Transactions: A Construct for Reliable Distributed Computing

Alfred Z. Spector and Peter M. Schwarz

January 4, 1983

Abstract

Transactions have proven to be a useful tool for constructing reliable database systems and are likely to be useful in many types of distributed systems. To exploit transactions in a general purpose distributed system, each node can execute a transaction kernel that provides services necessary to support transactions at higher system levels. The transaction model that the kernel supports must permit arbitrary operations on the wide collection of data types used by programmers. New techniques must be developed for specifying the synchronization and recovery properties of abstract types that are used in transactions. Existing mechanisms for synchronization, recovery, deadlock management and communication are often inadequate to implement these types efficiently, and they must be adapted or replaced.

Technical Report CMU-CS-82-143

This work was sponsored in part by: the USAF Rome Air Development Center under contract F30602-81-C-0297; the US Naval Ocean Systems Center under contract number N66001-81-C-0484; the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539; and the IBM Corporation.

1. Introduction

Distributed computing systems are potentially reliable, because the redundancy and autonomy present in them permit failures to be masked or localized. A major challenge in distributed computing research is to realize this potential without incurring intolerable penalties in complexity, cost, or performance. Consequently, there is currently great interest in general-purpose methodologies and practices that simplify the construction of efficient and robust distributed systems. This paper discusses a methodology based on *transactions* and includes a survey of considerations in the design of a *transaction kernel*: an abstract machine that supports transactions.

Transactions were originally developed for database management systems, to aid in maintaining arbitrary application-dependent *consistency constraints* on stored data. The constraints must be maintained despite failures and without unnecessarily restricting the concurrent processing of application requests.

In the database literature, transactions are defined as arbitrary collections of operations bracketed by two markers: *BeginTransaction* and *EndTransaction*, and have the following special properties:

- Either all or none of a transaction's operations are performed. This property is usually called *failure atomicity*.

- If a transaction completes successfully, the results of its operations will never subsequently be lost. This property is usually called *permanence*.

- If several transactions execute concurrently, they affect the database as if they were executed serially in some order. This property is usually called *serializability*.

- An incomplete transaction cannot reveal results to other transactions, in order to prevent *cascading aborts* if the incomplete transaction must subsequently be undone.

Transactions lessen the burden on application programmers by simplifying the treatment of failures and concurrency. Failure atomicity makes certain that when a transaction is interrupted by a failure, its partial results are undone. Programmers are therefore free to violate consistency constraints temporarily during the execution of a transaction. Serializability ensures that other concurrently executing transactions cannot observe these inconsistencies. Prevention of cascading aborts limits the amount of effort required to recover from a failure.

Database management systems are not the only ones that must assure the consistency of stored data despite failures and concurrency. Various ad hoc techniques have evolved for this purpose. For example, TOPS-10 [Digital Equipment Corporation 72] and numerous other file systems permit atomic updates to a single disk file. This technique lacks flexibility and generality, however, and leads to unnecessary restrictions on concurrency.

Considerable research effort is currently being expended towards extending the utility of transactions beyond database applications. At MIT, the Argus project [Liskov 82a] is adding transaction facilities to the CLU language. Transactions will also be available in the Clouds distributed operating system [Allchin 82].

At Carnegie-Mellon, we are exploring the idea of implementing a *transaction kernel* on each node of a distributed system. A transaction kernel is a basic system component that supplies primitives for supporting transactions and the shared abstract data types on which they operate. Complex, costly, and redundant error recovery mechanisms could be avoided elsewhere, if this facility were available. A transaction kernel should also lead to compatible structuring of the various systems that use it, simplifying their interconnection.

This report is an overview of recent research on transaction systems, and surveys issues that arise in developing a transaction kernel. We consider the extension of transactions to general programming and discuss how a transaction kernel should facilitate data abstraction. Subsequent sections examine what we believe to be the central issues in building a transaction kernel: synchronizing access to shared abstract types without unnecessarily restricting concurrency, managing deadlocks, recovering from failures, and communicating efficiently between sites. For more on the extended use of transactions, we refer the reader to recent reports by Liskov, Allchin, Jacobson, and ourselves [Allchin 82, Liskov 82b, Liskov 82a, Jacobson 82, Schwarz 82].

2. Extensions to the Transaction Model

A construct that gives programmers a uniform strategy for treatment of failures, controls interaction between concurrently executing processes, and ensures permanence of operations should simplify the production of reliable distributed systems. Except for database applications, however, the utility of transactions has not been widely demonstrated. Lomet hypothesized that transactions would be useful for general programming [Lomet 77], but the literature includes sketches of only a few non-database systems based on transactions [Liskov 82a, Allchin 82, Gifford 79, Daniels 82].

