TANDEMCOMPUTERS

The Transaction Concept: Virtues and Limitations

Jim Gray

Technical Report 81.3
June 1981
PN87602

The Transaction Concept: Virtues and Limitations

Jim Gray

June 1981

Tandem Technical Report 81.3

Tandem TR 81.3

The Transaction Concept: Virtues and Limitations

Jim Gray
Tandem Computers Incorporated
19333 Vallco Parkway, Cupertino CA 95014

June 1981

ABSTRACT: A transaction is a transformation of state which has the properties of atomicity (all or nothing), durability (effects survive failures) and consistency (a correct transformation). The transaction concept is key to the structuring of data management applications. The concept may have applicability to programming systems in general. This paper restates the transaction concepts and attempts to put several implementation approaches in perspective. It then describes some areas which require further study: (1) the integration of the transaction concept with the notion of abstract data type, (2) some techniques to allow transactions to be composed of sub-transactions, and (3) handling transactions which last for extremely long times (days or months).

CONTENTS

INTRODUCTION: What is a transaction?

The transaction concept derives from contract law. In making a contract, two or more parties negotiate for a while and then make a deal. The deal is made binding by the joint signature of a document or by some other act (as simple as a handshake or nod). If the parties are rather suspicious of one another or just want to be safe they appoint an intermediary (usually called an escrow officer) to coordinate the commitment of the transaction.

The Christian wedding ceremony gives a good example of such a contract. The bride and groom "negotiate" for days or years and then appoint a minister to conduct the marriage ceremony. The minister first asks if anyone has any objections to the marriage; he then asks the bride and groom if they agree to the marriage. If they both say "I do", he pronounces them man and wife.

Of course, a contract is simply an agreement. Individuals can violate it if they are willing to break the law. But legally, a contract (transaction) can only be annulled if it was illegal in the first place. Adjustment of a bad transaction is done via further compensating transactions (including legal redress).

The transaction concept emerges with the following properties:

Consistency: the transaction must obey legal protocols.

Atomicity: it either happens or it does not; either all are bound by the contract or none are.

Durability: once a transaction is committed, it cannot be abrogated.

A GENERAL MODEL OF TRANSACTIONS

Translating the transaction concept to the realm of computer science, we observe that most of the transactions we see around us (banking, car rental, or buying groceries) may be reflected in a computer as transformations of a system state.

A system state consists of records and devices with changeable values. The system state includes assertions about the values of records and about the allowed transformations of the values. These assertions are called the system consistency constraints.

The system provides actions which read and transform the values of records and devices. A collection of actions which comprise a consistent transformation of the state may be grouped to form a transaction. Transactions preserve the system consistency constraints — they obey the laws by transforming consistent states into new consistent states.

Transactions must be atomic and durable: either all actions are done and the transaction is said to commit, or none of the effects of the transaction survive and the transaction is said to abort.

These definitions need slight refinement to allow some actions to be ignored and to account for others which cannot be undone. Actions on entities are categorized as:

Unprotected: the action need not be undone or redone if the transaction must be aborted or the entity value needs to be reconstructed.

Protected: the action can and must be undone or redone if the transaction must be aborted or if the entity value needs to be reconstructed.

Real: once done, the action cannot be undone.

Operations on temporary files and the transmission of intermediate messages are examples of unprotected actions. Conventional database and message operations are examples of protected actions. Transaction commitment and operations on real devices (cash dispensers and airplane wings) are examples of real actions.

Each transaction is defined as having exactly one of two outcomes: committed or aborted. All protected and real actions of committed transactions persist, even in the presence of failures. On the other hand, none of the effects of protected and real actions of an aborted transaction are ever visible to other transactions.

Once a transaction commits, its effects can only be altered by running further transactions. For example, if someone is underpaid, the corrective action is to run another transaction which pays an additional sum. Such post facto transactions are called compensating transactions.

A simple transaction is a linear sequence of actions. A complex transaction may have concurrency within a transaction; the initiation of one action may depend on the outcome of a group of actions. Such transactions seem to have transactions nested within them, although the effects of the nested transactions are only visible to other parts of the transaction (see Figure 1).

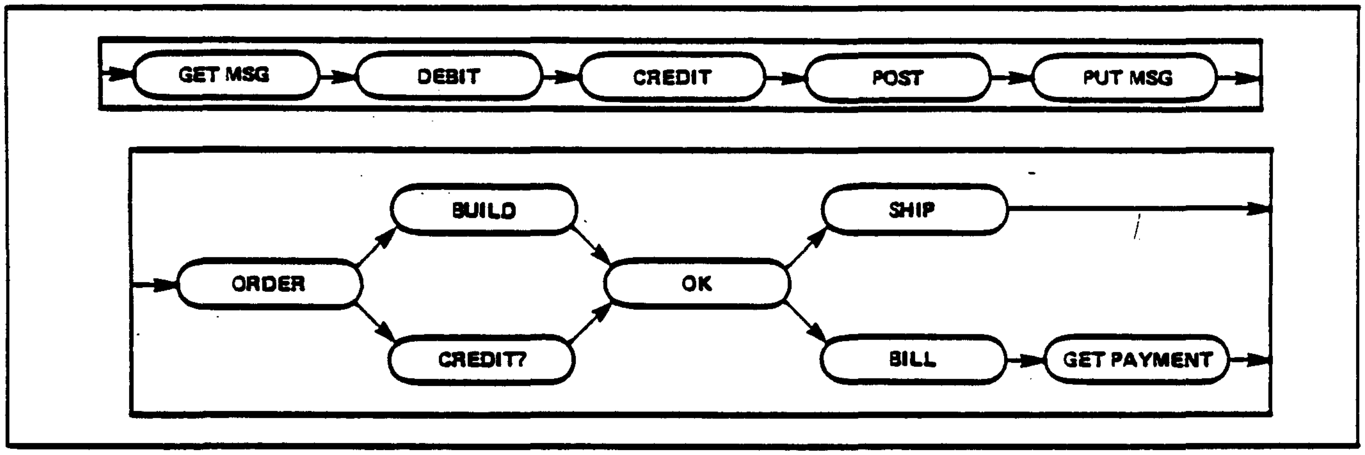

Figure 1. Two transactions. T1 is a simple sequence of actions. T2 is a more complex transaction which demonstrates parallelism and nesting within a transaction.

