An Introduction to Nested Transactions

J. Eliot B. Moss

COINS Technical Report 86–41
September 1986

Submitted to ACM Transactions on Database Systems

The concept of atomic transactions has proved to be useful in thinking about and implementing concurrency control and recovery management in reliable multi-user systems operating on shared data. Nested transactions enhance concurrency and recovery semantics by providing more composable, finer grained control. Here is offered a gentle introduction to nested transactions, including presentations of concurrency control (in terms of locking), deadlock detection and avoidance, and recovery (in terms of shadow copies). The concepts extend to other concurrency control and recovery methods, such as timestamps and logging, though details are not included. While they have uses in centralized systems, nested transactions are especially helpful in distributed systems. To illustrate this, some simple distributed applications are sketched, as well as techniques for implementing nested transaction in distributed systems.

1 Overview

What are nested transactions? What are they good for? How might they be implemented? These are some of the questions I attempt to answer in this introduction. You should be comfortable with database transaction processing notions such as two-phase commit, two-phase locking, etc. If not, the references can be used to gain such background.

Nested transactions are an extension and enhancement of atomic transactions. Hence, nested transactions will be presented largely by contrasting them with atomic transactions. To start on familiar ground, particularly simple locking and recovery methods are described. This concrete explanation is followed by examples to help bring the ideas home. Next, the concepts are generalized, pointing the way to more realistic schemes. At that point you should have a good feeling for how nested transactions might work in a centralized system. I then state some ways in which nested transactions are a valuable improvement over atomic transactions.

Once nested transactions for centralized systems have been treated, I expand the discussion to distributed systems. First, I describe a way of embedding nested transactions in a distributed system. After this, I describe a nested transaction management algorithm for distributed systems. Then, through the use of some particularly cogent examples (remote procedure call and update of replicated data) I argue the special significance of nested transactions to reliable distributed software.

Deadlock control for nested transactions is treated next. I include algorithms for distributed deadlock detection and resolution, as well as deadlock avoidance for nested transactions. Finally, I comment on related research.

2 Nested Transactions in Centralized Systems

After a brief review of the idea of an **atomic transaction**, I introduce nested transactions and some terminology. Then concurrency control for nested transactions is described, followed by some examples of locking. Next I consider a simple recovery technique for nested transactions, which also has an example. Then I consider how the locking and recovery techniques introduced might be generalized.

2.1 Atomic Transactions

For purposes of this discussion, a **transaction** is a related set of **actions** that accept inputs, modify system state, and produce outputs. The traditional transaction consists of a single thread of execution, having a definite beginning and end. For the discussion to be interesting, transactions must be processed concurrently (multiprogrammed or multipro-

cessed), and there must be some overlap in the parts of the system state that they access. Further, it is assumed that transactions may fail, for a variety of reasons including: bad inputs, hardware failure, deadlock, etc.

Atomic transactions guarantee to "hide" the effects of concurrent processing and failure. First, an atomic transaction is run in its entirety or not at all. Never is a transaction left only partly executed. This property is called **failure atomicity**. Second, concurrently executed transactions are guaranteed not to interfere with each other so as to produce inconsistent or invalid results. Lack of interference can most easily be guaranteed by promising that the system *appear* to have run transactions one at a time, each to its entirety. This property is called **serializability** and it provides **concurrency atomicity**. A third guarantee of interest is **permanence**: that the effects of completed transactions not disappear as a result of later failures. I will not treat permanence in detail, because nested transactions do not affect it.

2.2 Nesting Transactions

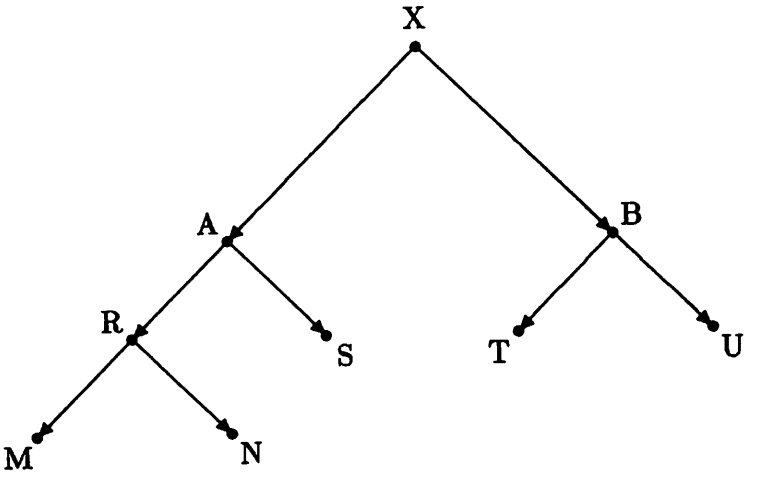
An **atomic transaction** groups together a sequence of **primitive actions** (such as reads, writes, inputs, outputs) and guarantees that the package is executed as a whole (or not at all) without interference from other transactions or events in the system. A **nested transaction** consists of *either* a group of primitive actions *or* a group of nested transactions — it is a recursive concept.

The following analogy may be useful, and holds up fairly well. If primitive actions are like the instructions of a computer, then an atomic transaction is like a block of code: a grouping together of related instructions intended to achieve a particular goal. Nested transactions are analogous to the generalization of such blocks of code to subroutines. It is even conceivable that nested transactions could be recursive. This analogy will find later application when I discuss remote procedure calls.

While the analogy with subroutines gives some flavor of the control flow possibilities of nested transactions, it does not begin to explain how to nest failure atomicity and concurrency atomicity. In fact, at first the nesting of atomicity appears to be somewhere between impossible and absurd. This is because something that is atomic cannot be further split down into components, yet I am saying that there are atomic transactions nested within atomic transactions.

The paradox is resolved in the following way. When I say a transaction is atomic, all I mean is that the transaction's internal structure is not visible from outside the transaction. An atomic transaction might have a rich and complex internal structure, but when viewed from the outside (that is, when viewed from other transactions), it is an indivisible unit of

Figure 1: A Transaction Tree



X is a *root* or *top-level* transaction.
A and B are *children* of X.
A and B are *siblings*.
X is the *parent* of A and B.
R and S are children of A (and grandchildren of X).
M, N, S, T, and U are *leaf* transactions.
M's *ancestors* are X, A, R, and M.
M's *superiors* are X, A, and R.
A's *descendants* are M, N, R, S, and A.
A's *inferiors* are M, N, R, and S.
A transaction's *universe* or *environment* consists of its inferiors.

action.

Nesting fits in with this notion of atomicity without difficulty. Each nested transaction provides a universe of execution within which actions may be performed without regard to system failure or concurrent execution of other transactions. Atomic transactions provide a similar universe. The extension made by going to nested transactions is that the universe includes the create-transaction operator. Thus the protected universe of a nested transaction can contain subtransactions. Such subtransactions will also exhibit failure and concurrency atomicity within the universe provided by the containing transaction. Specifically, failure of one subtransaction does not affect others; each subtransaction is run entirely or not at all; subtransactions must be serializable with respect to each other; etc.

Nesting of transactions consists only of strict containment of a subtransaction in a containing transaction. This implies that nested transactions form hierarchies and their relationships can be represented with trees. Hence, I use tree terminology and also that of

familial relationships (parent, child, ancestor, etc.). Let me just run through some terms I use when talking about nested transactions. Figure 1 illustrates this terminology.

A group of nested transactions forms a **forest** (set of trees). A particular transaction of the traditional kind corresponds to a single tree. Each nested transaction corresponds to a node of a transaction tree. The root of the tree is called a **root** or **top level** transaction. The root may have one or more **children**. Such children claim the root as their **parent**. Similarly, the children of the root may have children of their own. This nesting can continue to arbitrary depth. All transactions except roots have parents. **Leaf** transactions are those transactions with no children.

The parent-child relationship is generalized to **ancestor-descendant**, implying *zero* or more levels between the two transactions in question. Thus, each transaction is an ancestor and descendant of itself. When I need to restrict the relationship to *one* or more levels of difference, I will use the terms **superior** and **inferior** in place of ancestor and descendant (respectively).

