)	124 ms

Figure 6: Measured Commit Performance

Note that commit costs are divided between the local site and the storage site when a remote file is committed. The results shown for remote commits in figure 6 indicate the overhead measured at the requesting site. In this case, the costs decrease slightly at the requesting site, due to computation and page updates being offloaded to the storage site. Even so, network delays significantly increase the latency of remote commits.

The commit differencing algorithm compares the original and modified copies of each appropriate data page. In the performance measurements reported above, all necessary pages were in buffers (due to the LRU buffer replacement algorithm employed). If the data page were not available in a buffer, a read operation would have been necessary to retrieve that page. This case occurs very infrequently, as commits typically occur shortly after an update, and so this is not reflected in the results reported above.

6.4 Performance Summary

The above measurements show that reasonable performance can be achieved in a distributed record locking mechanism for both local and remote operation. This performance resulted from the use of lightweight communication protocols, a primary site locking mechanism, and local lock caches.

We have also shown that it is possible to build a reasonably performing record commit facility based on an intentions list mechanism. This mechanism properly manages the case of multiple updates on the same data page; the overall cost of copying the overlapping bytes from page to page does not substantially impact performance. This is a useful result, since we feel that constructing an intentions list mechanism is simpler than constructing an equivalent logging mechanism in some systems.

[11] In these measurements, 1k byte pages were used. An increase to 4k byte pages would add approximately 1 ms to the measured results, in the case where a substantial portion of the page were copied.

7 Related Work

Transaction mechanisms have been implemented or are being implemented in several research and production systems. These systems differ in several ways. Perhaps most significantly, there are transaction implementations in programming languages, database systems, and operating systems. These related efforts also differ with respect to functionality, method, granularity of concurrency control, performance, and degree of implementation. We review some of these projects and show how their work relates to ours.

7.1 Previous Locus Work

An implementation of nested transactions was operational in the Locus testing environment at UCLA in 1983 [Mueller83] [Moore82]. This facility was a process-based transaction mechanism, which permitted any process to create a new process which would be run (atomically) as a transaction. The transaction could itself create a sub-process to invoke a module as a subtransaction, which would also act atomically. The current work on transactions and synchronization in Locus is not based upon this previous implementation for several reasons.

The creation of a new Unix-style heavy-weight process for each transaction was judged to be too expensive for a transaction processing facility. The *BeginTrans* and *EndTrans* interface eliminates this requirement and makes operating system level transactions a reasonable tool for constructing database or other transaction processing systems.

The version stacks and intra-transaction synchronization required for nested transactions were found to be expensive. The primary advantage of the fully-nested transaction mechanism is that less work is lost in the case of a failure. However, in an optimistic scenario (where failures do not occur frequently), it makes more sense to optimize the more common case where subtransactions complete successfully.

The previous Locus transaction work performed locking at the file level. Whole file locking restricts the degree of concurrent access to data files, and is not a satisfactory base on which to implement a database system. The new transaction facility provides record-level locking. Also, the previous design and implementation did not permit transactions to be composed of processes at different sites, and did not support the ability of processes to migrate among sites.

7.2 Tandem ENCOMPASS*

ENCOMPASS is a distributed database management system designed to provide high reliability through replication at both the hardware and software levels. A node in the Tandem system consists of two to sixteen processors. Each process may have a twin backup process on another processor within the node. Similarly, each disk is connected to two controllers and has a mirror volume. The underlying message-based operating system provides transparent access to remote resources.

The ENCOMPASS Transaction Monitoring Facility (TMF) provides single level transactions [Borr81]. As in Locus, transactions may include processes at many sites. For each disk volume, there is a *DiskProcess* pair through which all accesses to that volume must pass. The DiskProcess implements an undo/redo logging scheme. Each node has a process pair known as the Transaction Monitor process (TMP). The messages implementing the two-phase commit protocol are sent from TMP to TMP.