NonStop™: Making failures rare

One way to get transaction atomicity and durability is to build a perfect system which never fails. Suppose you built perfect hardware which never failed and software which did exactly what it was supposed to do. Your system would be very popular and all transactions would always be successful. But the system would fail occasionally because the people who adapted your system to their environment would make some mistakes (application programming errors) and the people who operated the system would make some mistakes (data entry and procedural errors). Even with very careful management, the system would fail every few months or years and at least one transaction in 100 would fail due to data-entry error or authorization error [Japan].

One may draw two conclusions from this:

1. You don't have to make a perfect system. One that fails once every thousand years is good enough to please most customers.
2. Even if the system is perfect, some transactions will abort because of data-entry error, insufficient funds, operator cancelation, or timeout.

This section discusses techniques for "almost perfect" systems and explains their relationship to transaction processing.

Imperfection comes in several flavors. A system may fail because of a design error or because of a device failure. The error may become visible to a user or redundant checks may detect the failure and allow it to be masked from the user.

A system is unreliable if it does the wrong thing (does not detect the error). A system is unavailable if it does not do the right thing within a specified time limit. Clearly, high availability is harder to achieve than high reliability.

John Von Neumann is credited with the observation that a very reliable (and available) system can be built from unreliable components [Von Neumann]. Von Neumann's idea was to use redundancy and majority logic on a grand scale (20,000 wires for one wire) in order to get mean-times-to-failure measured in decades. Von Neumann was thinking in terms of neurons and vacuum tubes which have mean-times-to-failures measured in days and which are used in huge quantities (millions or billions) in a system. In addition, Von Neumann's model was flat so that any failure in a chain broke the whole chain.

Fortunately, computer systems do not need redundancy factors of 20,000 in order to get very long mean-times-to-failure. Unlike Von Neumann's nerve nets, computer systems are hierarchically composed of modules and each module is self-checked so that it either operates correctly or detects its failure and does nothing. Such modules are called fail-fast. Fail-fast computer modules such as processors and memories achieve mean times to failure measured in months. Since relatively few modules make up a system (typically less than 100), very limited redundancy is needed to improve the system reliability.

Consider the simple case of discs. A typical disc fails about once a year. Failures arise from bad spots on the disc, physical failure of the spindle or electronic failure of the path to the disc. It takes about an hour to fix a disc or get a spare disc to replace it. If the discs are duplexed (mirrored) and if they fail independently then the pair will both be down about once every three thousand years. More realistic analysis gives a mean-time-to-failure of 800 years. So a system with eight pairs of discs would have an unavailable disc pair about once a century. Without mirroring the same system would have an unavailable disc about eight times a year.

Although duplexed discs have been used since the late sixties [Heistand], we have been slow to generalize from this experience to the observation that:

- Mean-time-to-failures of modules are measured in months.
- Modules can be made fail-fast: either they work properly or they fail to work.
- Spare modules give the appearance of mean time to repair measured in seconds or minutes.
- Duplexing such modules gives mean times to failure measured in centuries.

The systematic application of these ideas to both hardware and software produces highly reliable and highly available systems.

High availability requires rethinking many concepts of system design. Consider for example, the issue of system maintenance: one must be able to plug components into the system while it is operating. At the hardware level this requires Underwriters Laboratory approval that there are no high voltages around and requires that components tolerate high power drains and surges and, and, and.... At the software level this means that there is no "SYSGEN" and that any program or data structure can be replaced while the system is operating. These are major departures from most current system designs.

Commercial versions of systems which provide continuous service are beginning to appear in the marketplace. Perhaps the best known are the Tandem systems. Tandem calls its approach to high availability NonStop (a Tandem trademark). Their systems typically have mean times to failure between one and ten years. At the hardware level, modules and paths are duplexed and all components are designed for reliable and fail-fast operation [Katzman]. At the software level, the system is structured as a message-based operating system in which each process may have a backup process which continues the work of the primary process should the primary process or its supporting hardware fail [Bartlett1], [Bartlett2]. Alsberg proposed a related technique [Alsberg].

It is not easy to build a highly available system. Given such a system, it is non-trivial to program fault-tolerant applications unless other tools are provided; takeover by the backup process when the primary process fails is delicate. The backup process must somehow continue the computation where it left off without propagating the failure to other processes.

One strategy for writing fault-tolerant applications is to have the primary process "checkpoint" its state to the backup process prior to each operation. If the primary fails, the backup process picks up the conversation where the primary left off. Resynchronizing the requestor and server processes in such an event is very subtle.

Another strategy for writing fault-tolerant software is to collect all the processes of a computation together as a transaction and to reset them all to the initial transaction state in case of a failure. In the event of a failure, the transaction is undone (to a save point or to the beginning) and continued from that point by a new process. The backout and restart facilities provided by transaction management free the application programmer from concerns about failures or process pairs.

The implementors of the transaction concept must use the primitive process-pair mechanism and must deal with the subtleties of NonStop; but thereafter, all programmers may rely on the transaction mechanism and hence may easily write fault-tolerant software [Borr]. Programs in such a system look no different from programs in a conventional system except that they contain the verbs BEGIN-TRANSACTION, COMMIT-TRANSACTION and ABORT-TRANSACTION.

Use of the transaction concept allows the application programmer to abort the transaction in case the input data or system state looks bad. This feature comes at no additional cost because the mechanism to undo a transaction is already in place.

In addition, if transactions are implemented with logging, then the transaction manager may be used to reconstruct the system state from an old state plus the log. This provides transaction durability in the presence of multiple failures.

In summary, NonStop™ techniques can make computer systems appear to have failure rates measured in decades or centuries. In practice, systems have failure rates measured in months or years because of operator error (about one per year) and application program errors (several per year) [Japan]. These now become the main limit of system reliability rather than the software or hardware supplied by the manufacturer.

This section showed the need for the transaction concept to ease the implementation of fault-tolerant applications. There are two apparently different approaches to implementing the transaction concept: time-domain addressing and logging plus locking. The following sections explain these two approaches and contrast them.

To give a preview of the two techniques, logging clusters the current state of all objects together and relegates old versions to a history file called a log. Time domain addressing clusters the complete history (all versions) of each object with the object. Each organization will be seen to have some unique virtues.