2.3 Concurrency Control

Let me try to sharpen your possibly vague conception of nested transactions by offering one concrete way of providing concurrency control for nested transactions. Generalizations and alternatives to this method will be discussed later.

Consider the simplest possible locking scheme: mutual exclusion locks. Concurrency control for atomic transactions via exclusion locks follows these rules:

Atomic Transaction Exclusion Locking

1. Uniformity: All transactions follow the rules.
2. Meaning of Locking: For a transaction to perform an operation, the transaction must hold the lock corresponding to that operation.
3. Exclusion: If a transaction requests a lock, the request can be granted only if no other transaction currently holds the lock.
4. Release: When a transaction completes (aborts or commits), it releases all of its locks.

The exact relationships between locks, operations, and data are unimportant for our discussion with one exception: holding of a lock must guarantee exclusive access to the data it protects.

The above scheme is called two-phase locking, because transactions have a lock acquisition phase (while they are running) and a lock releasing phase (in this case, when they

complete). A slightly more general scheme is examined in some detail in [Eswaren, et al. 76]. In that paper it is shown that two-phase locking guarantees **serializability**. An execution of a set of transactions is serializable if and only if it is equivalent to some serial (one at a time) execution of the same transactions. Loosely speaking, executions are equivalent if they produce the same results and leave the database in the same state when done. It is generally assumed that any transaction, if executed in isolation, will preserve consistency of the system state. Hence, serializability provides a promise of consistency (assuming, of course, that the system starts in a consistent state). In some cases consistency might be realized without serializability, but there is good evidence for the general appropriateness of serializability (see [Rosenkrantz, et al. 80]).

Note that different orders of execution of transactions may give different results. Serializability merely says that there is *some* equivalent serial execution. Note also that serializability is most easily understood and generally described in an "after the fact" way. That is, all transactions being considered are assumed to have completed successfully. Thus the issues of aborted transactions and deadlock are avoided by assuming they have already been resolved.

How can two-phase exclusion locking be extended from atomic transactions to nested transactions? First, I make a simplifying assumption: the only nested transactions that directly invoke operations on the system state are leaf transactions. This assumption can be made without loss of generality, as I now explain. Suppose we have a transaction T. It is desired that T perform operations directly and that T have inferior transactions. The desired effect can be achieved by providing T with an extra child transaction that performs the operations T was supposed to do directly. In this way it can be guaranteed that only leaves manipulate the system state directly, while their superior transactions play only a controlling role in transaction processing.

Here, then, are locking rules for nested transactions:

Nested Transaction Exclusion Locking

1. Uniformity: All transactions follow the rules.
2. Meaning of Locking: For a transaction to perform an operation, the transaction must hold the lock corresponding to that operation. (Only leaf transactions may perform operations, but other transactions may hold locks, as described below.)
3. Exclusion: If a transaction requests a lock, the request can be granted only if all holders of the lock (if any) are ancestors of the requesting transaction.
4. Inheritance: When a transaction succeeds (commits), its locks are given to its parent, if any, or discarded if the transaction is top level.

5. Release: When a transaction fails (aborts), its locks are released (discarded).

Some discussion may help in understanding these rules. Consider first the exclusion rule. Another way of stating this rule is that the requesting transaction must be *within* all environments currently holding the lock (if any). Recall that each transaction provides an environment for its inferiors. Only its inferiors are within that environment; all other transactions are outside of it. Obviously the exclusion rule works if the requested lock is not held at all, or is already held by the requesting transaction. If the lock is held by one or more superior transactions, then several facts together imply the correctness of the exclusion rule. First, no transaction can be performing operations controlled by the lock. This is because only leaf transactions can perform operations, and superior transactions (the only holders of the lock) are obviously not leaves. Second, while a leaf possesses a lock, no other transaction can modify the locked object. Finally, the inheritance rule prevents early release of modifications from a containing environment.

Now consider the inheritance rule. By inheritance I mean the propagation of locks from child transactions to their parents when the child transactions complete successfully. (Transaction failure is considered in more detail later.) In the universe of the parent, when a child completes, the locks held by the child are effectively released: the exclusion rule permits another inferior of the parent to obtain the lock, since the child no longer holds it (and the parent is necessarily superior to its inferiors). However, only inferiors of the parent can acquire the lock. From the point of view of transactions outside the parent's universe, the lock appears to be held by the parent. This is true from the time the lock is first acquired by the child until the parent commits.

Note that a lock can be held on several levels simultaneously. Note also the distinction between what happens to a transaction's locks when the transaction commits and when it aborts. Release of locks on abort gives better performance than inheritance: it allows more transactions to proceed, and requires fewer transactions to be aborted to resolve actual or potential deadlocks. This point should become more clear later. Additionally, it is hard to satisfy inheritance on abort when site crashes can occur in a distributed transaction system. It is just simpler to permit abort to release locks, since a crash has that effect anyway. Note that since all of a transaction's effects are assumed to be undone when the transaction aborts, no consistency problems arise from discarding the locks.

A transaction can commit even if one or more of its children aborts. This will be discussed in more detail later, but briefly, the advantage is that a transaction can retry a necessary action, use a different but satisfactory method of achieving its end, or, if the aborted actions were non-essential, commit without further corrective action. Thus the handling of locks when a transaction aborts is relevant.

One other point should be clarified. Locks are propagated up the tree, from the leaves toward the root. Lock requests are *not* sent up from the leaves to higher levels of the tree, and then granted locks propagated downwards. If locks were inherited on abort as well as commit, then the two concepts would be equivalent, but since locks are discarded on abort, they are not the same. Furthermore, since locks can be held at several levels, it is not correct to think of one lock moving up and down the tree and being possessed by different transactions at different times.

In sum, the locking rules make a lock *appear* to be held from outside any transaction holding the lock, while the lock *appears* free inside the innermost lock holder. Acquistion of a lock makes the lock look held in more environments; inheritance of a lock makes it look free in the parent's environment (and all subordinate environments). In this way holding of a lock has become a relative concept: it is relative to a transaction's position in the transaction tree.

2.4 Locking Examples

Let us see how nested transaction locking works by means of a simple example. Let there be two top level transactions A and B. Transaction A has two children, AA and AB. Transaction AA also has two children, AAA and AAB. This situation is illustrated by Figure 2a. The example uses a single lock, L. In the figures I have bracketed holders of L, and written a "+" by transactions that could hold L, and a "−" by ones that could not.

Initially, L is not held by any of the transactions. It could be acquired by any one of the leaf transactions: AAA, AAB, AB, or B. Suppose AAA requests L and is granted it. This is shown in Figure 2b. At this point, no transaction other than AAA may be granted L: AAA exclusively holds the lock. If AAA fails (aborts) L will be released (Figure 2c). However, if AAA commits, then L is inherited by AA (Figure 2d). Now AAB can be granted L, but AB and B cannot. Note the differences between Figure 2c and Figure 2d.

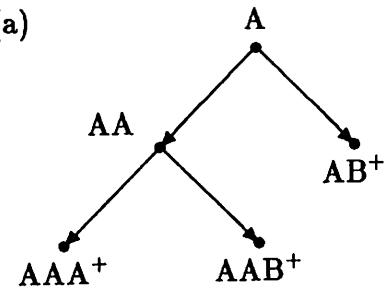
3 Recovery Management

To make the discussion clear and brief, highlighting the differences between atomic transaction and nested transactions, I present a very simple-minded recovery scheme. Other possibilities will be mentioned later. As with concurrency, I review recovery for atomic transactions. The method described is called **shadow file** or **double copy** updating. Here are the rules for atomic transactions:

Shadow Recovery for Atomic Transactions

Figure 2: Locking Example

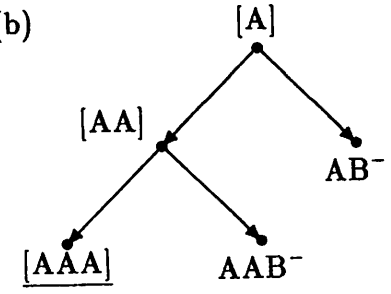
(a)



B^+

No transaction holds the lock.

