M.I.T. LABORATORY FOR COMPUTER SCIENCE July 2, 1979
Computer Systems Research Division Request for Comments No. 175

ATOMICITY AND COORDINATION

Term papers by Spring, 1979, 6.845 class members

Attached are six short term papers on the subject of atomic actions, recovery, and coordination of remote activities, prepared by students of a graduate seminar held this Spring. Comments may be directed to the original authors.

J. H. Saltzer

Atomicity and Reliable Data Abstractions

Maurice Herlihy
6.845 Term Paper
M.I.T. Dept of E.E. & C.S.
17 May 1979

Massachusetts Institute of Technology
Laboratory for Computer Science

6845 Term Paper 17 May 1979

Atomicity and Reliable Data Abstractions

Maurice Herlihy

1. Summary

This paper examines a number of schemes for implementing atomic actions in distributed systems, with particular attention given to the degree each supports construction of reliable data abstractions. Atomic actions associated with reliable data abstractions must have certain modularity properties beyond the usual atomicity properties. Of the schemes presented, only Reed [Reed] and Takagi [Takagi] provide the ability to construct atomic actions that satisfy these properties; Montgomery [Montgomery] and Gray [Gray] provide atomic actions that generally fail to satisfy the required modularity properties.

2. Vocabulary

An *atomic action* is one "that when it executes, no other operation can see or change the states of shared objects that it accesses" [Reed p80].

A *transaction* is an action that maps a data base with consistency constraints from one consistent state to another.

A *data abstraction* is "a named entity wholly characterized by its behavior in response to operations applied to it." [Reed p13]

Takagi [Takagi] provides the most complete terminology for object recovery techniques. Given an atomic action and an object, an *object-oriented* scheme for recovery is any scheme that can restore the object to the state it had at the start of the action. Takagi distinguishes *recoverable objects*, whose type managers provide recovery, from *non-recoverable objects*, whose users must provide recovery. For a given atomic action, a *process-oriented* scheme determines which objects' states were accessed by the action, and invokes the appropriate object-oriented scheme for each one.

3. Atomicity and Data Abstraction

At first glance, the concepts of atomicity and type abstraction appear somewhat similar. Both concepts involve the notion of hiding certain intermediate state information. Nevertheless, this similarity should not obscure an important difference between the two concepts: type abstraction is a tool intended to *facilitate* construction of complex systems, while atomicity is a *requirement* of many such systems. We would like to extend the ease of construction and maintenance associated with type abstractions to atomic actions by permitting construction of atomic actions in the same hierarchical manner as construction of data abstractions.

This leads to the definition of a *reliable data abstraction*: a data abstraction which provides atomic operations, which may be constructed hierarchically from other data abstractions and other reliable data abstractions. The hierarchical construction requirement means that a reliable data abstraction is not the same as a data abstraction with atomic operations. Atomic operations on a data abstraction must satisfy additional properties to form a reliable data abstraction.

3.1 Recovery Requirements

A failure of a reliable data abstraction operation on a component object must leave the higher level object in a well-defined state to allow it to take appropriate action. We take Reed's point of view [Reed p16], that if an operation on a reliable data abstraction fails, the effects of the operation should be completely undone, as this is the simplest way to define the effects of an arbitrary failure on an abstract state of an object.

All component objects of a reliable data abstraction must have object-oriented schemes to provide individual recoverability in the event of failure. Such a recovery scheme may be implemented by creating tentative versions, Takagi's backout cache, Gray's undo log entry, or the failed action may itself be atomic, with recovery provided at some lower level.

A failure of the higher level operation requires that a reliable data abstraction be able to undo any of its own incomplete effects, including the effects of successfully completed atomic component operations. This implies that a reliable data abstraction must provide a process-oriented recovery scheme for component objects. This recovery might be implemented by a form of dependent commit such as Reed's possibilities, or Gray's two-stage commit protocol.