Tandem provides its transaction mechanism in a database management system. Consequently ENCOMPASS transactions are not as general purpose a programming tool as are Locus transactions. However, Tandem provides one of the few commercially available distributed transaction facilities.

7.3 TABS

TABS [Spector 84] provides distributed transaction services for processes running under the Accent kernel [Rashid 81]. The Accent system kernel is a message-based operating system that provides support for an extensible abstract typing mechanism. TABS is implemented primarily as a collection of Accent type managers, as well as a small number of kernel enhancements.

The TABS transaction approach differs substantially from that of Locus as it provides type-oriented transaction support. In TABS, each type manager provides the locking and commit mechanisms for the types it implements. TABS uses a log-based recovery strategy, which aids in the implementation of type-specific recovery schemes. Also, TABS communicates with client processes via message-passing mechanisms, rather than via kernel calls.

7.4 Argus

Argus [Liskov 84] [Moss 82] provides support for distributed nested transactions using a combination of language-level and kernel-based mechanisms. The Argus programming language provides access to the transaction mechanisms which are implemented in the Argus kernel. *Atomic* abstract data types, implemented by *guardians*, are manipulated by language-provided transaction control primitives. Access to objects controlled by guardians is made through *handlers*; each handler invocation causes a process to be created within the guardian, which runs as a nested subtransaction of the requestor's transaction.

Run-time support for transaction control is provided by the Argus kernel. As in Locus, substantial support for enhanced process scheduling, message passing, and other services has been built into the kernel. However, the Argus approach provides these services only to Argus language applications.

As of this writing, a centralized implementation of Argus has been completed, and a distributed version is in progress.

*ENCOMPASS is a trademark of Tandem Computers.

7.5 R*

Like Tandem's ENCOMPASS, IBM San Jose's R* Distributed Database Manager [Daniels 82] [Lindsay 84] provides a distributed transaction facility within a database system. Unlike the Tandem distributed transaction architecture in which remote service is obtained by communicating with remote servers, or Argus which creates a process for each request, R* creates a new process for each user session.

R* uses secure virtual circuits and datagrams for remote access; a virtual circuit is established between an R* process at the requesting site and a server process running R* at the serving site. A remote R* process, serving an initial request, may require service from yet another site. In this case, a new virtual circuit is set up between the server process and a newly created server process at the third site. These connections are reused, so the overhead of setting up these virtual circuits is amortized over an entire R* session.

R* uses the two-phase commit protocol to ensure that all actions are either commited or aborted. However, because an R* transaction can constitute a tree of processes, the commit protocol follows this model: at each level of the tree, when a process receives a *prepare to commit* message, it propagates the message to all of its subordinate processes, and collects *prepared* messages for eventual return to its parent. This differs from Locus, where all but the top-level process have completed when the transaction commits. In Locus, the exchange of messages is between the kernels at the coordinator site, and the kernels at all participant sites; this protocol involves less latency.

8 Conclusions

We have found that it is reasonably straightforward to design and implement fully transparent, fine-grain transaction facilities in a distributed operating system. These facilities are universally available to applications software. In doing so it was surprising that a shadow page mechanism could be so readily adapted to record-level synchronization. Approximately one man-year of effort was required to accomplish this.

Nevertheless, the sophistication required to correctly construct the facilities reported here makes it clear that it is best to construct them once, and provide them in a generally available manner to applications builders. As distributed environments are increasingly composed, at least in part, of significant numbers of relatively modest power workstations, the need to build reliable distributed applications software is certain to increase.

References

[Bartlett78] J. Bartlett, "A NonStop Operating System," *Eleventh Hawaii International Conference on System Sciences*, Honolulu, HA, January 1978, pp. 103-117.

[Bartlett81] J. Bartlett, "A NonStop Kernel," *Proceedings of the Eighth Symposium of Operating Systems Prinicples*, Pacific Grove, CA, December 1981.

[Borr81] A.J. Borr, "Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing," *Proceedings of Very Large Database Conference*, Cannes, France, 1981, pp. 155-165.