The traditional transaction model, as described by Gray [Gray 80], was designed primarily for understanding database management applications. It must be extended to model the additional requirements imposed by general-purpose distributed systems. For instance, real-time systems may require real-time synchronization of the participants in transactions (see Section 6). File and mail systems that are both highly available and highly reliable are also difficult to implement unless constructs not in the traditional transaction model are used. Their transactions are more complex than those in database systems, and their performance requirements are potentially higher. Gray also comments on the limitations of the traditional transaction model [Gray 81a].

Database systems, and their transaction mechanisms, do not fully support the abstract data types that are required in more general systems. In a database, the basic unit of information is the typed record, which can be aggregated into indexed files. The only operations on records are read and write, and only operations such as insert, lookup, or sort are defined for files.

Systems that encourage data abstraction must be more flexible. They must permit the definition of arbitrary object types with corresponding sets of type-specific operations. They must also allow new object types to be implemented by combining existing ones, and the resulting types should appear to their users as primitive types. Rather than a sequence of reads and writes on records, a transaction becomes a hierarchy of typed operations on objects. Transactions can be nested, if some of the operations in the hierarchy are themselves implemented with transactions.

Nested transactions are also useful for controlling the interaction of multiple processes within a single transaction, or salvaging partial results when a transaction aborts. For example, some real-time applications employ fairly lengthy transactions. If aborting such transactions and restarting them from the beginning would cause intolerable delays, the transactions must instead fall back to intermediate *save points* [Gray 81c]. See Reed's and Moss' theses for more about nested transactions [Reed 78, Moss 81].

In database systems, application programmers do not have to specify the consistency constraints that they wish transactions to preserve. By guaranteeing serializability of all transactions, database transaction mechanisms assure that any consistency constraint preserved when a transaction runs in isolation will also be preserved when transactions run concurrently. The transaction manager must delay or abort transactions as necessary to make this guarantee. If the system were aware of the specific consistency constraints that transactions were intended to maintain, it could use this extra information in deciding whether or not to delay or abort transactions. Avoiding unnecessary delays and aborts would improve performance. Semantic knowledge about individual types, their operations, and their implementations could also be used to make better-informed decisions regarding concurrent access to objects. A transaction mechanism efficient enough for use in general-purpose distributed systems must be flexible enough to allow such use of semantic information to achieve greater concurrency.

Our approach is to focus on the individual shared abstract types that programmers use in constructing transactions. In addition to the traditional properties of abstract types, these types can be characterized by their synchronization and recovery properties. The specification of these properties defines the types' exact behavior under conditions of concurrency or failure. Assuming different types cooperate in a reasonable fashion, the specifications allow programmers to determine whether particular types will meet their needs. Sections 3 and 5 discuss in detail the synchronization and recovery properties of shared abstract types.

We have found types with specialized synchronization and recovery properties to be useful in designing a highly available message system. For example, message repositories, which are replicated on several sites, are highly specialized shared abstract objects with unique sets of operations. In addition to reading and writing messages, special operations permit out-of-date message repositories to "catch up" with current ones. The recovery and synchronization properties of these operations are type-specific and must be carefully specified and analyzed.

3. Synchronization

In a transaction-based system, synchronization is important to both the specification and the implementation of shared abstract types. Traditional methods for synchronizing access to objects (e.g., monitors [Hoare 74]) just prevent concurrent operations on a particular object from interfering with one another. Maintaining consistency constraints that encompass groups of objects necessitates additional synchronization. However, the mechanisms that enforce this additional synchronization must not unnecessarily restrict concurrency. Because transactions are arbitrarily large collections of operations, a synchronization action that is in force over the entire scope of a transaction can potentially degrade performance more severely than a synchronization action that only affects a single operation.

One approach to synchronization in a general transaction-based system is to classify each operation on an abstract type as either a Read or a Write. A two-phase Read/Write locking scheme [Eswaran 76] ensures serializability and, if locks are held until end-of-transaction, prevents cascading aborts as well. However, such techniques for managing concurrency make minimal use of semantic knowledge about the objects that transactions manipulate, and therefore they may prevent or delay operations unnecessarily.

For example, consider two transactions that each insert a new entry in a directory object. Since the insertion operations modify the directory object, one must classify them as Write operations. The standard rules for Read/Write locking prohibit modification of an object by more than one incomplete transaction. The system would therefore delay the second insertion until the transaction making the first one either committed or aborted. Closer examination of the semantics of insertion reveals that this is unnecessary if the two insertions specify different keys. A synchronization mechanism that could use this extra knowledge could achieve greater concurrency.

Similarly, specifying serializability as the goal of a transaction synchronization strategy reflects a limited use of semantic knowledge. Serializability makes sure that any invariant preserved by an individual transaction will also be preserved when transactions execute concurrently. This guarantee is frequently too strong. For instance, consider a queue that buffers units of work between activities that produce and consume them. Serializing the transactions that operate on the buffer queue groups together all entries made by a single transaction, in order to enforce their consecutive removal. In many applications, ordering of entries in the buffer is not crucial as long as entries for which the inserting transaction has committed eventually reach the head and can be removed. Entries inserted by incomplete transactions must not be removed, however, so that cascading aborts cannot occur. As in the preceding example, using more semantic knowledge about the object and its intended purpose can lead to greater concurrency.