3.2 Propagation of Changes

The other side of guaranteeing recovery is guaranteeing commit. A decision to commit by a nested type operation must make the operation's effects visible to the next highest level. For components with atomic actions, this reduces to forcing commit of successfully completed component operations. Methods for forcing commit of lower-level operations following commit at a higher level include Takagi's commit cache, Grey's redo log entries, and numerous schemes for forcing changes to stable storage.

3.3 Dependent Commit Requirement

In summary, operations on a reliable data abstraction must satisfy the following requirement:

> A successful component operation that is not undone by its containing operation is eventually committed if and only if the containing operation is eventually committed.

The techniques for forcing commit satisfy the "if" clause, and the recovery techniques satisfy the "only if" clause.

3.4 Atomicity

In order to make a reliable data abstraction's operations atomic, no other module should be able to observe the state of the object when some component actions have completed, but others have not.

3.5 Consistency Requirement

> The effects of an operation on a reliable data abstraction must be atomic with respect to other concurrent actions accessing the abstraction's component objects.

This can be viewed as a synchronization problem: in the absence of concurrency, recoverability is sufficient to provide atomicity. Note that this requirement is independent of the dependent commit requirement: atomicity requires that only final states be visible, commit dependency specifies what those final states may be.

3.6 Modularity Requirement

Finally, we need to insure the ability to hierarchically construct reliable data abstractions:

> The component objects of a reliable data abstraction may be reliable data abstractions.

In order to satisfy the commit dependency and modularity requirements, no reliable data abstraction may unilaterally commit any of its effects. (For simplicity, we are ignoring such secondary issues as memoization or metering.) The decision to commit must always be passed to a higher level, until the level from which the atomic action was originally invoked is reached (an application program?). Reliable data abstractions are thus an example of nested spheres of control [Bjork].

We shall see that the schemes that do not support reliable data abstractions fail to properly synchronize nested atomic actions. This leads to the conjecture that the problem of synchronization is the hardest problem involved in implementing reliable data abstractions.

4. Reed

Reed's thesis [Reed] presents a scheme whose explicit goal is to provide reliable data abstractions of the type discussed in this paper.

4.1 Recovery

The dependent commit requirement means only that component objects of a reliable data abstraction have associated recovery schemes, a weaker requirement than that they be recoverable. Both Reed and Takagi observe that non-recoverable objects are of limited use in constructing reliable data abstractions, although they cite different reasons. Reed observes that in a distributed system of autonomous nodes, the provider of a data abstraction may not trust the abstraction's users to properly recover the object after a failure. Takagi observes that cascading of backout requires the type manager to keep track of operations and transactions accessing its objects, a problem that does not arise in Reed's scheme.

At the level of pseudo-times and object versions, all Reed's objects are recoverable. Recoverability is provided by creating tokens, tentative versions of objects, which are thrown out in the event of a failure. Process-oriented recovery is implemented by possibilities, which keep track of all versions created by a given atomic action. Once an atomic action commits, all the tokens associated with its possibility are turned into versions.

5. Atomicity

Ordered pseudo-temporal environments are the mechanisms that provide synchronization. An operation has exclusive access to an object within a pseudo-temporal environment; If it successfully completes, an action accessing that object within a later environment will observe all of the changes made by the previous action; in any other case no changes will be visible.

5.1 Modularity

Dependent possibilities provide the mechanism that permits a successfully completed nested atomic action to pass the decision to commit to a higher level.

An atomic action may invoke a lower level action without having to be aware of the data it accesses, and without having to make any kind of special preparation (locking, pre-computation of conflicts).

The ability to construct pseudo-temporal environments that are subranges of given pseudo-temporal environments permit us to modularly compose atomic actions into larger atomic actions at each level.

6. Takagi

Takagi provides a scheme for a distributed system intended to maximize concurrent execution of conflicting atomic actions. Some minor clarifications are needed to support reliable data abstractions.