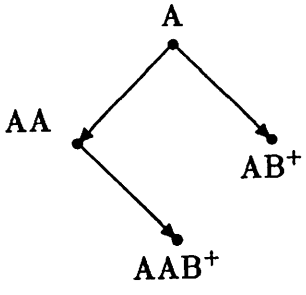
(b)



B^-

AAA acquires the lock.

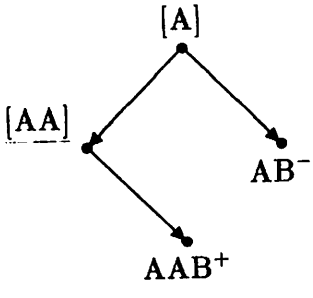
(c)



B^+

AAA aborts and the lock is discarded.

(d)



B^-

AAA commits and AA inherits the lock.

Legend:
- T^+ indicates T can acquire the lock
- T^- indicates T cannot acquire the lock
- [T] indicates T appears to hold the lock
- $\underline{[T]}$ indicates T actually holds the lock

1. When a transaction acquires a lock on some data, a copy of the data is made, and all work is performed on that copy. The copy is known as a **shadow**.
2. When a transaction succeeds (commits), its shadows are installed in favor of the original versions.
3. When a transaction fails (aborts), its shadows are discarded, and the original versions are left intact.

Provided that installation of shadows can be made to appear atomic, the above rules make transactions all-or-nothing, guaranteeing failure atomicity. Stable storage techniques [Lampson and Sturgis] provide one way of installing shadows atomically. A closely related recovery method is to operate on the originals instead of the shadows, install the shadows on abort, but leave the originals on commit. That method performs better in the commit case by reducing delay. Restoring shadows on abort is very similar to logging. Log based recovery involves writing a sequential record of all changes made, and scanning that record in case of an abort, undoing the operations performed. For more details on logging see [Gray 78]; for more about shadow methods see [Verhofstad 78].

I hope that the simplicity of the above rules makes their correctness obvious. Below are the corresponding recovery rules for nested transactions.

Shadow Recovery for Nested Transactions

1. When a transaction acquires a lock on some data, a copy of the data is made, and all work is performed on that copy. The copy is known as a shadow.
2. When a transaction succeeds (commits), its shadows are inherited by its parent, in favor of any versions the parent may hold for the same data. When a top-level transaction commits, its shadows are installed in the database.
3. When a transaction fails (aborts), its shadows are discarded, and all other versions are left intact.

The first rule may need to be clarified. The current view of the data (i.e., the data from which a copy is made) is the shadow held by the youngest (most inferior, i.e., most deeply nested) transaction holding a shadow for the data at that time; or, if no transactions hold shadows of the data, the original. This notion of current value, together with the shadow inheritance rules, insures that updates are visible within all environments where the updates have been committed. The locking rules prevent updates from being visible too widely.

Recovery for atomic transactions needs at most two copies of any data: the old and the new versions. Where the two versions reside and their representations are the concern of the various recovery mechanisms. Recovery for nested transactions may require many

copies of a data item: up to one copy for each level of nesting of transactions that modify the item, plus the original. This is because the state of the item must be available whether or not each of the transactions commits or aborts, and each environment may have a different starting state for the item (because of changes previously made in the enclosing environments).

Note that shadows are created and propagated up the transaction tree in exactly the same manner as the corresponding locks.

3.1 Recovery Example

Suppose a top level transaction T has two children T_1 and T_2, and database variable X starts with value 0. This is illustrated by Figure 3a. T_1 modifies X, setting it to 1 (Figure 3b). Note the shadow $X = 1$ next to T_1 in the drawing. T_1 then commits (Figure 3c), and T inherits the shadow. T_2 then sets X to 2 (Figure 3c). Now there are two shadows: one held by T, and one by T_2.

There are three possible outcomes of this situation:

1. T_2 commits (Figure 3d) and then T commits, leaving X set to 2. Note how T inherited T_2's shadow in the figure.
2. T_2 aborts (Figure 3e) and then T commits. X will be set to 1.
3. T aborts (it does not matter if T_2 commits or aborts). X is left with its original value, 0.

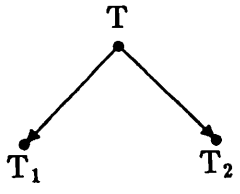
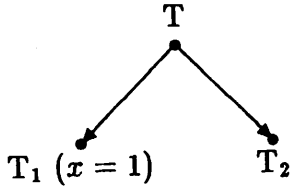
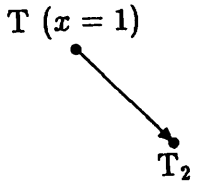
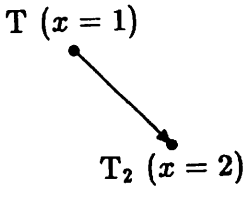
This example generalizes to as many levels as one likes. Note that a transaction will possess a shadow only if it possesses a lock on the corresponding data. Because there can be at most one lock per level of transaction tree (a consequence of the locking rules), there can be at most one shadow per level of the tree. It is not too hard to construct examples where there is exactly one shadow per level, so the bound on the number of shadows is tight.

3.2 Generalizations

Exclusion locking and shadow file recovery serve to illustrate the differences between single level atomic transactions on the one hand, and nested transactions on the other. But as pointed out before, nested transactions are a more general concept, not restricted by specific concurrency and recovery control techniques. Let us see some of the ways in which nested transactions are easily generalized.

First, exclusion locking may be extended to read-write locking quote readily. Here are the rules:

Figure 3: Recovery Example

	Database Version	Transaction Structure	Comments
(a)	$x = 0$		Initial situation.
(b)	$x = 0$		T_1 sets x to 1.
(c)	$x = 0$		T_1 commits, T inherits $x = 1$.
(d)	$x = 0$		T_2 sets x to $x + 1$.
(e)	$x = 0$	T ($x = 2$)	T_2 commits after (d).
(f)	$x = 0$	T ($x = 1$)	T_2 aborts after (d).

Nested Transaction Read-Write Locking

1. Uniformity: All transactions follow the rules.
2. Meaning of Locking: For a transaction to perform a read (non-modifying) operation, the transaction must hold either a read or write lock for the data to be accessed. For a transaction to perform a write (modifying) operation, the transaction must hold a write lock for the data to be modified.
3. Exclusion: If a transaction requests a read lock, the request can be granted only if all holders of write locks for the same data are ancestors of the requesting transaction. If a transaction requests a write lock, the request can be granted only if all holders of either read or write locks for the same data are ancestors of the requesting transaction.
4. Release: When a transaction commits, its locks are inherited by its parent (if any). When a transaction aborts, its locks are released.

We have omitted the two-phase rule since inheritance actually embodies it. Note also that being two-phase is not always sufficient for reliable concurrency control. Not only must lock holding be two-phase, but no locks can be released until the transaction has either committed or aborted. Otherwise results would be released early and there would be problems undoing transactions that aborted after releasing some locks. It would be possible to implement such a scheme only if transactions that accessed the released data could also be tracked down and aborted, etc. For a particular scheme, see [Takagi 79]. Davies [Davies 73] covers these concepts in a more general framework.

The crucial part of the extension from exclusion locks to read-write locks (also known as share and exclusion locks) is the third rule, which states the conditions under which a lock can be granted. The methods generalize to, for example, the intention locks of [Gray 78]. However, I will not present such extensions here. Note also that the locking methods I have discussed do not prevent deadlocks. Deadlocks and their resolution will be covered in detail later.

Along other lines, nested transaction concurrency control can also be done with timestamps instead of locks. Reed [Reed 78] presents such a design. His design is more general than is required, for he allows the system to keep multiple versions for (essentially read-only) reference by old transactions. Generalization of other timestamp schemes (e.g., SDD-1 [Bernstein, et al. 78]) to nested transactions has not been done.