[Coffman71] E.G. Coffman, Jr., M.J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys*, 3(2), June 1971, pp. 67-78.

[Daniels82] D. Daniels, P. Selinger, L. Haas, B. Lindsay, C. Mohan, A. Walker, and P. Wilms, "An introduction to distributed query compilation in R*," *Proceedings Second International Symposium on Distributed Databases*, Berlin, September 1-3, 1982. Also IBM Research Report RJ3497, San Jose, CA, June 1982.

[Elhardt84] K. Elhardt, R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Transactions on Database Systems*, Vol. 9, No. 4, December 1984, pp. 503-525

[Eswaran76] K.P. Eswaran, J. Gray, R.A. Lorie, and I.L. Traiger, "The notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.

[Gray81] J. Gray, "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.

[Gray78] J. Gray, "Notes on Data Base Operating Systems," Operating Systems An Advanced Course, *Lecture Notes in Computer Science 60*, Springer-Verlag, 1978, pp. 393-481.

[Kent85] J. Kent, H. Garcia-Molina, J. Chung, "An Experimental Evaluation of Crash Recovery Mechanisms," *Symposium on Principles of Database Systems*, Portland, OR, March 1985.

[Lampson79] B.W. Lampson and H.E. Sturgis, "Crash Recovery in a Distributed Data Storage System," XEROX Palo Alto Research Center, April 1979.

[Lindsay79] B.G. Lindsay, P. Selinger, C. Galtieri, J. Gray, R. Lorie, T. Price, F. Putzolu, I. Traiger, and B. Wade, "Notes on Distributed Databases," IBM Research Report RJ2571(33471), IBM Research Laboratory, San Jose, CA, July 14, 1979, pp. 44-50.

[Lindsay84] B.G. Lindsay et. al., "Computation and Communication in R*: A Distributed Database Manager," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 24-38.

[Liskov82] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*, Albuquerque, NM, 1982.

[Liskov84] B. Liskov, "Overview of the Argus Language and System," Programming Methodology Group Memo 40, Laboratory for Computer Science, M.I.T., 1984.

[Locus84] "The Locus Distributed System Architecture", Edition 3.1, Locus Computing Corporation Technical report, June 1984.

[Moore82] J.D. Moore, "Simple Nested Transactions in LOCUS: A Distributed Operating System," Master's Thesis, Computer Science Department, University of California, Los Angeles, 1982.

[Moss82] J. Eliot B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, M.I.T., 1981.

[Mueller83] E. Mueller, J. Moore, G. Popek, "A Nested Transaction Mechanism for LOCUS," *Proceedings of the Ninth Symposium on Operating Systems Principles*, Bretton Woods, NH, October 10-13, 1983.

[Popek81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Theil, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium of Operating Systems Principles*, Pacific Grove, CA, December 1981.

[Spector84] A. Spector, J. Butcher, D. Daniels, D. Duchamp, J. Eppinger, C. Fineman, A. Heddaya, P. Schwarz, "Support for Distributed Transactions in the TABS Prototype," Technical Report CMU-CS-84-132, Computer Science Deparment, Carnegie Mellon University, 1984.

[Stonebraker81] M. R. Stonebraker, "Operating System Support for Database Management," Communications of the ACM, Vol. 24, No. 7, July 1981, pp 412-418

[Walker83] B. Walker, G. Popek, R. English, C. Kline, and G. Theil, "The LOCUS Distributed Operating System," *Proceedings of the Ninth Symposium on Operating Systems Principles*, Bretton Woods, NH, October 10-13, 1983.

[Wecker80] S. Wecker, "DNA: the Digital Network Architecture," *IEEE Transactions on Communications*, vol. 28, April 1980, pp. 510-526.

[Weinstein85] M.J. Weinstein, "Some Performance Aspects of Shadow Paging Mechanisms," Locus Memorandum #21, Department of Computer Science, University of California, Los Angeles, 1985.