6.1 Recovery

Whereas Reed assumes that reliable data abstractions will be composed primarily from other reliable data abstractions, Takagi explicitly states that different object-oriented recovery schemes are appropriate at different levels. At a low level, where there is little or no concurrency, and information flow may be easily monitored, non-recoverable objects will incur less overhead and require a simpler implementation.

For higher level atomic actions, where objects may be distributed or concurrently accessed, the only practical place to put recovery information is in the underlying system or type manager, since none of the individual actions accessing the object are in a position to monitor dependency flow, or to become involved in recovery issues [p19].

Takagi introduces the notion of backout and commit caches as universal process-oriented recovery schemes. Backout and commit caches combine features of Gray's write-ahead log protocol and do-undo-redo paradigm for log records.

6.2 Commit Dependency

Dependent commits are implemented using a form of two-phase commit protocol, with the complication that complete but uncommitted results are released, and a decision not to commit may result in cascading of backout. The dependent commit requirement induces dependencies among data abstractions within a hierarchy; Takagi's multiple uncommitted versions introduce dependencies that cross hierarchy boundaries. Since data abstractions *per se* concern only relations within a type hierarchy, multiple uncommitted versions represent a refinement that does not really affect his system's ability to support reliable data abstractions.

6.3 Atomicity

All read and write requests in Takagi's system bear a timestamp. Access to objects is sequenced in timestamp order. Since all of the accesses performed by an atomic action bear the same timestamp, no other action may observe an intermediate state of any updated object.

We can naively nest atomic actions by allowing nested actions to inherit the timestamp used by the highest-level action. Although Takagi makes the simplifying assumption [p22] that a transaction never accesses the same object twice, this assumption seems less credible if we are to nest abstractly specified operations. Reed solves this same problem by introducing pseudo-temporal environments as an extension of pseudo-times. Takagi's scheme lends itself to the same approach. By representing timestamps as lists of integers, the way Reed represents pseudo-times, and by insuring that the top-level clock increments in discrete "ticks", we may use the interval between ticks as a pseudo-temporal environment. Transaction, paraction, next and current operations can be implemented just as with Reed's scheme.

6.4 Modularity

Takagi's timestamp synchronization scheme has the same modularity advantages as Reed's scheme, i.e. the ability to hide which data are accessed, and that higher level actions do not need to make any implementation-specific access preparations for lower levels.

Takagi explains in some detail how implicit dependent commits among multiple uncommitted versions are implemented (controlling the cascading of backout [p29]), however, he does not specify how an action may explicitly make its commit dependent on a higher level commit. Two simple schemes suggest themselves.

One could construct a hierarchical two-phase commit protocol, with each coordinator passing an "abort" or "ready to commit" message to its superior, until a decision is made at the very top level.

We might also make "prepared", but uncommitted object versions depend on commit records, using the machinery developed by Reed. When a subsequent uncommitted version examines the state of such a dependent version, the associated commit record could be polled. Conversely, when the commit record is set, it could inform any dependent versions.

7. Gray

The techniques that Gray outlines in [Gray] do not support reliable data abstractions because the locking scheme he uses does not correctly synchronize nested objects.

7.1 Recovery

In the absence of a type-specific recovery scheme, Gray provides a do-undo-redo paradigm for log records as a means to implement recoverable objects. Associated with each action on an object, a log entry is made with sufficient information to undo or redo the action performed.[p89] This strategy may be identified with Takagi's backup cache. In the event an atomic action encounters an error, the recovery manager at that node will use the redo entries in the log to restore the state of any modified objects.

7.2 Dependent Commit

Two mechanisms are provided to commit (or abort) a sequence of actions: a commit record, which is used to commit a number of actions performed at a single node, and the two-phase commit protocol, which serves to commit a group of actions performed at a number of nodes. Operations on objects are distinct from the operations that commit transactions: Gray assumes that none of the actions which commit together or abort together using these mechanisms themselves contain commit or abort actions. Nevertheless, in the absence of concurrency (a big if), it appears possible to extend the commit mechanisms to include dependent commit records resembling Reed's dependent possibilities.