Turning to recovery, it is easy to generalize logging to nested transactions, though the details have not been published. Checkpoint/restart should not be instrinsically more complicated for nested transactions. However, nested transactions encourage concurrency within transactions, and such concurrency might be a problem, especially in a distributed system. This is because a distributed system checkpoint/restart algorithm must guard

against the **domino effect** [Russell 80]. When one site in a system rolls back, other sites may be forced to roll back, too, in an attempt to maintain consistency across the system. However, if checkpoints are not properly coordinated, the original site may be forced to roll back farther. This can keep up indefinitely, hence the term domino effect.

4 Why Nested Transactions?

There are two basic reasons why nested transactions are desirable: they provide safe concurrency *within* transactions, enhancing both performance and modularity; and they offer finer grained recovery, permitting better control over transaction execution, simplifying the programming of reliable transaction systems. Let us consider each reason in turn.

With single level transactions, concurrency within a transaction could be as dangerous as concurrency without a transaction mechanism at all. For example, suppose there were two transactions you wished to have executed together as one transaction. Further suppose that the two transactions generally would not interfere with one another. For higher throughput, you might be tempted to run the two old transactions concurrently within a new transaction. But if the two old transactions shared data in a way you did not understand, inconsistencies could result. This problem would be especially insidious if the transactions' behaviors changed substantially in different situations.

Nested transactions provide a simple solution to the problem: just run the two old transactions as separate children of a new top level transaction. The children could run in parallel. In cases where there was no conflict on data items, the full benefits of concurrency would accrue. When there was conflict, the transactions would be automatically serialized: no inconsistencies could result, while considerable overlap of execution might still occur. Proper handling of concurrency within transactions can thus simplify programming and make a system more robust.

Nested transactions also provide better control over recovery. When a part of a transaction is considered more than usually likely to fail (such as a lock request or an interaction with a remote computer or operator), that part can be made a child transaction. This permits the rest of the transaction's work to be preserved even when one part fails. It also permits the transaction to handle the failure by either retrying the failed operation or taking another approach. Of course, this is useful only if a retry might succeed, if there is an alternative action to be taken, or the operation was not essential in the first place.

The ability to recover from failure of a remote operation is a strong advantage for nested transactions in distributed systems. Later we will see how simple some things become if you have a nested transaction mechanism in a distributed system. Part of the simplicity

comes from being able to handle failure in the first place; another part comes from the clean semantics of failure: any work done by a failed transaction will be automatically undone (by the recovery manager). This leads to a tradeoff: a nested transaction recovery manager will necessarily be somewhat more complicated than a recovery manager for atomic transactions. But the benefits are great, and the added complexity is not severe. A later section contains a sketch of a distributed transaction management algorithm for nested transactions.

5 Distributed Systems

Up to this point nested transactions have been discussed without regard to whether or not they are embedded in a centralized or distributed system. The concepts are general enough for distributed systems, but there might be many ways to build distributed nested transactions. I will suggest one way that has the appeal of being simple yet very powerful.

First, in my model neither data objects nor individual transactions are distributed. Another way of putting this is: each data object and each transaction has a single site of the distributed system called its **home**. The home site of a data object has complete control over that object and its manipulation, following the concurrency and recovery rules presented in previous sections. This means that locks and shadows (or logs) are entirely local and need never be copied across the computer network. However, as will be seen later, a transaction can effectively hold locks and shadows at other sites.

The main implication of this approach (that each object has a single home) is that any replication or movement of objects is explicit. It does not mean that replication or migration cannot be implemented — only that they are not built in. In fact, one of the examples offered later is the reading and updating of replicated data objects.

Just as each object has a single home, so does each transaction. However, a transaction's children need not have the same home as the parent or each other. A transaction accomplishes remote actions through its inferior transactions, which run at other sites on its behalf. In this way the restriction of transactions to single sites is not really a limitation of functionality. It does simplify system design and implementation, though. This is because the abort/commit decision of each transaction can be made locally, without negotiation between multiple sites. There is one exception: a top level transaction must coordinate using a two phase commit protocol of some sort. Only top level transactions need two phase commit because only they make permanent changes to the database.

My model does not specify the relationship of any user- or application-level message passing (communication) mechanism to the nested transaction mechanism. If applications

may freely communicate, care must be taken not to violate serializability through early release of tentative information. Ill disciplined communication can destroy consistency just as easily as ill disciplined object manipulation. For a number of applications it may be sufficient to have communication modeled after procedure calls. A child transaction is provided parameter information from its parent when the child is created (even if the child is at a different site from the parent). Similarly, a child transaction can provide result data back to its parent if and when the child commits. This rather simple mechanism may not be convenient for all applications. More work is needed in this area.

Let me make a few comments concerning assumptions I make about distributed systems. I assume that sites alternate between periods in which they run correctly and periods in which they are **crashed**. A crash causes a site (effectively) to halt, losing information in volatile storage, but retaining (perfectly) all information in stable storage. I also assume that no site stays crashed forever, but that assumption can be overcome by using more sophisticated two phase commit procedures combined with appropriate data replication. Appropriate replication could also overcome some losses of permanent (stable) storage.

The communications medium used for transferring messages from site to site is not assumed to be reliable. Messages may be lost, corrupted (effectively the same as lost), duplicated, or delivered out of order. However, messages may not be spontaneously created. A delivered message may not differ from what was sent unless the message is obviously corrupted (has a bad checksum). That is, any message received with a good checksum is assumed to be good. I also assume that repeated transmission of a message from one site to any other particular site will eventually work. This does not permit sites to fail permanently. More complicated protocols, such as Byzantine agreement, might overcome this objection, but the expense could be great.

All together, my assumptions about sites and communication are rather weak, if not minimal. Still, in the real world they can only be achieved in a probabilistic sense. The assumptions do have the property that investment of additional resources (redundancy, time, etc.) will decrease the probability of system failure below any fixed positive quantity. These assumptions, their implications and justifications are discussed in more detail in [Moss 85] and [Moss 81].

5.1 Distributed Nested Transaction Management

First, let us consider the data structures necessary for distributed nested transaction management. It is assumed that each transaction's **transaction identifier** (abbreviated tid) is unique over the entire distributed system, and that given a tid the transaction's home node can be derived easily. A simple way of achieving this property is to append the node

number of the creating node to a locally unique identifier. Uniqueness of local identifiers over time can be achieved through using a counter that is saved to stable storage periodically. For example, the counter could be saved once every 1000 transactions. Then the saved value must be 999 or fewer transactions behind the real value. Hence, adding 1000 to the saved value on crash recovery will preserve uniqueness. Note that since the creating node and home node need not be the same (e.g., when a foreign child is being created), two node numbers may need to be included in a tid, one for the creator, and one for the home.

In addition to being able to derive a transaction's home from its tid, it must also be easy to derive a transaction's superiors from its tid. One way to do that is to make tid's variable in length, and include all superiors' ids as a part of each new tid. This is similar to giving the entire path to a file from the root in a hierarchical directory system (e.g., /usr/moss/papers/tods/main.tex). To simplify local manipulation, tid's could be handled as indexes into a local table, but must be translated to their full form before being communicated to another site. Such a scheme allows tid's to be referenced using fixed length codes (the indexes).

In addition to the tid's and their implicit relationships resulting from nesting, concurrency and recovery control information must be maintained. Such information is associated with each transaction, and the associations change when transactions abort and commit, as well as when new resources are touched and processed by a transaction. The changes made in a distributed system are pretty much the same as those made in a centralized system. The primary difference is that some changes to the data structures are instigated by other sites rather than all causes being local. Let us now consider the various events that may occur and the appropriate responses to them.

Touching and processing of objects is a purely local phenomenon and is managed as discussed in previous sections. Differences arise when transactions are created, aborted, and committed.

First, let us turn our attention to transaction creation. Creation of a top level transaction is a local action, with no particular difference from a centralized system. Creation of local children proceeds as described before. It is only creation of foreign children that is new. A record of the children should be kept with the parent: it is useful to know the identities and status (aborted, committed, running, questionable) of a transaction's children. Grandchildren and lower inferiors are not of direct interest.