Many authors [Eswaran 76, Kung 79, Allchin 82, Garcia-Molina 82, Sha 83] have observed that using semantic knowledge can increase concurrency. While Garcia and Sha consider the properties of entire transactions, we are concentrating on the semantics of operations on individual types. To exploit this approach, one must first be able to specify precisely and concisely how a type behaves under conditions of concurrent access by multiple transactions. Prospective users need such a means of specification to define their own requirements and to compare them with the properties of available types. We have investigated *dependencies* as a tool for this purpose. Dependencies were originally used in database research for proving the correctness of two-phase locking protocols [Eswaran 76, Gray 75]. A dependency exists between any two transactions that perform an operation on a common object, and the dependency defines the order in which the two transactions operate on the object.

One can prove that if the transitive closure of all the dependencies among transactions forms a partial order, then the execution of the transactions is serializable [Eswaran 76]. If the transitive closure contains cycles, the ordering of transactions is ambiguous. Not all dependencies are equivalent, however. For example, the semantics of the Read operation tell us that the order in which two transactions read a common object has no effect on the transactions' outcome. Even though the transitive closure of all dependencies has cycles, disregarding these meaningless dependencies and recomputing the transitive closure may result in a partial order of the transactions. In general, a group of transactions is *orderable* with respect to a particular group of *proscribed* dependencies if the transitive closure of the proscribed dependencies yields a partial order. Serializability in a database with Read/Write locking can be defined in these terms as orderability with respect to all dependencies except those for which both operations are Reads [Gray 75].

In a general-purpose system with arbitrary shared abstract types, a set of proscribed dependencies must be defined for each type. Semantic knowledge about individual types can be used in constructing this set, to achieve high concurrency while still helping the programmer to preserve consistency. For instance, the proscribed set of dependencies for directories would not include dependencies between transactions operating on entries with different keys. Like dependencies in which both operations are Reads, these dependencies cannot affect consistency. To specify a queue type for which grouping of elements by inserting transaction is not assured, dependencies between transactions performing the insert operation can be removed from the proscribed dependency set.

When a transaction accesses several objects of different types, the types must cooperate to maintain global consistency. In addition to guaranteeing orderability with respect to the proscribed dependency sets of the individual types, the transaction manager must also preserve orderability with respect to the union of the proscribed dependency sets.

Dependencies can also be used to specify which operations must be delayed to prevent potential cascading aborts. Whenever a dependency is about to form between two incomplete transactions, the second transaction may have to be delayed in case the first one aborts. The decision whether or not to delay depends on the exact dependency being formed. Analogous to the proscribed dependency set, each type must specify a deferred dependency set that determines the circumstances under which operations will be delayed until a prior transaction commits or aborts. Usually, dependencies that represent a transfer of information between the two transactions must be deferred. A more extensive treatment of the dependency technique, including detailed examples, can be found in a related paper [Schwarz 82].

Shared abstract types can be divided into three categories based on their synchronization behavior. The categories are listed in order of increasing potential for concurrent access, and each properly includes the preceding ones.

1. Types that serialize access to objects. These types can use semantic knowledge to permit greater concurrent access to an object without losing the advantages of serializability. The directory that allows concurrent operations on entries with different keys is in this category. The proscribed dependency sets for these types includes all dependencies that have a detectable effect on transaction outcomes.

2. Types that do not permit incomplete transactions to reveal their results to other transactions. Since transactions are not necessarily serializable, this strategy does not guarantee arbitrary consistency constraints, but can lead to higher concurrency while still preserving properties that are crucial to the purpose of the type. The queue that does not guarantee grouping by inserting transaction is in this category. Some dependencies that affect transaction outcomes can be excluded from these types' proscribed dependency sets.

3. Types with arbitrary synchronization policies. Incomplete transactions that operate on objects of these types may reveal data to other transactions; it is assumed that these data are acceptable (i.e., will not cause cascading aborts) even if the revealing transaction subsequently aborts. An update

> that is used only as a "hint" can be revealed, for instance. Even if another transaction reads the hint while it has an incorrect value, no fatal error will occur. In terms of dependencies, the deferred dependency sets for these types may exclude some dependencies that transfer information.

Given dependencies as a means of specification, a second key to achieving efficient synchronization by utilizing semantic knowledge is the definition of a synchronization mechanism flexible enough to implement a wide variety of shared abstract types. We have examined using *type-specific locking* as this mechanism. Bernstein, Goodman, and Lai [Bernstein 81] discuss some of this method's basic principles. Korth [Korth 83] has described a type-specific approach to locking based on commutativity of operations, which employs a hierarchy of locks to allow variable-granularity locking. Weihl, in connection with the Argus Project, has described *crowds*: an alternative synchronization mechanism for exploiting type-specific semantics [Weihl 81]. Transactions must join a crowd before accessing an object and only leave the crowd when the transaction is complete. Type-specific rules determine whether a transaction should be admitted a crowd or be forced to wait until some other conflicting transaction leaves.