UPDATE IN PLACE: A poison apple?

When bookkeeping was done with clay tablets or paper and ink, accountants developed some clear rules about good accounting practices. One of the cardinal rules is double-entry bookkeeping so that calculations are self-checking, thereby making them fail-fast. A second rule is that one never alters the books; if an error is made, it is annotated and a new compensating entry is made in the books. The books are thus a complete history of the transactions of the business.

The first computer systems obeyed these rules. The bookkeeping entries were represented on punched cards or on tape as records. A run would take in the old master and the day's activity, represented as records on punched cards. The result was a new master. The old master was never updated. This was due in part to good accounting practices but also due to the technical aspects of cards and tape: writing a new tape was easier than re-writing the old tape.

The advent of direct access storage (discs and drums) changed this. It was now possible to update only a part of a file. Rather than copying the whole disc whenever one part was updated, it became attractive to update just the parts that changed in order to construct the new master. Some of these techniques, notably side files and differential files [Severence] did not update the old master and hence followed good accounting techniques. But for performance reasons, most disc-based systems have been seduced into updating the data in place.

TIME-DOMAIN ADDRESSING: One solution

Update-in-place strikes many systems designers as a cardinal sin: it violates traditional accounting practices which have been observed for hundreds of years. There have been several proposals for systems in which objects are never altered; rather an object is considered to have a time history and object addresses become <name,time> rather than simply name. In such a system, an object is not "updated"; it is "evolved" to have some additional information. Evolving an object consists of creating a new value and appending it as the current (as of this time) value of the object. The old value continues to exist and may be addressed by specifying any time within the time interval that value was current. Such systems are called "time-domain addressing" or "version-oriented systems". Some call them immutable object systems but I think that is a misnomer since objects do change values with time.

Davies and Bjork proposed an implementation for time-domain addressing as a "general ledger" in which each entity had a time sequence of values [Davies], [Bjork]. Their system not only kept these values but also kept the chain of dependencies so that if an error was discovered, the compensating transaction could run and the new value could be propagated to each transaction that depended on the erroneous data. The internal book-keeping and expected poor performance of such a system discouraged most who have looked at it. Graham Wood at University of Newcastle showed that the dependency information grows exponentially [Wood].

Dave Reed has made the most complete proposal for a transaction system based on time-domain addressing [Reed1]. In Reed's proposal an entity E has a set of values Vi each of which is valid for a time period. For example the entity E and its value history might be denoted by:

E: <V0,[T0,T1)>, <V1,[T1,T2)>, <V2,[T2,*)>

meaning that E had value V0 from time T0 up to T1, at time T1 it got value V1 and at time T2 it got value V2 which is the current value. Each transaction is assigned a unique time of execution and all of its reads and writes are interpreted with respect to that time. A transaction at time T3 reading entity E gets the value of the entity at that time. In the example above, if T3>T2 then the value V2 will be made valid for the period [T2,T3). A transaction at time T3 writing value V3 to entity E starts a new time interval:

E: <V0,[T0,T1)>,<V1,[T1,T2)>,<V2,[T2,T3)>, <V3,[T3,*)>

If T2 >= T3 then the transaction is aborted because it is attempting to rewrite history.

The writes of the transaction all depend upon a commit record. At transaction commit, the system validates (makes valid) all of the updates of the transaction. At transaction abort the system invalidates all of the updates. This is done by setting the state of the commit record to commit or abort and then broadcasting the transaction outcome.

This is a simplified description of Reed's proposal. The full proposal has many other features including a nested transaction mechanism. In addition, Reed does not use "real" time but rather "pseudo-time" in order to avoid the difficulties of implementing a global clock. See [Reed2] and [Svobodova] for very understandable presentations of this proposal.

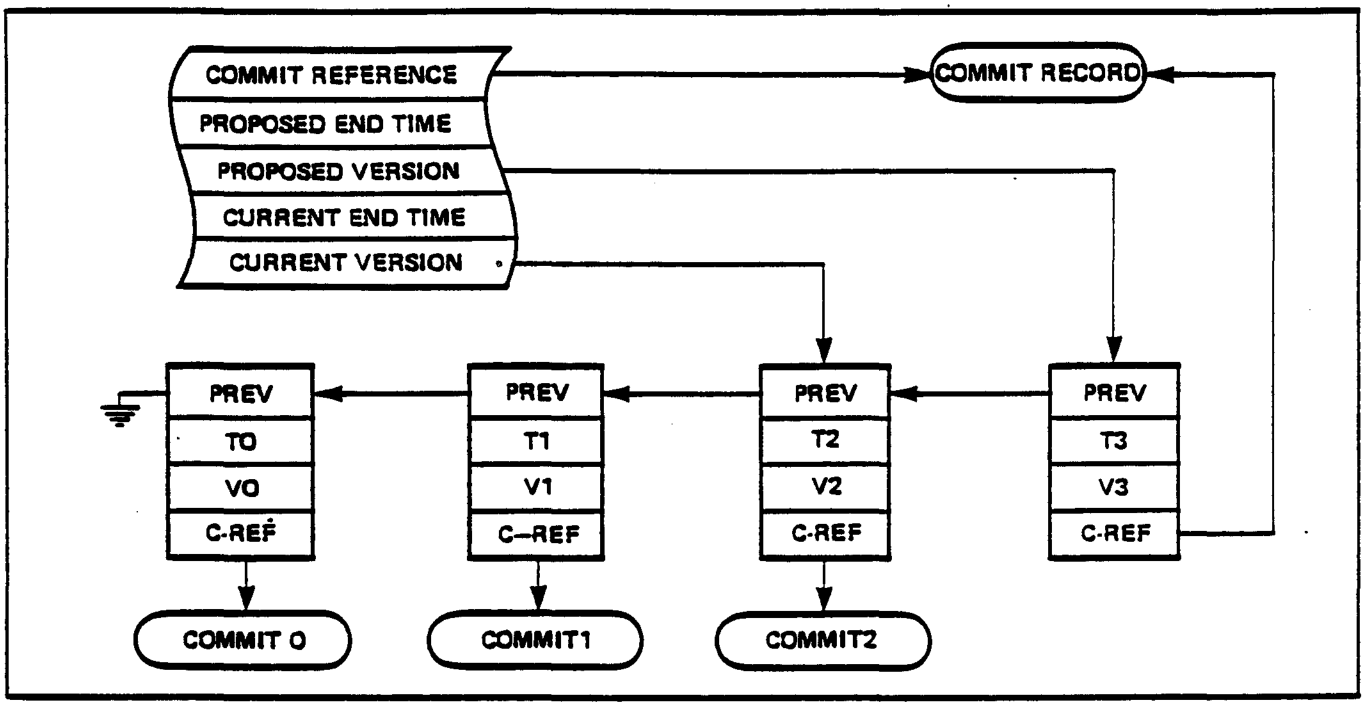

Figure 2. Representation of versions of object E. Three committed versions and one proposed version are shown. When version V3 is committed or aborted, the commit record and object header will be updated. Adapted from [Svobodova].