The steps required to create the child transaction at the foreign node are more complicated. In order to handle later commitment properly, and to provide for the possibility of requests for later child transactions from the same parent, the foreign node needs to record

the parent and all other superiors of the new transaction in its transaction management tables. I call these entries **transaction images**: they are a kind of ghost or plcaeholder of the real transactions. The purpose of images is to permit correct bookkeeping of concurrency and recovery information at the foreign node. When the request to create the child transaction arrives, the foreign node insures that images exist for all superiors of the new transaction, and if they do not exist, it creates them. Then the new child is created, in essentially the same manner as a local child.

Let us now consider transaction commitment. For the moment, ignore top level transactions; their treatment is somewhat different because of the requirement to update the permanent database. When a (non-top level) transaction commits, the concurrency and recovery information is passed up the transaction tree; i.e., it is inherited by the local image of the parent transaction. In this way images render unnecessary the sending of such information across the network. The information is of only local relevance anyway, since it deals only with local objects.

In addition to updating the local tables, which occurs as in a centralized system, the transaction's parent is notified of the commitment. This requires a message to be sent if the parent's home is a different node. In order to take care of lost messages, etc., the parent's node should periodically query the child's node concerning the child's status. Querying allows the system to detect lost requests, lost commitment notifications, and crashes of the child's node. Similarly, the child's node should periodically query the parent's node, to check for crashes, or abort of any of the child's superiors (which requires abort of the child; more on this later). The frequency of queries required for good performance depends on the desired response time, the message passing and handling overhead, and the reliability of the networks and the processors. In many cases queries and responses to them could be "piggy-backed" on other message traffic between the nodes in question.

If the transaction that is committing had any inferiors, their sites should also be notified of the commitment. Such notifications permit the relevant nodes to propagate concurrency and recovery information in their local transaction tables in order to reflect the current abort/commit status of transactions. In particular, commit notices can allow lock propagation, which in turn can free locks and permit blocked transactions to make progress. So that such notifications can be sent, each transaction needs to gather a list of its successful inferiors. For sending notices, a list of the nodes would be sufficient, but the list of successful inferiors will be used later anyway. It could also be useful to keep a list of known aborted inferiors, or at least the nodes on which they were run. This will also play a role later.

In sum, upon transaction commitment, local bookkeeping information is propagated as

for a centralized system, a notice is sent to the parent indicating commitment and including a list of successful and known unsuccessful inferiors of the committing transactions, and similar notices of commitment are sent to all other nodes where the committing transaction is known to have had inferiors.

When a transaction aborts, the same actions are taken except that an aborting transaction has no successful inferiors (by definition), so the lists sent with the abort notice contain only lists of known unsuccessful inferiors.

When an abort or commit notice is received, a node updates its tables appropriately. Note that the node can learn of abort or commit of transactions other than the one that is the subject of a message. The list of successful inferiors allows the local records of any of those transactions to be updated. Suppose there is a transaction T that is an inferior of the subject S of the abort or commit notice. Further suppose that T is not on the list of successful inferiors of S. Then T should be aborted, whether or not it is on the list of unsuccessful inferiors of S. The node should update its tables to reflect the abort of such transactions.

To summarize, in addition to peforming local table updates, transaction creation, abort, and commit can require requests or notices to be sent to other nodes. Abort and commit require notification of the transaction's parent and all of the transaction inferiors. Such notices must contain a list of all the transaction's successful inferiors.

Note that it is not required that a transaction wait for all of its children to complete before committing. This does introduce some problems (see the later discussion of "orphans"), but gives better performance when crashes or communications failures prevent determination of the status of children. Of course, a transaction should never commit unless "enough" of its children are known to have committed. The "enough" means that the transaction's specification will be met, although the ideal may not have been attained. The scheme described here will eventually abort all in-doubt or status-unknown children. With slight additional complexity one could indicate which unknown or in doubt children are "desirable" but not essential - the ones that it would be good to have commit, but not necessary in order to commit the parent.

5.2 Top Level Transaction Commitment

Up to this point only the commit procedure for non-top level transactions has been described. Committing top level transactions requires a **two phase commit** protocol, such as that described in [Gray 78] and elsewhere. The exact protocol used is not dictated by the use of nested transactions: many different protocols are feasible. I will now briefly describe a simple one that you might have seen before.

The home node of the top level transaction plays the role of the **coordinator** of the protocol. All nodes known to have run inferiors of the transaction to be committed are **participants**. It does not matter whether the inferiors were successful or unsuccessful. The coordinator's node is included as a participant in order to simplify the description. The coordinator requests each participant to **prepare** to commit. This is the first phase of the protocol. The prepare request includes a list of the successful inferiors of the committing transaction. This permits each participant to insure that its information concerning the transaction and its inferiors is up to date. As a result, a participant may need to update its tables to reflect transaction aborts and commits of various inferiors for which the participant's information was old. In most cases the situation is all right, and the participant will get its stable storage ready and tell the coordinator to go ahead. (The stable storage preparation involves getting all changes into stable storage such that the changes can either be performed, in case the transaction is committed, or removed, in case the transaction aborts; see [Oki 83] and [Oki et al. 85] for more details.)

However, if the top level transaction believes a particular inferior committed, and the participant sees that the inferior either aborted, or was local to the participant and the participant has no record of it, then the top level transaction cannot be committed. Such a situation will arise if an inferior transaction commits and the node on which it was run crashes after the inferior commits but before the top level transaction commits. Some sort of checkpointing or early commit mechanism would be required in order to avoid this problem.

In any case, the coordinator tallies the responses from the participants. If any participant says it cannot commit (or if the coordinator times out and decides to give up), the transaction will be aborted; otherwise the transaction can be committed. If the transaction is committed, a permanent record of the commit is made before the corrdinator notifies any participants. This deals with the case in which the coordinator crashes.

The participants should probably be notified in case of abort, but it is not required, since they can query the top level transaction's node and find that there is no record of the transaction. This method requires keeping a record of each committed transaction until we are sure every participant knows the transaction is committed. Unfortunately it is impossible to record only the aborted transactions. This is because a transaction is implicitly aborted by a crash, and in that case no record of it may exist. Actually, such records could be kept, but it would slow down transaction creation, since a permanent record would have to be made before each transaction could actually start.

In short, top level transactions are committed through the use of a two phase commit protocol, which insures that either all changes made by the transaction are permanently

recorded, or none are. The usual two phase commit protocol is extended to use the list of successful inferiors to resolve differences in transaction table information, and to detect certain crashes and force transaction abort.

5.3 Orphans

Distributed nested transactions introduce the possibility of discrepancies between a transaction's actual state (running, committed, aborted) and its state as known at another node. This is inherent in any distributed system. The distributed transaction management algorithms discussed in the previous sections automatically handle most of these discrepancies well. However, node crashes do introduce the possibility of inferior transactions that continue running even after one or more of their superiors has aborted or committed. Such inferiors are known as **orphans**.

Orphans arise when a transaction aborts without informing its inferiors appropriately. Most often this would occur because the transaction's node crashed, but sometimes lost messages might create orphans. Orphan creation can be restricted to crashes by requiring a transaction definitely to complete (abort or commit) all of its children before its own completion. Such a protocol could be expensive, however, in terms of delays and message traffic. (It is similar to two phase commit.) Avoiding orphans as a result of crashes requires saving the identity of foreign child transactions in stable memory before sending the creation request to the foreign node. This might also be expensive and time consuming.

The automatic querying of parent transactions will cause orphans to detect abortion of their parents eventually. Once an orphan detects that it is in fact an orphan, it can abort. Note also that orphans cannot have any permanent effect on the database, *provided* that all appropriate locking and recovery rules are in fact enforced, with no loopholes. Thus, if one is strict in the application of the theory to practice, orphans are not dangerous. Their only harm is the needless consumption of resources, and eventual detection is moderately cheap (the querying previously described).