A set of basic principles underlies all locking schemes. Before a transaction manipulates an object, it must obtain a *lock* on the object. Possession of the lock restricts further access to the object by other transactions, until it is released. Locking mechanisms thus control the formation of dependencies among transactions. Whenever one transaction waits for a lock held by another, formation of a dependency between the two transactions is delayed until the lock is released. The protocol for acquiring and releasing locks ensures that if the dependency would become part of a cycle in the transitive closure of a set of proscribed dependencies, a deadlock results and the cycle never forms.

The simplest locking mechanisms have only one kind of lock, regardless of the type of the object to be locked or the operation to be performed. This form of locking uses no semantic knowledge, and cannot distinguish between proscribed and non-proscribed dependencies. Many database systems use a locking mechanism that provides two *lock classes*, Read and Write. Operations that modify an object must first obtain a Write lock, whereas operations that merely reference an object's value need only obtain a Read lock. The rules for obtaining locks specify that multiple transactions may simultaneously hold Read locks on an object, but holding a Write lock reserves the object exclusively for one transaction. By making this coarse distinction among different kinds of operations, Read/Write locking uses limited semantic information to permit some cyclic dependencies while prohibiting others. This yields greater concurrency without compromising consistency.

Type-specific locking generalizes the ideas behind Read/Write locking. Instead of dividing all operations into two broad classes, the implementor of each type can define appropriate type-specific lock classes and associated rules for acquiring and releasing locks. The rules specify the kind(s) of lock required by each of the type's operations and which kinds of locks are compatible with each other. By tailoring the locking strategy to suit a specific type and implementation, type-specific locking preserves only what is promised by the type's specification. A large amount of semantic information about both the specification and the implementation of the type can be used in deciding whether an operation must be delayed or prevented.

Additional research is needed to determine specific primitives for locking, unlocking, definition of new object types, etc. For example, data stored in an object or supplied as an argument to an operation is sometimes crucial in determining the compatibility of two operations. Recall that insert operations on directories are compatible only if they refer to entries with different keys. Type-specific locking primitives must permit the association of auxiliary information with locks on objects.

A related paper by Schwarz and Spector [Schwarz 82] contains further details and examples of type-specific locking. It appears that implementations of type-specific locking mechanisms will be reasonably simple and, in order to understand their details more completely, we are building one using directories as a sample shared abstract type.

4. Deadlock

One must consider the possibility of deadlock in any system where processes may wait for dynamically allocated resources. In a transaction, the resources are the objects that the transaction accesses. There are many strategies for coping with deadlocks, but it is not clear which are most appropriate for transaction-based systems with arbitrary shared abstract types.

One approach is to impose a global ordering on all system resources, and force all transactions to obtain resources according to this ordering. This method is unsuitable, because it does not allow transactions that access a data-dependent collection of objects. When the system initiates a transaction, it must know a priori all the resources the transaction will need. Another technique uses timestamps on transactions or objects to avoid deadlock [Rosenkrantz 78, Reed 78]. A third approach to the problem is to allow deadlocks, subsequently detect them, and ultimately resolve them by selecting a transaction to abort. Either timeouts or an algorithm that analyzes waiting transactions can be used for detection. Unfortunately, employing timeouts causes the timing behavior of an abstract data type's implementation to become a critical aspect of the type's specification. In either case, detection and resolution of deadlocks could become a bottleneck that would constrain performance.

Arbitrary type-specific locking protocols can cause another problem. If a protocol allows the release of some locks prior to end-of-transaction, it may be necessary to re-acquire them later to process an abort. Re-acquisition violates the common simplifying assumption that aborting a transaction never requires additional resources. Deadlocks can therefore occur during abort processing, and the standard approach of aborting a waiting transaction cannot resolve them.

There has been fairly little formal analysis of the relationships between the probability of waiting or deadlock and such factors as degree of multi-programming, number of operations in a transaction, and size of the shared database. However, Gray et al. and Lin et al. [Gray 81b, Lin 82] have each modeled both the probability of waiting for a lock request and the probability of deadlock in two phase locking protocols, and they conclude that both probabilities rise with the degree of multiprogramming. They also report that the probabilities of deadlock and waiting rise more than linearly in the number of operations per transaction. These pessimistic conclusions are based on very simple models. They must be adapted if they are to represent accurately the behavior of transactions that access a hierarchically structured graph of typed objects. It seems reasonable, however, to conclude that if many transactions frequently access small groups of objects, contention and deadlock would become serious problems.