At a single node, robustness in the face of node crashes is implemented by a reliable, stable log (another incarnation of atomic stable storage). A transaction is delimited by a BEGIN_TRANSACTION log entry, followed either by a COMMIT_TRANSACTION or ABORT_TRANSACTION entry. The commitment of the actions recorded between the two entries depends on whether the last is a commit or abort entry. Once a commit entry has been made, the crash recovery manager uses the redo entries in the log to insure that no results from any atomic actions that had been committed before the crash are lost. Gray provides a high water mark strategy to insure that redo entries are idempotent.

By introducing a dependent commit record, containing a transaction identifier, indicating that the contained actions are committed only if the indicated transaction commits, we may derive the same functionality as Reed's dependent possibilities for dependent atomic actions that take place at a single node.

In a distributed transaction, the two-phase commit protocol serves the same purpose as does one of Reed's dependent possibilities: it enables the decision whether to commit to be made by a higher level operation. By nesting commit coordinators as was suggested in the section on Takagi, the commit dependency requirement is easily satisfied.

7.3 Modularity and Atomicity

In the previous paragraph we saw that, by slightly modifying the machinery presented in [Gray], we can construct recoverable objects, but we shall see that a locking approach to synchronization completely precludes making operations on such objects atomic in the presence of concurrency. In order to maintain consistency, all atomic operations on objects must follow a two-phase lock protocol: no locks are acquired after any previously held locks are released. Furthermore, in order to permit undoing an action, all locks must be held until the end of the transaction. This prevents any sequence of atomic operations from being made atomic since all locks acquired by any operation on any component object must be acquired at the start of the transaction. Thus no operations that acquire and release their own locks may be nested within an atomic action. If we attempt to separate the acquisition and release of locks from the type operations, we hopelessly compromise the abstraction and modularity of our types.

In summary: in the absence of concurrency, we may nest atomic actions and build reliable data abstractions. In the presence of concurrency, we may nest objects with atomic operations, but the two phase lock protocol prohibits any meaningful degree of abstraction.

8. Montgomery

Montgomery's thesis [Montgomery] presents a scheme that does not support reliable data abstractions chiefly because atomic actions may not be hierarchically constructed.

8.1 Recovery and Modularity

Montgomery actually presents two schemes for implementing a distributed data base. The first specifies the effects of errors in rather a different way than any of the other schemes examined in this paper. The second scheme, presented as a refinement of the first, has more conventional error effects.

Robust sequenced messages and robust sequenced process steps are used to hide the effects of failures from transactions. Lost messages and node crashes are visible only in so far as their recovery may cause arbitrarily long delays. For this reason, transactions never fail, they may just take an unbounded amount of time to complete.

Since no transactions are ever aborted, no node's work ever needs to be undone, and the requirement to provide recovery schemes is vacuously fulfilled. The issue of recovery from internal failures is not addressed. In fact, Montgomery assumes that a node will always be able to come up with a correct output at the end of its process step. Without the ability to abort, an unexpected software error could destroy the atomicity of a transaction.

The completion of a process step may be indefinitely delayed by the failure of another node or by a communications failure. To avoid this, Montgomery refines his scheme by introducing "abortable locking", which allows a node to abort a transaction in which it is participating. In this model, one node, acting as a transaction coordinator, locks each node that will participate in the transaction. The decision to commit the transaction is reached by a standard two-phase commit protocol. If the transaction aborts, each participating node undoes its updates by restoring the process state it had at the start of the transaction. Again, by nesting the two-phase commit protocol, we can satisfy the commit dependency requirement.

8.2 Atomicity

The synchronization and deadlock avoidance aspects of Montgomery's scheme require knowledge of a transaction's activity graph - the locations of and dependencies between all the data read and updated. Message forwarders need to know of cycles in the union of the activity graphs of all the transactions using descendant nodes. If abortable locking is being used, the existence of cycles anywhere in the joint activity graph will require that all descendant nodes be locked, and that they prepare for a possible abort. All the nodes involved in a transaction must be known at the start, since a common ancestor must be chosen. Making an operation's activity graph part of its interface does not correspond to our idea of a data abstraction, as changing such implementation details as the location of data, forces changes in any containing operation. Thus, even when transactions never abort, Montgomery's scheme does not permit hierarchical construction of reliable data abstractions.