Reed observes that this proposal is a unified solution to both the concurrency control problem and the reliability problem. In addition, the system allows applications the full power of time domain addressing. One can easily ask questions such as "What did the books look like at year-end?"

There are some problems with time-domain addressing proposals.

1. Reads are writes: reads advance the clock on an object and therefore update its header. This may increase I/O activity.
2. Waits are aborts: In most cases a locking system will cause conflicts to result in one process waiting for another. In time-domain systems, conflicts abort the writer. This may preclude long-running "batch" transactions which do many updates.
3. Timestamps force a single granularity: reading a million records updates a million timestamps. A simple lock hierarchy allows sequential (whole file) and direct (single record) locking against the same data (at the same time.)
4. Real operations and pseudo-time: If one reads or writes a real device, it is read at some real time or written at some real time (consider the rods of a nuclear reactor, or an automated teller machine which consumes and dispenses money). It is unclear how real time correlates with pseudo-time and how writes to real devices are modeled as versions.

As you can see from this list, not all the details of implementing a time domain addressing system have been worked out. Certainly the concept is valid. All but the last issue are performance issues and may well be solved by people trying to build such systems. Many people are enthusiastic about this approach, and they will certainly find ways to eliminate or ameliorate these problems. In particular, Dave Reed and his colleagues at MIT are building such a system [Svobodova].

LOGGING AND LOCKING: Another solution

Logging and locking are an alternative implementation of the transaction concept. The legendary Greeks, Ariadne and Theseus invented logging. Ariadne gave Theseus a magic ball of string which he unraveled as he searched the Labyrinth for the Minotaur. Having slain the Minotaur, Theseus followed the string back to the entrance rather than remaining lost in the Labyrinth. This string was his log allowing him to undo the process of entering the Labyrinth. But the Minotaur was not a protected object so its death was not undone by Theseus' exit.

Hansel and Gretel copied Theseus' trick as they wandered into the woods in search of berries. They left behind a trail of crumbs which would allow them to retrace their steps by following the trail backwards and would allow their parents to find them by following the trail forwards. This was the first undo and redo log. Unfortunately, a bird ate the crumbs and caused the first log failure.

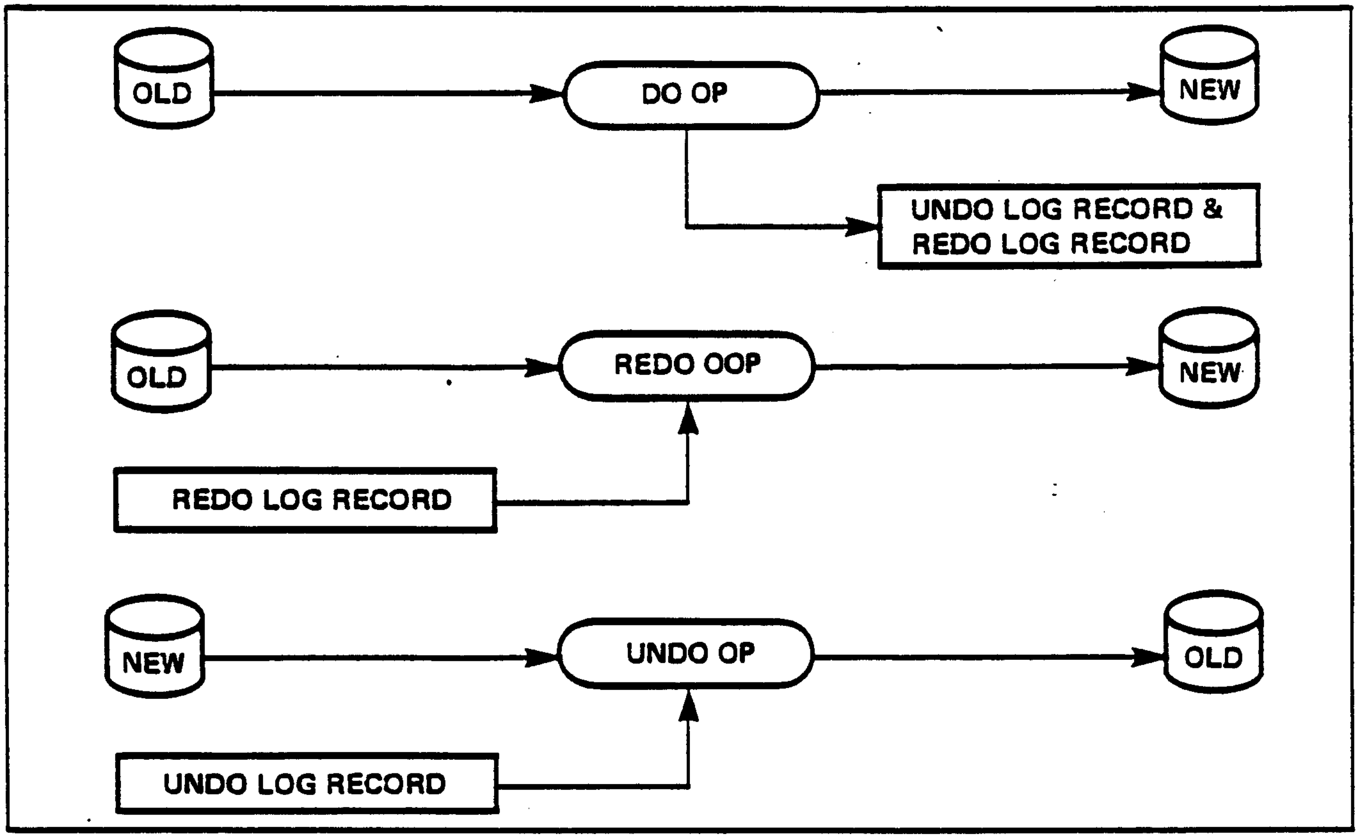

Figure 3. The DO-UNDO-REDO protocol. The execution of each protected action generates a log record which allows the action to be undone or redone. Unprotected actions need not generate log records. Actions which are not undoable (called real actions) use a related but slightly different protocol (see next figure).