To summarize, the problems of deadlock are exacerbated in general-purpose transaction-based systems. Further research is needed to examine the applicability of traditional solutions, and to determine the tradeoffs among those solutions in this environment. This research may yield variations on the traditional solutions, or demonstrate the need for new algorithms specifically designed for shared abstract types. For an example of a new approach to deadlock avoidance, see Korth's hierarchical variable-granularity locking protocol [Korth 81], which uses *edge locks*.

5. Recovery

Recovery is the process of restoring consistency after a failure. Recovery properties can be used like synchronization properties to classify types, and different recovery techniques are appropriate for different classes of types.

Some types have operations that are uninvertible. Gray has called such types *real* [Gray 80], because their operations correspond to events in the "real" world that are either unrepeatable or irreversible. An operation that causes a banking terminal to dispense cash is an example of an uninvertible update. These operations must be deferred until the invoking transaction commits.

Other types can be characterized by two properties of their operations: failure atomicity and permanence. Failure-atomic operations are always undone upon transaction abort, and if all operations in a transaction are failure-atomic then the entire transaction will be failure-atomic. Failure-atomic operations must be undone both when transactions abort during normal processing and when transactions are interrupted by failures. After a failure, recovery must identify and then abort any transactions that were in progress. Operations that are not failure-atomic are useful for implementing hints efficiently. As discussed in Section 3, incorrect hints do not cause fatal errors or loss of consistency.

Permanent operations are never undone once a transaction has committed. Guaranteeing permanence, unlike failure atomicity, requires that the system store some information in a failure-resilient manner. This is potentially expensive, and there are many types that do not need to survive failures. The cost of reconstructing an object's state from other information after a failure can be less than the continued cost of ensuring permanence for each operation. Operations that are non-permanent but failure-atomic are useful for preserving consistency of objects that can be discarded after failures, but should remain consistent when aborts occur during normal processing.

Underlying any recovery mechanism is an abstract model for failures. Lampson has developed a model that distinguishes between two kinds of failures: errors and disasters [Lampson 81]. Under this model, one of the purposes of recovery is to mask the undesirable properties of real system components by providing new, better-behaved abstract components. These *stable* components function identically to their real counterparts, except that they are not subject to errors. However, stable components remain vulnerable to disasters. By distinguishing between these two kinds of incorrect behavior, the model encourages a clear delineation of the failures that recovery must handle successfully.

For example, reading or writing detectably incorrect data is a storage error, as is *media failure*: the "infrequent" spontaneous decay of correct data. However, reading or writing undetectably corrupted data is a storage disaster. Unlike real storage, *stable storage* always reads and writes data correctly unless a disaster happens. There are several ways to implement stable storage, including duplexed disk or error-correcting RAM with a backup power source.

Incorrect behavior by processors can similarly be classified as erroneous or disastrous. If a processor detects an inconsistency and "crashes" by resetting itself and the system's volatile memory to a standard state, the behavior is considered to be an error. If an inconsistency slips by undetected, then a disaster has taken place. Stable processors that recover from crashes can be built using stable storage to save processor state.

Stable storage gives programmers the ability to make atomic modifications to disk pages or other small, fixed-size units of data. To provide types with failure-atomic or permanent operations, the properties of stable storage must be used to implement atomic modification of arbitrary collections of data. Database systems frequently use *logging* [Gray 78, Gray 81c, Lindsay 79] to achieve failure atomicity and permanence of transactions. We will briefly summarize this technique and consider its suitability for implementing shared abstract types with these properties.

Unlike "shadow" techniques [Lorie 77, Lampson 81], in which transactions manipulate temporary copies of objects, logging allows transactions to modify objects in place. Furthermore, objects can be transferred between volatile storage (which does not survive processor errors) and non-volatile storage in a way that is independent of transaction commitment. Thirdly, when logging is used, objects themselves do not have to be stored in stable storage. To permit the restoration of consistency if a failure occurs, transactions append information to a *log* in stable storage as they execute. Because objects are modified in place, the following types of inconsistency can be present after a failure:

- Some objects that committed transactions have modified may not have been copied to non-volatile storage prior to the failure. The log must contain sufficient information to redo those modifications during recovery.

- Some objects that incomplete (aborted) transactions have modified may have been copied to non-volatile storage prior to the failure. The log must contain sufficient information to undo those modifications during recovery.

- A media failure may detectably damage the most recent copy of an object on non-volatile storage. The log must have sufficient information to restore the object's current state from an archived version.

Output of the log to stable storage must be coordinated with the commitment of transactions and with the movement of objects between volatile and non-volatile storage. A transaction may not commit until the information needed to redo its modifications has been written to the log. Likewise, a modified object cannot be migrated to non-volatile storage before the information necessary to undo the change has been recorded in the log. This tactic is often referred to as the Write Ahead Log protocol [Gray 78].

There are many ways to represent the required information in the log, but they all have one aspect in common. By definition, a log is a linear sequence of typed records that can only be modified by appending new records at the end. Log records can be read in any order.