9. Conclusions

All of the schemes we have examined permit construction of recoverable objects, and, with minor extensions, permit construction of hierarchies of recoverable objects. Modular construction of recoverable objects seems to be easily provided.

When no objects are shared by concurrent processes, all the requirements for reliable data abstractions seem to be met. Gray and Montgomery provide synchronization schemes that require specifying all the data that an atomic action might access before that action is performed. The closer the data specified matches the data actually accessed, the better the system's performence. The activities required to insure atomicity of a number of actions are separate from those actions, and the information necessary to perform that synchronization is strongly dependent on specific details of those actions. In Reed's and Takagi's schemes, the synchronization activities performed by the higher level action (getting a timestamp or creating a pseudo-temporal environment), do not depend on details of the implementations of component actions. Modular construction of atomic actions appears to be much more difficult than modular construction of recoverable actions.

References

[Bjork] Bjork, L. A., "Recovery Scenario for a DB/DC System," Proceedings of the ACM National Conference 28, 1973.

[Gray] Gray, J. G., "Notes on Data Base Operating Systems," I.B.M. RJ2188 (30001), February 1978.

[Montgomery] Montgomery, W. A., "Robust Concurrency Control for a Distributed Information System," M.I.T. Technical Report 207, December 1978.

[Reed] Reed, D. P., "Naming and Synchronization in a Decentralized Computer System," M.I.T. Technical Report 205, September 1978.

[Takagi] Takagi, A., "Concurrent and Reliable Updates of Distributed Databases," M.I.T. Laboratory for Computer Science Request for Comments 167, November 1978.

Characterization of the Effects of Autonomy in a Decentralized Computing System

by Steven Krueger

May 18, 1979

A number of functional and psychological pressures towards decentralization of computer systems can be grouped together under the name of autonomy [d'Oliveira]. The purpose of this paper is to characterize the effects of autonomy of nodes in a distributed system into the desired, undesired expected, and undesired unexpected framework of Lampson and Sturgis [Lampson]. This is an attempt to build a coherent view of the effects of autonomy as failures.

The characteristic of an autonomous node of a distributed system is that it is managed by its local users and that local operations are more important to that management than network operations. Thus, a network of these nodes resembles a loose confederacy [Saltzer]. A node may refuse a network request because it is overloaded, because local work has a higher priority, for security or privacy, or for arbitrary administrative reasons. The autonomous node may accept a request but perform it improperly, due to either failures in the node or an autonomous decision giving the request unexpected semantics.

A further consequence of local management is that since even the acquisition of a node is localy managed, the nodes will be heterogeneous[Saltzer]. But, for the network to function each node must present a uniform interface to the network. This interface must provide a set of useful requests and replies with common semantics on all nodes. This much autonomy must be yielded to gain the benefits of the network. Thus the network structure of figure 1 is suggested. The network interface may be physically and logically separate from the node or it may be incorporated into the node.