If one is not entirely strict concerning the actions that can be performed by inferior transactions, then orphans can be more of a problem. The difficulty is that a subtransaction, like a procedure, can be written in such a way that it assumes certain conditions hold in its input data. In technical terms, the specification of the subtransaction includes a **precondition**. Such a precondition would generally be obtained by running a prior subtransaction whose postcondition implies the required precondition. A typical example is screening input data for validity in some way. However, because of crashes, a precondition can fail to hold, the required prior effects having been (implicitly) undone by the reload after the crash. When a program's precondition fails to hold, its behavior can be arbitrary.

That is the problem.

There seem to be only three solutions to the potential problem of orphans: first, one can be strict and guarantee no permanent effects; second, one can devise a way of checkpointing, and thus avoid the effects of crashes at the appropriate junctures; and third, one can pass around appropriate information so that the non-serializable situations that would cause orphans to misbehave are detected before the misbehavior can occur. The Argus project has been working on the last approach. Their technique is based on tracing information flow between transactions in the transaction tree and guaranteeing that prior transactions' effects have not been wiped out. The approach is somewhat intricate and hard to understand, but may work at acceptable cost. Checkpointing would be a simpler method to handle non-strict system designs, but the details have not been worked out. For checkpointing to be acceptable in terms of performance, high speed stable storage would be required. That also might be feasible through clever interfaces to fast (e.g., semi-conductor) memory set up to survive power failures.

Finally, orphans may not be much of a problem in practice. However, it seemed important to point out that they might present some difficulties, and that some systems might have to deal with them.

5.4 Distributed Example 1: Remote Procedure Call

Now that I have described how to imbed nested transactions in distributed systems, and sketched a set of protocols that can be used for their management, I will present two examples to demonstrate the usefulness of nested transactions to distributed processing. The intended point is that nested transactions permit simple and elegant implementation of reliable applications in distributed systems. Though I have no specific evidence to support my belief, I think that nested transactions can be used to achieve the stated objective with competitive performance.

My first example is remote procedure call. A remote procedure call is a request made of another node to perform an action and/or return results. It requires the ability to send procedure arguments (input variable values) and obtain results (output values). The idea is that a remote procedure call will boil down to a request/response message pair when everything is working well. It is desirable that remote procedure calls be performed either exactly once (by appropriate retry and duplicate detection) or at most once (skipping retry). It is also nice if the foreign node, as part of its actions in handling a remote procedure call request, can make remote procedure calls of its own, to request assistance in fulfilling its specification.

Nested transactions provide a very natural way of implementing remote procedure

calls: each call is an inferior transaction run at the target node. However, it is useful to add an extra layer of nesting. A remote call then works as follows. The caller creates a *local* child transaction to handle the remote call attempt. This child creates a *remote* grandchild to run at the foreign node, and transmits the call request. The local child may hear from the foreign grandchild that the grandchild has aborted or committed. In the case of commitment, the child commits and the call was successful. In the case of abort, the child can retry by creating another grandchild; or the child can abort (perhaps after a certain number of failed attempts), indicating failure to the original caller.

It is also possible for the child not to hear anything and then time out. In that case, the child can abort. This will cause any actions at the foreign node to be undone, whether or not the grandchild was created, aborted, running, or committed. Lost messages or foreign node crashes could result in call timeout. In case of a failure, extra communication will be required before the nodes are in step with each other (appropriate transactions aborted/committed), but the definitions of nested transaction locking and recovery guarantee good behavior. The two levels of nesting just described can guarantee at-most-once execution of remote calls when a local timeout is used. For exactly-once execution, the caller can loop retrying the at-most-once protocol until it succeeds.

Some very nice properties result from building remote procedure calls with nested transactions. First, at-most or exactly once execution can be guaranteed regardless of the semantics of the call. This is in contrast to the remote procedure call scheme developed in [Lampson and Sturgis], where only certain kinds of actions and sequences of actions are permitted. Granted, my scheme requires concurrency control and recovery, but reliability requires such features somewhere in the system. I believe that straightforward and uniform application of the simple semantics of nested transactions makes programming easier and reduces the likelihood of errors in programs. Another feature of nested transaction based remote procedure call is that the procedure calls nest to any depth with no problems, even when the calls are to sites already visited. Again, [Lampson and Sturgis] cannot make the same claim. Communications and node failures are handled simply and uniformly by the distributed nested transaction management protocols — no additional data structures or communication are required. Finally, multiple calls may proceed in parallel.

5.5 Distributed Example 2: Replicated Data

This example shows how nested transactions can make it simple to access and update a distributed database in which logical objects can be replicated. The techniques presented are very flexible and can achieve a wide range of policies, allowing considerable control over the possible tradeoffs involved.

First, assume that a fixed, known, number of copies of some object exist, and that the location of the copies is known. The number and location of copies of a logical object would be maintained by a directory system, the details of which will not be discussed. Such a directory scheme could use the same algorithms described below. However, the copies of the directory should be in known locations (or available from a known name server, etc.).

Based on the ideas presented in [Gifford 79], each copy of a replicated object is given a number of **votes**. Each lock mode has a **quota**. To acquire a lock on the logical object in a given mode requires locking individual copies in the same mode. The sum of votes of the copies successfully locked must equal or exceed the established quota. For correctness, the quotas and votes must follow this rule: if two lock modes conflict, then the sum of their quotas must exceed the sum of the votes of all the copies. If that rule is followed, it is impossible to lock the logical object in simultaneously conflicting modes. Note that some modes, e.g. exclusive mode, conflict with themselves. Hence the exclusive mode quota must exceed half the total votes.

Here is a simple example. Assume the lock modes are the traditional exclusive and share modes (read and write locks). Say there are five copies of the logical object and each has a single vote. Then the write quota must be at least 3 ($2 + 2 = 4 <= 5$ but $3 + 3 = 6 > 5$), and the read quota must be at least 6 minus the write quota. Even in this simple case, we could opt for single copy read (which would require locking all copies for write), majority read and write (both read and write quotas set to three), or an intermediate policy (read quota of 2 and write quota of 4). This reflects flexibility in the trading off of read versus write overhead and delay. It also reflects a robustness tradeoff: if all copies must be locked to write, then all sites must be up to write.

Note that the algorithm works in the face of the permissible failures in the system model. Network partition, for example, will not cause incorrect results (but it may not permit some operations to proceed). The **weighted voting** approach just outlined can also model more interesting situations when the votes are not even. There are additional policy options attainable if copies are locked in particular orders. One might achieve effects similar to centralized locking during normal system operation, while still providing simple recovery from failure (if the primary copy is unavailable, just proceed to the secondary, etc.).

It is easy to implement replicated object access using nested transactions. When it is desired to access a logical object that is replicated, a local access manager child transaction is created. It performs remote procedure calls to each of the copies, requesting the appropriate locks. These remote calls could be done in sequence or in parallel. The manager transaction sees how many of its remote calls succeed and tallies their votes. It can retry

in case of some failures. The manager can also abort after a timeout if that is desired. In any case, the nested transaction management system does the appropriate bookkeeping at the remote sites. If the manager gets sufficient votes, it can commit, otherwise it must abort. Note that the manager can go ahead and request that changes be performed at the remote sites. If such optimism proves to be unfounded, everything is still all right, because the changes will be undone automatically when the manager aborts.

A nested transaction mechanism provides a uniform and simple way to program interesting tasks reliably. It is better to build replication, remote procedure call, and many other application level protocols, on top of nested transactions. The powerful common mechanism provided by nested transactions should simplify other aspects of a distributed system.

6 Nested Transaction Deadlock

As previously mentioned, nested transaction concurrency control is not immune to deadlock. Nested transactions can deadlock in ways similar to single level transactions, with a few new twists now discussed. The essence of deadlock analysis is the **waits-for** relation among transactions. A deadlock exists if and only if the waits-for relation has a cycle. Hence, to understand nested transaction deadlock requires an understanding of the difference between traditional waits-for relations and those for nested transactions.

A direct wait occurs when one transaction requests a lock in conflict with a lock held by another transaction. The lock requestor is said to wait for the lock holder. In single level transaction systems, this is the only kind of waits-for relationship possible. Nested transactions, however, introduce new possibilities.