The basic idea of logging is that every undoable action must not only do the action but must also leave behind a string, crumb or undo log record which allows the operation to be undone. Similarly, every redoable action must not only do the operation but must also generate a redo log record which allows the operation to be redone. Based on Hansel's and Gretel's experiences, these log records should be made out of strong stuff (not something a bird would eat). In computer terms, the records should be kept in stable storage—usually implemented by keeping the records on several non-volatile devices, each with independent failure modes. Occasionally, a stable copy of each object should be recorded so that the current state may be reconstructed from the old state.

The log records for database operations are very simple. They have the form:

NAME OF TRANSACTION:
PREVIOUS LOG RECORD OF THIS TRANSACTION:
NEXT LOG RECORD OF THIS TRANSACTION:
TIME:
TYPE OF OPERATION:
OBJECT OF OPERATION:
OLD VALUE:
NEW VALUE:

The old and new values can be complete copies of the object, but more typically they just encode the changed parts of the object. For example, an update of a field of a record of a file generally records the names of the file, record and field along with the old and new field values rather than logging the old and new values of the entire file or entire record.

The log records of a transaction are threaded together. In order to undo a transaction one undoes each action in its log. This technique may be used both for transaction abort issued by the program and for cleaning up after incomplete (uncommitted) transactions in case of a system problem such as deadlock or hardware failure.

In the event that the current state of an object is lost, one may reconstruct the current state from an old state in stable storage by using the redo log to redo all recent committed actions on the old state.

Some actions need not generate log records. Actions on unprotected objects (e.g. writing on a scratch file), and actions which do not change the object state (e.g. reads of the object) need not generate log records.

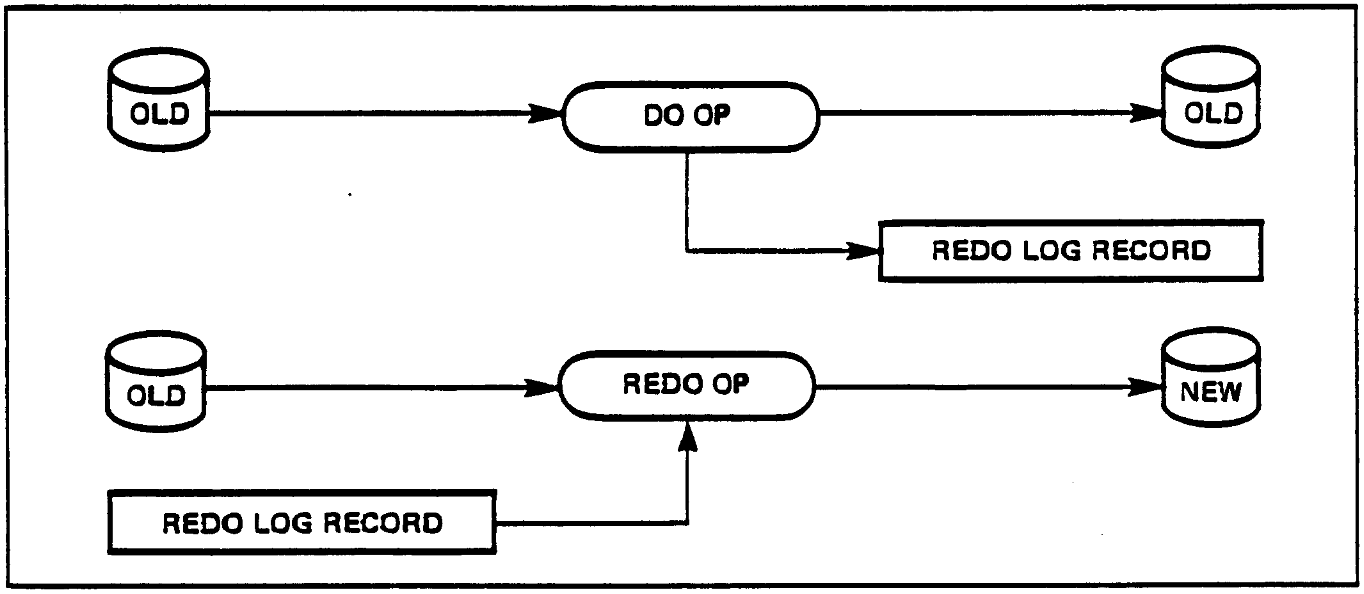

Figure 4. Real actions (ones that cannot be undone) must be deferred until commit. The logging approach to this is to apply the redo log of deferred operations as part of commit completion.

On the other hand, some actions must initially only generate log records which will be applied at transaction commit. A real action which cannot be undone must be deferred until transaction commit. In a log-based system, such actions are deferred by keeping a redo log of deferred operations. When the transaction successfully commits, the recovery system uses this log to do the deferred actions for the first time. These actions are named (for example by sequence number) so that duplicates are discarded and hence the actions are restartable (see below).

Another detail is that the undo and redo operations must be restartable, that is if the operation is already undone or redone, the operation should not damage or change the object state. The need for restartability comes from the need to deal with failures during undo and redo processing. Restartability is usually accomplished with version numbers (for disc pages) and with sequence numbers (for virtual circuits or sessions). Essentially, the undo or redo operation reads the version or sequence number and does nothing if it is the desired number. Otherwise it transforms the object and the sequence number.