Perhaps the simplest way to represent log information is by recording the old and new values of modified objects [Lindsay 79]. Old values can be used to undo aborted transactions; new values can be used to redo committed transactions. The limitations of this representation technique come from its close relation to synchronization policy. If the synchronization rules for an object permit concurrent modification by more than one incomplete transaction, it is frequently impossible to use the old value/new value log representation.

This limitation also applies to recovery techniques based on "shadow" copies.

An abstract type that implements a counter provides a simple example of this limitation. The abstract properties of a counter do not prohibit concurrent increment operations by multiple transactions, as long as the increment operation does not also return the counter's value. Suppose the counter has an initial value of 0. The first increment operation records an old value of 0 in the log. The second transaction records an old value of 1, setting the current value to 2. If the second transaction commits but the first transaction later aborts, restoring the first transaction's old value of 0 is incorrect.

The principles behind this argument can be formalized, and rigorous criteria for the applicability of this log representation can be specified. Our investigation thus far of synchronization for shared abstract types has indicated that there is a lot of concurrency to exploit without violating reasonable type-specific synchronization properties, and synchronization policies that take advantage of this concurrency will not always be compatible with old value/new value logging.

A second logging technique is based on recording transitions rather than old or new states [Gray 81c]. Appropriate inverse transitions can correctly and independently abort forward transitions. For the counter, transition logging records "Increment" for each transaction rather than the counter value. In this case, the inverse operation is to decrement the counter.

The limitations of the transition method come from the difficulty of constructing types with operations that are practical to invert. Sometimes it is difficult to know at the time the log record is written exactly what information will be needed to invert the operation later on. For instance, suppose the counter also offers a reset operation. If a reset occurs and later is aborted, the proper restored value for the counter depends not only on its value at the time of the reset, but also on the operations that have occurred since. Examining the log and redoing these intervening operations may be prohibitively expensive.

The cost and complexity of logging depends on a type's implementation as well as on its abstract properties. For instance, the logging algorithm for a set implemented as a bit vector is quite different from the logging algorithm for a linked-list implementation. Further research is needed to evaluate the power of existing algorithms. This research should lead to new or modified logging techniques that support recovery for a variety of types and implementation strategies.

The composability of types also complicates recovery. In a database, the records at the leaves of the hierarchy are critical. Files and indices serve only to organize this data, and their function is explicitly understood by the system. It is therefore appropriate to provide recovery facilities at the record level; the system can automatically correct any related file or index structures. In a system allowing general shared abstract types, it is more difficult to decide which operations should be permanent or failure-atomic and which should not. If one type is used in the implementation of another, the recovery behavior of the component type may not be appropriate in the larger context. Like synchronization properties, it is necessary to include recovery properties in the abstract specification for a type.

6. Communication

Communication systems aim to provide useful and efficient communication primitives. Though these goals are easy to state, individual communications systems attempt to meet them in different ways. The communication mechanism of a transaction-based system is used both for the inter-node operation calls that occur within transactions as well as for transaction management operations themselves. The latter group includes transaction initiation, transaction migration, commit coordination, and distributed deadlock detection. Though much is known about communication in transaction-based distributed databases [Lindsay 79, Gray 78], more general transaction-based systems have additional communication requirements and their communication systems must be the subject of more study.

The foremost of these requirements is high communication efficiency. General distributed systems may contain many brief transactions that execute frequently. In current distributed database systems, transactions last at least a few hundred milliseconds, because they perform reads or writes to secondary storage. Performance of the communication system is therefore not critical. General distributed systems, however, will use new types of low-latency stable storage, and very efficient communication is likely to be important. More frequent distributed deadlock detection may also be necessary, especially in real-time systems.

High availability also demands high communication efficiency. For example, frequent operations across node boundaries are required to maintain many data replicas. Communication efficiency can be increased by simplifying protocols, reducing cross-level context switching, and increasing hardware support for the communication system. Communication primitives and their implementations must take advantage of the properties of the underlying communication media and not rely on excessive protocol layering [Spector 82].

For instance, consider remote operation calls on a network. Assume that the network's error rate is low in comparison with the rate of occurrence of other errors such as deadlock. Though remote call primitives could be implemented with complex error-correction facilities, it is only necessary that these primitives have *at-most-once* semantics. That is, the communication system must prevent duplicated, corrupted, or out-of-order operation calls, but it need not guarantee that remote operations are actually executed [Liskov 82b]. If the communication medium is a typical local area network, those semantics can be provided efficiently. It is deliberately left to the transaction manager to detect and recover from other communication errors (e.g., lost messages) by causing a transaction abort. This is an application of Saltzer's "end-to-end" argument [Saltzer 81].