First, suppose a transaction T_a is in a direct wait for transaction T_b. (See Figure 4 for an illustration of this example.) It is possible that commitment of T_b will not permit T_a to proceed. In fact, this will always occur if T_b is not a sibling of T_a or one of T_a's ancestors. The point is that certain superiors of T_b must commit (in addition to T_b) before access can be granted to T_a. Let $T_{b'}$ be the oldest ancestor of T_b that is *not* an ancestor of T_a. Then T_a must wait until $T_{b'}$ commits. Thus T_a waits indirectly for $T_{b'}$ (and all ancestors of T_b inferior to $T_{b'}$).

Another effect is that the parent of T_a cannot commit until T_a does. This is also an indirect wait. Returning to the example of the previous paragraph, let $T_{a'}$ be the oldest ancestor of T_a that is not an ancestor of T_b. Then $T_{a'}$ indirectly waits for T_a, T_b, and $T_{b'}$, etc. Note that the "real" waits-for situation is represented by the siblings (or top level transactions) $T_{a'}$ and $T_{b'}$.

Figure 4: Deadlock

T_a is waiting directly for T_b.

$T_{a'}$ is the oldest ancestor of T_a that is not an ancestor of T_b. It is possible that $T_a = T_{a'}$.

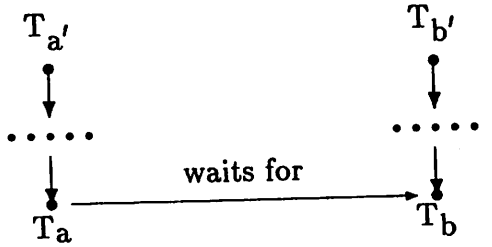
$T_{b'}$ is the oldest ancestor of T_b that is not an ancestor of T_a. It is possible that $T_b = T_{b'}$.

$T_{a'}$ and $T_{b'}$ are either distinct top-level transactions or siblings.

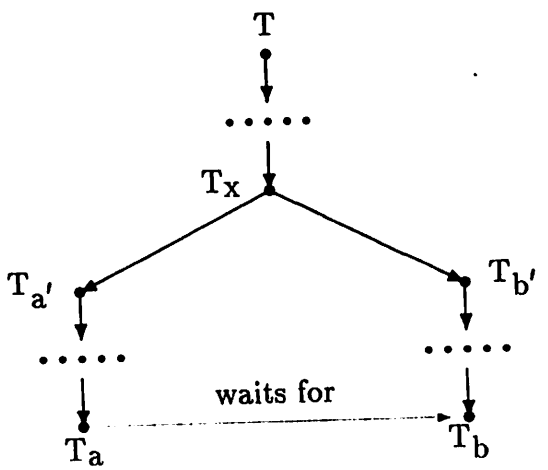
T_a indirectly waits for $T_{b'}$.

$T_{a'}$ indirectly waits for T_b and $T_{b'}$.

(a) T_a and T_b are descendants of different top-level transactions.



(b) T_a and T_b are descendants of the same top-level transaction T.



Once the above described indirect wait conditions have been taken into account, deadlocks take the same form as in a single level transaction system: they are cycles in the waits-for relation. Note, though, that because a transaction can achieve parallelism through the action of its children, all its progress need not cease when it is in a deadlock. However, progress will eventually stop.

The next section describes an algorithm for detecting and resolving nested transaction deadlock in distributed systems. In addition to a basic algorithm, two improvements are suggested, which could substantially reduce the number of messages required to detect deadlocks. The section after that covers nested transaction deadlock avoidance techniques, which can be used if one prefers to avoid the additional complexity of deadlock detection, or if the application domain is such that deadlock avoidance is more efficient than detection.

6.1 Distributed Deadlock Detection and Resolution

Detecting nested transaction deadlock in a centralized system is conceptually simple: maintain a data structure representing the waits-for relation (including the indirect waits implied by nested transaction relationships), and check that data structure for cycles when new waits-for edges are added. The method I suggest for detecting nested transaction deadlock in a distributed system is somewhat different: **edge chasing.**

Suppose a transaction attempts to acquire a lock and finds it cannot proceed because one or more other transactions hold conflicting locks. The requesting transaction will start to wait directly for those other transactions. For each such blocking transaction T_b, the requesting transaction T_a determines the oldest ancestor of T_b that is not an ancestor of T_a; call this ancestor transaction $T_{b'}$. Note that $T_{b'}$ is the transaction that is "really" blocking T_a. T_a sends a message to the transaction manager for $T_{b'}$ indicating that T_a is waiting for T_b, an inferior of $T_{b'}$. This transmission of a message because of a new wait is called **initiation** of deadlock detection.

When a transaction manager receives a message describing waits, it can determine the awaited transaction (in this case, $T_{b'}$). The notice is forwarded to each of child of $T_{b'}$; the children forward it to their children; etc. In this way the notice is propagated to all inferiors of $T_{b'}$. Now if any descendant of $T_{b'}$ is currently waiting to acquire a lock, deadlock detection is **continued** by forwarding a new message to the awaited transaction. For example, let $T_{b''}$, an inferior of $T_{b'}$, be awaiting T_c directly, and $T_{c'}$ indirectly. Then when the notice concerning T_a and T_b reaches $T_{b''}$, a notice listing both the T_a/T_b wait and the $T_{b''}/T_c$ wait is sent to $T_{c'}$. This notice is forwarded to the inferiors of $T_{c'}$, and so on.

Any leaf transaction that receives a wait notice and is not waiting, simply discards

the wait notice. However, if there is a deadlock, the notices will follows the waits-for relationship, and eventually some transaction manager will find that one of its transactions is waiting (directly or indirectly) for one of the transactions in the list in a just received wait-notice. In sum, the wait notices trace out the waits-for graph and record a path through that graph in the messages. Since the messages are propagated along all edges in the graph, if there is a cycle it will be detected. Also, there is no false detection of deadlocks, as might occur if waits-for information were periodically gathered and joined together in a centralized or hierarchical fashion. For the moment, assume that wait notice messages are never lost.

The deadlock detection algorithm just described is perfectly feasible, and straightforward to implement. However, some simple techniques can substantially reduce the number of messages required. Let us assign each transaction a unique **priority**. Priorities are most appropriately assigned based on time of entry into the system, but limited exceptions to that can be tolerated without introducing the possibility of continually aborting a transaction because higher priority transactions keep entering the system. The priorities can be made unique by using a locally unique timestamp with the number of the generating node appended. Note that the same method can be used to generate tid's, so in fact a transaction's tid, interpreted as an unsigned number, can be its priority. In such a case, higher numbers indicate lower priorities. For reasonable comparisons between transactions created at different nodes, the priorities should be based on global system time. It is not necessary to absolutely sychronize all sites; approximate algorithms for maintaining globally consistent times are well known.

The first improvement to the basic algorithm is to initiate edge chasing only when a higher priority transaction waits for a lower priority one. (The converse condition could be used if there were some good reason, but the system must uniformly apply a single rule.) Note that a cycle of transactions cannot exist without such a drop in priority, since all priorities are distinct. Wait messages must be propagated as before. However, this small change should reduce the number of messages by half (if waiting is entirely random), or more possibly even more (higher priority transactions are more likely to be older and possess more locks, hence they are more likely to be awaited than to wait).

Suppose T_a is directly waiting for T_b, using the example discussed just a few paragraphs above. Do *not* compare the priorities of T_a and T_b: those priorities are irrelevant. One must compare the priorities of $T_{a'}$ (the oldest ancestor of T_a that is not an ancestor of T_b) and $T_{b'}$. That is, the comparison must be done at the level in which the cycle would occur for the argument concerning drops in priority to be valid.

The second improvement is this. When, in propagating a wait notice, a drop in priority

is found that goes to a transaction of lower priority than that causing initiation of this wait notice, this wait notice need not be propagated. This rule causes cycles to be detected only as a result of the wait for the lowest priority transaction in the cycle. Again, it could reduce the number of wait messages by a factor of two. Using the example mentioned above, we would stop propagating the message initiated by the drop in priority at $T_{a'}$ to $T_{b'}$ if a drop were encountered having priority even lower than $T_{b'}$.