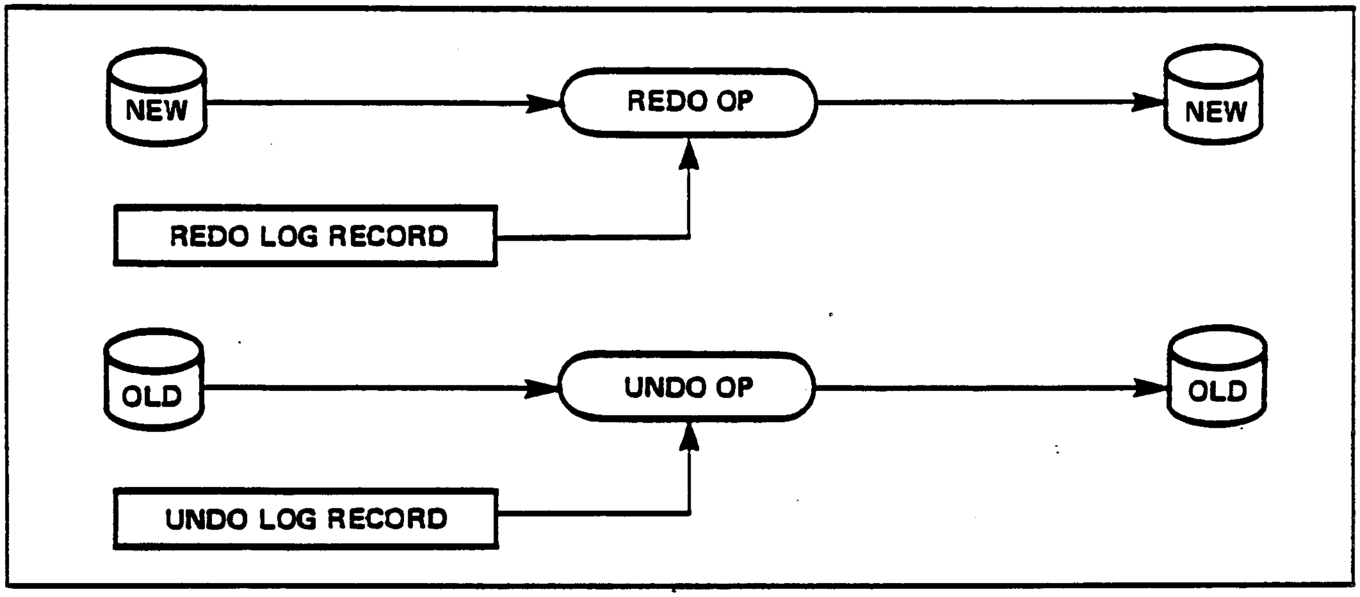

Figure 5. UNDO and REDO must be restartable, that is if the action has already happened, they must not alter the object.

In a log-based scheme, transaction commit is signaled by writing the commit record to the log. If the transaction has contributed to multiple logs then one must be careful to assure that the commit appears either in all logs or in none of the logs. Multiple logs frequently arise in distributed systems, since there are generally one or more logs per node. They also arise in central systems.

The simplest strategy to make commit an atomic action is to allow only the active node of the transaction to decide to commit or abort (all other participants are slaves and look to the active node for the commit or abort decision). Rosenkrantz, Sterns and Lewis describe such a scheme [Rosenkrantz].

It is generally desirable to allow each participant in a transaction to unilaterally abort the transaction prior to the commit. If this happens, all other participants must also abort. The two-phase commit protocol is intended to minimize the time during which a node is not allowed to unilaterally abort a transaction. It is very similar to the wedding ceremony in which the minister asks "Do you?" and the participants say "I do" (or "No way!") and then the minister says "I now pronounce you", or "The deal is off". At commit, the twophase commit protocol gets agreement from each participant that the transaction is prepared to commit. The participant abdicates the right to unilaterally abort once it says "I do" to the prepare request. If all agree to commit, then the commit coordinator broadcasts the commit message. If unanimous consent is not achieved, the transaction aborts. Many variations on this protocol are known (and probably many more will be published).

If transactions run concurrently, one transaction might read the outputs (updates or messages) of another transaction. If the first transaction aborts, then undoing it requires undoing the updates or messages read by the second transaction. This in turn requires undoing the second transaction. But the second transaction may have already committed and so cannot be undone. To prevent this dilemma, real and protected updates (undoable updates) of a transaction must be hidden from other transactions until the transaction commits. To assure that reading two related records or rereading the same record will give consistent results, one must also stabilize records which a transaction reads and keep them constant until the transaction commits. Otherwise a transaction could reread a record and get two different answers [Eswaran].

There appear to be many ways of achieving this input stability and hiding outputs. They all seem to boil down to the following ideas:

- A transaction has a set of inputs "I".
- A transaction has a set of outputs "O".
- Other transactions may read "I" but must not read or write "O".

Some schemes try to guess the input and output sets in advance and do set intersection (or predicate intersection) at transaction scheduling time to decide whether this transaction might conflict with some already executing transactions. In such cases, initiation of the new transaction is delayed until it does not conflict with any running transaction. IMS/360 seems to have been the first to try this scheme, and it has been widely rediscovered. It has not been very successful. IMS abandoned predeclaration (called "intent scheduling") in 1973 [Obermarck].

A simpler and more efficient scheme is to lock an object when it is accessed. This technique dynamically computes the I and O sets of the transaction. If the object is already locked, then the requestor waits. Multiple readers can be accommodated by distinguishing two lock modes: one indicating update access and another indicating read access. Read locks are compatible while update locks are not.

An important generalization is to allow locks at multiple granularities. Some transactions want to lock thousands of records while others only want to lock just a few. A solution is to allow transactions to issue a lock as a single predicate which covers exactly the records they want locked. Testing for lock conflict involves evaluating or testing membership in such predicates [Eswaran]. This is generally expensive. A compromise is to pick a fixed set of predicates, organize them into a directed acyclic graph and lock from root to leaf. This is a compromise between generality and efficiency [Gray1].

If a transaction T waits for a transaction T' which is waiting for T, both transactions will be stalled forever in deadlock. Deadlock is rare, but it must be dealt with. Deadlock must be detected (by timeout or by looking for cycles in the who-waits-for-whom graph), a set of victims must be selected and they must be aborted (using the log) and their locks freed [Gray1], [Beeri]. In practice waits seem to be rare (one transaction in 1000 [Beeri]) and deadlocks seem to be miracles. But it appears that deadlocks per second rise as the square of the degree of multiprogramming and as the fourth power of transaction size [Gray3], indicating that deadlocks may be a problem in the future as we see large transactions or many concurrent transactions.