Other optimizations to transaction communication facilities include batching transmissions and using multicast. Batching can be used to transmit a group of updates that were deferred until commit time. Multicast can be used for the transmission of similar remote operations to multiple sites [Rowe 79], for example, when transactions access replicated data. For sufficiently reliable communications media, multicast messages can be sent without requesting acknowledgments. The error recovery facility of the transaction manager is responsible for recovering from communication errors.

Though eliminating functions from intermediate protocol levels can improve efficiency, there are some problems to consider. For example, flow control and security are often functions of intermediate-level protocols, and, when required, must instead be added to the high-level transaction protocols. Additionally, reflecting many communication errors back to the transaction manager can actually result in lower performance if relatively unreliable communication media are used.

Beyond added communication efficiency, more demanding transaction management operations may induce other new requirements. The transaction coordinator may require that the various nodes participating in a transaction agree to commit their operations *cotemporally*. This problem is relevant in real-time transaction processing where transactions must simultaneously activate several devices.

The cotemporal commit problem is described by Gray as the problem of *N* generals trying to agree, via exchange of messages along an unreliable path, on a time for simultaneous attack [Gray 78]. Protocols that can be used to solve it are analogous to 2-phase commit protocols but with an added constraint concerning the time the participants actually carry out the commit operation. For a communication medium that can lose messages, there is no protocol that guarantees that the participants will agree to commit cotemporally. However, protocols similar to the centralized 2-phase commit protocols are better than ones similar to the linear commit protocol, because the centralized protocol permits the parallel transmission of messages to the participants. This increased parallelism reduces the interval during which some participants may have agreed to commit at a certain time, whereas others have not yet been so informed.

Increased efficiency and cotemporal transaction commit are only two examples of requirements for communication systems that support general transaction mechanisms. Though such requirements are similar to those of distributed database systems, there are differences that must be studied further.

7. Summary

The goal of constructing a transaction kernel is to make transactions available as a fundamental programming construct for reliable distributed computing, thereby reducing the complexity of designing and implementing reliable distributed systems. There is considerable evidence that transactions free the programmer from continual reimplementation of complex synchronization and recovery code, and that they will be useful in the construction of distributed systems. This paper has suggested the possibility of building a transaction kernel to support transactions containing calls on user-definable shared abstract data types. It has also described important research questions, such as what modifications to the traditional transactional model will be necessary, and whether systems built using transactions will have acceptable efficiency.

We are attempting to answer these questions at Carnegie-Mellon, in an effort that overlaps with the Archons project and currently includes five researchers. Specifically, we are pursuing research on the topics indicated by the major section headings of this paper:

- Extensions to the transaction model and the overall structuring of distributed systems that utilize transactions, including the identification of useful shared abstract data types.
- The specification and lock-based implementation of synchronization for shared abstract types.
- The impact of high-concurrency shared abstract types on deadlock detection and resolution algorithms.
- The specification and implementation of recovery for shared abstract types.
- The fulfillment of communication requirements for systems utilizing transactions.

There are other issues concerning the general use of transactions, but this subset forms a good basis for research on extending their utility. We are not considering deadlock avoidance mechanisms, alternatives to lock-based synchronization, or incorporation of transactions into programming languages. Work that overlaps ours and also addresses some of these other topics is occurring elsewhere [Liskov 82b, Allchin 82]. When the results of present research on transactions become available, it should be possible to construct a transaction kernel that encourages more universal use of transaction-based programming.

Acknowledgments

We gratefully acknowledge the contributions of Dean Daniels and Cynthia Hibbard, who helped us organize and present the ideas in this paper.

References

[Allchin 82] James E. Allchin and Martin S. McKendry.
Object-Based Synchronization and Recovery.
1982.
School of Information and Computer Science, Georgia Institute of Technology, submitted for publication.

[Bernstein 81] P. A. Bernstein, N. Goodman, and M. Y. Lai.
Two Part Proof Schema for Database Concurrency Control.
In *Proc. Fifth Berkeley Wkshp. on Dist. Data Mgmt. and Computer Networks*, pages 71-84. February, 1981.

[Daniels 82] Dean Daniels.
Query Compilation in a Distributed Database System.
Master's thesis, MIT, March, 1982.

[Digital Equipment Corporation 72]
Decsystem10 Assembly Language Handbook
2 edition, Digital Equipment Corporation, Maynard, MA, 1972.

[Eswaran 76] K. P. Eswaran, James N. Gray, Raymond A. Lorie, I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Comm. of the ACM 19(11), November, 1976.

[Garcia-Molina 82]
H. Garcia-Molina.
Using Semantic Knowledge for Transaction Processing in a Distributed Database.
Technical Report 285, Department of Electrical Engineering and Computer Science, Princeton, June, 1982.

[Gifford 79] David K. Gifford.
Violet, an Experimental Decentralized System.
In *Proceedings I.R.I.A. Workshop on Integrated Office Systems.* Versailles, France, November, 1979.
Also available as Xerox Palo Alto Research Center Report CSL-79-12.