Transaction wait notices should be retransmitted periodically for two reasons. First, retransmission makes the algorithm robust in the face of lost messages, etc. The second reason is best explained by giving a simple example. Suppose that T_a started to wait for T_b and T_a has higher priority. Then T_a will send T_b an initiation message. At this point, assume T_b is not waiting for any transaction. T_b would just discard the message. If T_b later starts to wait for T_a, we have a deadlock. But because of the relative priorities, T_b will not initiate. Even if we had *any* new waiter (regardless of relative priorities) send up to n initiation messages, all n could still be lost and a deadlock would still not be detected. However, periodic retransmission of initiation by T_a solves the problem: the deadlock will be detected as soon as one of the messages is received by T_b after T_b begins its wait for T_a.

Since retransmission is required to overcome communications failures anyway, we might as well use the initiation and propagation enhancements to the basic deadlock detection algorithm. I do not believe that schemes that attempt to save waits-for information and give acknowledgements to wait notices would offer better performance than the retransmission scheme. Among other things, acknowledgements double the number of messages in cases where retransmissions do not actually occur.

When a cycle is found, the lowest priority transaction in the cycle should be notified. Then it, or an appropriate inferior, should be aborted, to break the cycle and allow progress. Note that the whole transaction need not be aborted — only the inferior that is holding the lock preventing progress. This is one way in which the finer granularity of recovery provided by nested transactions might improve system performance.

It has been conjectured that almost all deadlock cycles that occur in practice are of length two, and I have heard of measurements that seem to support the conjecture [Gray 80]. If it is true that cycles are mainly of length two, then it could be that most distributed deadlocks would be found and resolved using just one message. In the case of larger cycles, the extra messages used by the edge chasing algorithm might be negligible because of the rarity of the situation in the first place. Thus, it could be that nested transaction deadlock detection in distributed systems would be cheap.

6.2 Nested Transaction Deadlock Avoidance

There might be systems in which deadlock detection is not appropriate and deadlock avoidance is better. Avoidance prevents deadlocks by aborting transactions in situations that can lead to deadlock. Detection waits for an actual deadlock to exist.

Avoidance might be better in some systems for at least two reasons. First, if deadlocks are very rare, the extra cost of detection (in terms of system code, etc.) might not be justified. Another possibility is that detection could cause system bottlenecks (such as high traffic locks) to back up too much during the time it takes to detect and resolve a deadlock. In such circumstances it is conceivable, but by no means definite, that avoidance would increase overall throughput by increasing average utilization, even if more transactions are aborted.

In a nested transaction system, the schemes for avoidance proposed in [Rosenkrantz, et al. 78] will work just fine. There are two basic procedures that can be followed, known as **wound-wait** and **wait-die**. Both procedures involve the assignment of unique priorities to transactions and the comparison of these priorities. The priority assignment can be performed as previously described. The comparisons should also be performed as explained there. In particular, the priorities of the directly waiting transactions should not be compared, but rather the priorities of their oldest distinct ancestors. Those ancestors will be either siblings or two different top-level transactions.

Suppose that T_a is about to wait for T_b, and that $T_{a'}$ and $T_{b'}$ are the appropriate ancestors of T_a and T_b respectively. Here is a description of each of the two procedures. Note that any given system must uinformly follow either wound-wait or wait-die, but not both or a mixture.

Wound-Wait:

> If priority (T_a) > priority (T_b), T_a **wounds** T_b, causing T_b to abort.
> If priority (T_a) < priority (T_b), T_a **waits** for T_b.

Wait-Die:

> If priority (T_a) > priority (T_b), T_a **waits** for T_b.
> If priority (T_a) < priority (T_b), T_a **dies** (aborts itself), rather than wait for T_b.

In each scheme higher priority transactions will make more progress. The schemes work by preventing either drops in priority along waits-for edges (wound-wait), or increases in priority along such edges (wait-die). The two schemes can be expected to have different

performance. For example, if older (higher priority) transactions hold more locks and request fewer locks, then wound-wait will probably abort fewer transactions.

In sum, well known deadlock avoidance schemes can be used for nested transactions, once it is understood how to take the extra indirect waits of a nested transaction system into account.

7 Summary and Comments

This paper has introduced the concept of nested transactions and sketched their implementation. This includes concurrency control via locking and recovery control via shadow pages. It has also described how to use and implement nested transactions in distributed systems, including some particularly relevant examples: remote procedure call and replicated databases. In addition, deadlock detection and avoidance techniques were presented, which apply to both centralized and distributed systems. The implementations that have been suggested for nested transactions, deadlock detection, and deadlock avoidance in distributed systems are quite robust and work in the face of a variety of failures. I believe that nested transaction are a promising tool for structuring reliable distributed systems and reducing the complexity of applications software for such systems, without sacrificing reliability or performance.

The concepts embodied in nested transactions seem to have been recognized first by Davies [Davies 73]. My doctoral research, reported in [Moss 81, Moss 82, Moss 85], built substantially on those concepts and presented a comparatively detailed model of distributed computing, nested transactions, and their implementation. But I am grateful to Davies and others for pointing the way, particularly: Prof. Jerry Saltzer at MIT for leading a graduate seminar on atomicity, concurrency, and recovery; Dr. Jim Gray, for his "Notes on Database Operating Systems" and other works; Prof. Barbara Liskov, for her continuing support.

References

[Bernstein, et al. 78] P. A. Bernstein, J. B. Rothnie, N. Goodman, C. A. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, May 1978, pp. 154-167.

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

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

[Gifford 79] David K. Gifford, "Weighted Voting for Replicated Data", *Proceedings of the 7th Symposium on Operating Systems Principles*, Pacific Grove, CA, December 1979, pp. 150-162.

[Gray 78] J. N. Gray, "Notes on Database Operating Systems", in *Lecture Notes in Computer Science*, R. Bayer et al., eds., Springer-Verlag, 1978, pp. 393-481.

[Gray 80] J. N. Gray, photocopies of transparencies for a talk concerning experience with System R, 1980.

[Lampson and Sturgis] Butler Lampson and Howard Sturgis, "Crash Recovery in a Distributed Data Storage System", Xerox Palo Alto Research Laboratory.

[Moss 81] J. Eliot B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, April 1981; also available as Massachusetts Institute of Technology Laboratory for Computer Science Technical Report 260.

[Moss 82] J. Eliot B. Moss, "Nested Transactions and Reliable Distributed Computing", *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1982, pp. 33-39.

[Moss 85] J. Eliot B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", MIT Press, 1985 (updated and revised edition of PhD thesis).

[Oki 83] Brian M. Oki, "Reliable Object Storage to Support Atomic Actions", MS thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, available as Massachusetts Institute of Technology Laboratory for Computer Science Technical Report 308, May 1983.

[Oki et al. 85] Brian M. Oki, Barbara H. Liskov, and Robert W. Scheifler, "Reliable Object Storage to Support Atomic Actions", *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985, pp. 147-159.

[Reed 78] David P. Reed, "Naming and Synchronization in a Decentralized Computer System", PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1978; also available as Massachusetts Institute of Technology Laboratory for Computer Science Technical Report 205.

[Rosenkrantz, et al. 78] D. J. Rosenkrantz, R. E. Stearns, P. M. Lewis II, "System Level Concurrency Control for Distributed Systems", *ACM Transactions on Database Systems*, Vol. 3, No. 2, June 1978, pp. 178-198.

[Rosenkrantz, et al. 80] D. J. Rosekrantz, R. E. Stearns, P. M. Lewis II, "Consistency and Serializability in Concurrent Database Systems", Deptartment of Computer Science, State University of New York at Albany, Technical Report 80-12, August 1980.

[Russell 80] David L. Russell, "State Restoration in Systems of Communicating Processes", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 2, March 1980, pp. 183-194.

[Takagi 79] Akihiro Takagi, "Concurrent and Reliable Updates of Distributed Databases", Massachusetts Institute of Technology Laboratory for Computer Science Technical Memorandum 144, November 1979.

[Verhofstad 78] J. S. M. Verhofstad, "Recovery Techniques for Database Systems", *ACM Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 167-195.