SUMMARY

The previous sections discussed apparently different approaches to implementing the transaction concept: time-domain addressing and logging. It was pointed out that to make log operations restartable, the object or object fragments are tagged with version numbers. Hence, most logging schemes contain a form of time-domain addressing.

If each log record is given a time stamp, then a log can implement time-domain addressing. If Gretel had written a time on each crumb, then we could find out where they were at a certain time by following the crumbs until the desired time interval was encountered. Logging systems write the old value out to the log and so do not really discard old values. Rather, the log is a time-domain addressable version of the state and the disk contains the current version of the state.

Time-domain addressing schemes "garbage collect" old versions into something that looks very much like a log and they use locks to serialize the update of the object headers [Svobodova].

I conclude from this that despite the external differences between time domain addressing and logging schemes, they are more similar than different in their internal structure. There appear to be difficulties in implementing time-domain addressing. Arguing by analogy, Dave Reed asserts that every locking and logging trick has an analogous trick for time-domain addressing. If this is true, both schemes are viable implementations of transactions.

LIMITATIONS OF KNOWN TECHNIQUES

The transaction concept was adopted to ease the programming of certain applications. Indeed, the transaction concept is very effective in areas such as airlines reservation, electronic funds transfer or car rental. But each of these applications has simple transactions of short duration.

I see the following difficulties with current transaction models:

1. Transactions cannot be nested inside transactions.
2. Transactions are assumed to last minutes rather than weeks.
3. Transactions are not unified with programming languages.

NESTED TRANSACTIONS

Consider implementing a travel agent system. A transaction in such a system consists of:

1. Customer calls the travel agent giving destination and travel dates.
2. Agent negotiates with airlines for flights.
3. Agent negotiates with car rental companies for cars.
4. Agent negotiates with hotels for rooms.
5. Agent receives tickets and reservations.
6. Agent gives customer tickets and gets credit card number.
7. Agent bills credit card.
8. Customer uses tickets.

Not infrequently, the customer cancels the trip and the agent must undo the transaction.

The transaction concept as described thus far crumbles under this example. Each interaction with other organizations is a transaction with that organization. It is an atomic, consistent, durable transformation. The agent cannot unilaterally abort an interaction after it completes, rather the agent must run a compensating transaction to reverse the previous transaction (e.g., cancel reservation). The customer thinks of this whole scenario as a single transaction. The agent views the fine structure of the scenario, treating each step as an action. The airlines and hotels see only individual actions but view them as transactions. This example makes it clear that actions may be transactions at the next lower level of abstraction.

An approach to this problem which seems to offer some help is to view a transaction as a collection of:

- actions on unprotected objects
- protected actions which may be undone or redone
- real actions which may be deferred but not undone
- nested transactions which may be undone by invoking compensating transactions.

Nested transactions differ from protected actions because their effects are visible to the outside world prior to the commit of the parent transaction.

When a nested transaction is run, it returns as a side effect the name and parameters of the compensating transaction for the nested transaction. This information is kept in a log of the parent transaction and is invoked if the parent is undone. This log needs to be user-visible (part of the database) so that the user and application can know what has been done and what needs to be done or undone. In most applications, a transaction already has a compensating transaction so generating the compensating transaction (either coding it or invoking it) is not a major programming burden. If all else fails, the compensating transaction might just send a human the message "Help, I can't handle this".

This may not seem very satisfying, but it is better than the entirely manual process which is in common use today. At least in this proposal, the recovery system keeps track of what the transaction has done and what must be done to undo it.

At present, application programmers implement such applications using a technique called a "scratchpad" (in IMS) and a "transaction work area" in CICS. The application programmer keeps the transaction state (his own log) as a record in the database. Each time the transaction becomes active, it reads its scratchpad. This re-establishes the transaction state. The transaction either advances and inserts the new scratchpad in the database or aborts and uses the scratchpad as a log of things to undo. In this instance, the application programmer is implementing nested transactions. It is a general facility that should be included in the host transaction management system.

Some argue that nested transactions are not transactions. They do have some of the transaction properties:

Consistent transformation of the state

Either all actions commit or are undone by compensation

Once committed, cannot be undone

They use the BEGIN, COMMIT and ABORT verbs. But they do not have the property of atomicity. Others can see the uncommitted updates of nested transactions. These updates may subsequently be undone by compensation.

LONG-LIVED TRANSACTIONS

A second problem with the travel agent example is that transactions are suddenly long-lived. At present the largest airlines and banks have about 10,000 terminals and about 100 active transactions at any instant. These transactions live for a second or two and are gone forever. Now suppose that transactions with lifetimes of a few days or weeks appear. This is not uncommon in applications such as travel, insurance, government, and electronic mail. There will be thousands of concurrent transactions. At least in database applications, the frequency of deadlock goes up with the square of the multiprogramming level and the fourth power of the transaction size [Gray3]. You might think this is a good argument against locking and for time-domain addressing, but time-domain addressing has the same problem.

Again, the solution I see to this problem is to accept a lower degree of consistency [Gray2] so that only "active" transactions (ones currently in the process of making changes to the database) hold locks. "Sleeping" transactions (travel arrangements not currently making any updates) will not hold any locks. This will mean that the updates of uncommitted transactions are visible to other transactions. This in turn means that the UNDO and REDO operations of one transaction will have to commute with the DO operations of others (i.e. if transaction T1 updates entity E and then T2 updates entity E and then T1 aborts, the update of T2 should not be undone). If some object is only manipulated with additions and subtractions, and if the log records the delta rather than the old and new value, then UNDO and REDO may be made to commute with DO. IMS Fast Path uses the fact that plus and minus commute to reduce lock contention. No one knows how far this trick can be generalized.

A minor problem with long-running transactions is that current systems tend to abort them at system restart. When only 100 transactions are active and people are waiting at terminals to resubmit them this is conceivable (but not nice). When 10,000 transactions are lost at system restart then the old approach of discarding them all at restart is inconceivable. Active transactions may be salvaged across system restarts by using transaction save points: a transaction declares a save point and the transaction (program and data) is reset to its most recent save point in the event of a system restart.