[Gray 75] J. N. Gray, R.A. Lorie, G. R. Putzolu, and I. L. Traiger.
Granularity of Locks and Degrees of Consistency in a Shared Data Base.
IBM Research Report RJ1654, IBM Research Laboratory, San Jose, Ca., September, 1975.

[Gray 78] James N. Gray.
Notes on Database Operating Systems.
In R. Bayer, R. M. Graham, and G. Seegmuller (editors), *Lecture Notes in Computer Science.* Volume 60: *Operating Systems - An Advanced Course,* pages 393-481. Springer-Verlag, 1978.
Also available as IBM Research Report RJ2188, IBM San Jose Research Laboratories, 1978.

[Gray 80] Jim Gray.
A Transaction Model.
IBM Research Report RJ2895, IBM Research Laboratory, San Jose, Ca., August, 1980.

[Gray 81a] Jim Gray.
The Transaction Concept: Virtues and Limitations.
In *Proc. of Very Large Database Conference,* pages 144-154. September, 1981.

[Gray 81b] Jim Gray, Pete Homan, Ron Obermarck, Hank Korth.
A Straw Man Analysis of Probability of Waiting and Deadlock.
IBM Research Report RJ3066, IBM Research Laboratory, San Jose, Ca., February, 1981.

[Gray 81c] James N. Gray, et al.
The Recovery Manager of a Data Management System.
ACM Computing Surveys (2):223-242, June, 81.

[Hoare 74] C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Comm. of the ACM 17(10):549-557, October, 1974.

[Jacobson 82] David M. Jacobson.
Transactions on Objects of Arbitrary Type.
Technical Report 82-05-02, University of Washington, May, 1982.

[Korth 81] Henry F. Korth.
A Deadlock-Free Variable Granularity Locking Protocol.
In *Proc. Fifth Berkely Wkshp. on Dist. Data Mgmt. and Computer Networks.* February, 1981.

[Korth 83] Henry F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1), Jaunary, 1983.

[Kung 79] H. T. Kung and C. H. Papadimitriou.
An Optimality Theory of Concurrency Control for Databases.
In *Proceedings of the 1979 SIGMOD Conference.* ACM, Boston, MA., May, 1979.

[Lampson 81] Butler W. Lampson.
Atomic Transactions.
In G. Goos and J. Hartmanis (editors), *Lecture Notes in Computer Science.* Volume 105: *Distributed Systems - Architecture and Implementation: An Advanced Course,* chapter 11pages 246-265. Springer-Verlag, 1981.

[Lin 82] Wen-Te K. Lin, Jerry Nolte.
Performance of Two Phase Locking.
In *Proceedings 7th Berkeley Conference on Distributed Data Management and Computer Networks.* February, 1982.

[Lindsay 79] Bruce G. Lindsay, et al.
Notes on Distributed Databases.
IBM Research Report RJ2571, IBM Research Laboratory, San Jose, Ca., July, 1979.

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

[Liskov 82b] Barbara Liskov.
On Linguistic Support for Distributed Programs.
IEEE Trans. on Software Engineering SE-8(3):203-210, May, 1982.

[Lomet 77] David B. Lomet.
Process Structuring, Synchronization, and Recovery Using Atomic Actions.
ACM SIGPLAN Notices 12(3), March, 1977.

[Lorie 77] Raymond A. Lorie.
Physical Integrity in a Large Segmented Database.
ACM Trans. on Database Systems 2(1):91-104, March, 1977.

[Moss 81] J. Eliot B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, MIT, April, 1981.

[Reed 78] David P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, MIT, September, 1978.

[Rosenkrantz 78] D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis.
System Level Concurrency Control for Distributed Databases.
ACM Trans. on Database Systems 3(2), June, 1978.

[Rowe 79]	Lawrence A. Rowe, Kenneth P. Birman. Network Support for a Distributed Data Base System. In *Proceedings 4th Berkeley Conference on Distributed Data Management and Computer Networks*, pages 337-352. August, 1979.
[Saltzer 81]	J. H. Saltzer, D.P. Reed, D.D. Clark. End-To-End Arguments in System Design. In *Proceedings 2nd International Conference on Operating Systems*, pages 519-512. Paris, France, April, 1981.
[Schwarz 82]	Peter M. Schwarz, Alfred Z. Spector. *Synchronizing Shared Abstract Types.* Carnegie-Mellon Report CMU-CS-82-128, Carnegie-Mellon University, Pittsburgh, PA, September, 1982.
[Sha 83]	Lui Sha, E. Douglas Jensen, Richard F. Rashid, J. Duane Northcutt. Distributed Co-operating Processes and Transactions. In *Proceedings ACM SIGCOMM Symposium.* 1983.
[Spector 82]	Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Comm. of the ACM* 25(4), April, 1982.
[Weihl 81]	William E. Weihl. Atomic Actions and Data Abstractions. 1981. MIT Laboratory for Computer Science.