INTEGRATION WITH PROGRAMMING LANGUAGES

How should the transaction concept be reflected in programming languages? The proposal I favor is providing the verbs BEGIN, SAVE, COMMIT and ABORT. Whenever a new object type and its operations are defined, the protected operations on that type must generate undo and redo log records as well as acquiring locks if the object is shared. The type manager must provide UNDO and REDO procedures which will accept the log records and reconstruct the old and new version of the object. If the operation is real, then the operation must be deferred and the log manager must invoke the type manager to actually do the operation at commit time. If the operation is a nested transaction, the operation must put the name of the compensating transaction and the input to the compensating transaction in the undo log. In addition, the type manager must participate in system checkpoint and restart or have some other approach to handling system failures and media failures.

I'm not sure that this idea will work in the general case and whether the concept of transaction does actually generalize to non-EDP areas of programming. The performance of logging may be prohibitive. However, the transaction concept has been very convenient in the database area and may be applicable to some parts of programming beyond conventional transaction processing. Brian Randell and his group at Newcastle have a proposal in this area [Randell]. The artificial intelligence languages such as Interlisp support backtracking and an UNDO-REDO facility. Barbara Liskov has been exploring the idea of adding transactions to the language Clu and may well discover a new approach.

SUMMARY

Transactions are not a new idea, they go back thousands of years. The idea of a transformation being consistent, atomic and durable is simple and convenient. Many implementation techniques are known and we have practical experience with most of them. However, our concept of transaction and the implementation techniques we have are inadequate to the task of many applications. They cannot handle nested transactions, long-lived transactions and they may not fit well into conventional programming systems.

We may be seeing the Peter Principle in operation here: "Every good idea is generalized to its level of inapplicability". But I believe that the problems I have outlined here (long-lived and nested transactions) must be solved.

I am optimistic that the transaction concept provides a convenient abstraction for structuring applications. People implementing such applications are confronted with these problems and have adopted expedient solutions. One contribution of this paper is to abstract these problems and to sketch generalizations of techniques in common use which address the problems. I expect that these general techniques will allow both long-lived and nested transactions.

ACKNOWLEDGMENTS

This paper owes an obvious debt to the referenced authors. In addition, the treatment of nested and long-lived transactions grows from discussions with Andrea Borr, Bob Good, Jerry Held, Pete Homan, Bruce Lindsay, Ron Obermarck and Franco Putzolu. Wendy Bartlett, Andrea Borr, Dave Gifford and Paul McJones made several contributions to the presentation.

REFERENCES

[Alsberg] Alsberg, P.A., J.D. Day, *A Principle for Resilient Sharing of Distributed Resources*, Proc. 2nd Int. Conf. on Software Engineering, IEEE 1976.

[Bartlett1] Bartlett, J.F., *A NonStop Operating System*, Eleventh Hawaii International Conference on System Sciences, 1978.

[Bartlett2] Bartlett, J.F., *A NonStop Kernel*, Proceedings of Eighth Symposium on Operating Systems Principles, ACM, 1981. (also Tandem TR 81.4).

[Beeri] Beeri, C., R. Obermarck, *A Resource Class Independent Deadlock Detection Algorithm*, Proceedings of Very Large Database Conference, 1981 (also IBM RJ-3077 (38123)).

[Bernstein] Bernstein, P.A., D.W. Shipman, J.B. Rothnie, *Concurrency Control in a System of Distributed Databases (SDD-1)*, ACM TODS V. 5, No. 1, 1980.

[Borr] Borr, A.J., *Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing*, Proceedings of Very Large Database Conference, 1981 (also Tandem TR 81.3).

[Davies] Davies C.T., Bjork L.A. private communication 1972.

[Bjork] Bjork, L.A., C.T. Davies, *The Semantics of the Preservation and Recovery of Integrity in a Data System*, IBM TR-02.540, 1972.

[Eswaran] Eswaran K.E., J.N. Gray, R.A. Lorie, I.L. Traiger, *On the Notions of Consistency and Predicate Locks*, CACM V. 19, No. 11, 1976.

[Gray1] Gray, J., *Notes on Database Operating Systems*, Operating Systems - An Advanced Course, Springer Verlag Lecture Notes in Computer Science, V. 60, 1978.

[Gray2] Gray, J., *A Transaction Model*, Automata Languages and Programming, Springer Verlag Lecture Notes in Computer Science, V. 80, 1980.

[Gray3] Gray, J., P. Homan, H. Korth, R. Obermarck, *A Strawman Analysis of Deadlock Frequency*. To be published in SIGOPS Review.

[Heistand] Heistand, R.E., *Airlines Control Program System, Concepts and Facilities*, IBM form number GH20-1473-1, 1975.

[Japan] Papers from the Tutorial on Reliable Business Systems in Japan, AFIPS Press, 1978.

[Katzman] Katzman, J.A., *A Fault-Tolerant Computing System*, Eleventh Hawaii International Conference on System Sciences, 1978.

[Obermarck] Obermarck R., *IMS/VS Program Isolation Feature*, IBM RJ2879 (36435), 1980.

[Randell] Randell, B., *System Structure for Fault Tolerance*, IEEE Trans. on Software Engineering, V. 1, No. 2, 1975.

[Reed1] Reed, D.P., *Naming and Synchronization in a Decentralized System*, MIT/LCS TR-205, 1978.

[Reed2] Reed, D.P., *Implementing Atomic Actions on Decentralized Data*, Proc. Seventh ACM/SIGOPS Symposium on Operating Systems Principles, 1979.

[Rosenkrantz] Rosenkrantz, D.J., R.D. Stearns, P.M. Lewis, *System Level Concurrency Control for Database Systems*, ACM TODS, V. 3, No. 2, 1977.

[Severence] Severence, D.G., G.M. Loman, *Differential Files: Their Application to Maintenance of Large Databases*, ACM TODS V. 1, No. 3, 1976.

[Svobodova] Svobodova, L. *Management of Object Histories in the Swallow Repository*, MIT/LCS TR-243, 1980.

[Von Neumann] Von Neumann, J. *Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components*, Automata Studies, Princeton University Press, 1956.

[Wood] Wood, W.G., *Recovery Control of Communicating Processes in a Distributed System*, U. Newcastle upon Tyne, TR-158, 1980.

Distributed by
TANDEMCOMPUTERS
Corporate Information Center
19333 Vallco Parkway MS3-07
Cupertino, CA 95014-2599