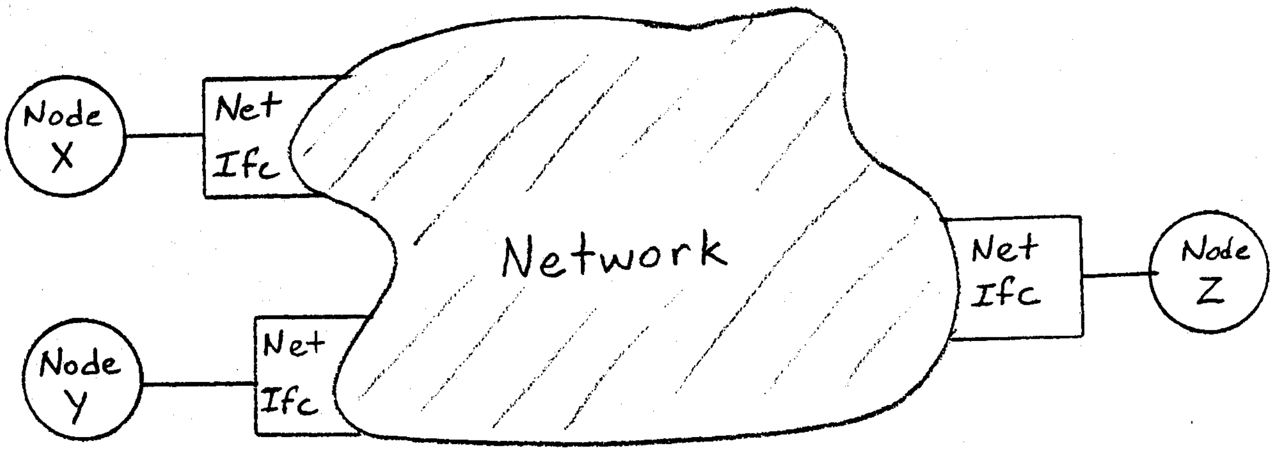


Figure 1 Network Structure

If a node sends a request to another node, several events are possible. For the requester, the desired event is that the semantics of the request are carried out by the destination node. To the requester any other event is undesired. The destination may ignore the request, possibly because it is disconnected from the network. This is the same as a lost message and is expected. This is a partition of the system. While disconnected, autonomy demands partial operability of both the isolated node and the rest of the distributed system [Montgomery]. On reconnection the system must not be inconsistent. In the case of greatest autonomy, no two nodes would agree to cooperate enough to replicate data, nor would any node trust another node to keep data for it. Since autonomy implies locality of reference, partial operability is achieved. If transactions involving inaccessible data abort rather than queue requests for the inaccessible nodes, consistency will be preserved.

Another possible event is that the destination node may refuse the request and indicate refusal with a reply. This seems appropriate if the node processes the request and finds it in violation of some local policy (security for example). To the requester, this is undesired but expected.

Another possible event is that the request is acted on but with the wrong semantics. If the reply makes the misunderstanding apparent then this can be expected (request is READ and reply is READY_TO_COMMIT). If the reply is not apparently improper, the event is unexpected (since it can not be readily recognized). Acting on a request without preserving its semantics can also be the result of some undetected and hence unexpected failure of the node.

For useful work to be done by the distributed system, the autonomy of the nodes must be further diminished. Each node must either act on a request according to its universally known semantics, refuse the request, or ignore it. Thus the loss of semantics of a request is an undesired and unexpected event whether the result of a node failure or an autonomous decision. Node failures that affect semantics may be limited to unexpected failures of the node by making the actions initiated by the request atomic and verifying the correctness of these actions.

Then, the events at the destination with their characterization to the requester are:

request acted upon	desired
request ignored	undesired expected
request refused	undesired expected
request semantics lost	undesired unexpected

It is interesting to note that because of autonomy all of these events except for lost semantics can be desired events for the destination node. Which is the desired event for the distributed system on a particular request cannot be resolved since there is no control or even objective for the system as a whole. In a system of autonomous nodes, it is the destination node that decides which event is desirable. The requester may only try a similar request at another node (if there is another node that has the resources needed to act on the request) or abort. The requester handles an expected event as it would an error.

The objective of this paper is to characterize the effects of autonomy in a distributed computer system. Some autonomy must be given up to provide a uniform interface to the network. The effects of autonomy at a node can be characterized as events that are desired, undesired expected, and undesired unexpected as suggested by Lampson and Sturgis for modeling physical systems. The undesired events should be handled by the requester as errors.

References

d'Olivereira, C. R., "An Analysis of Computer Decentralization", M. I. T. Laboratory for Computer Science Technical Memo TM-90 (October, 1977).

Lampson B. and Sturgis H., "Crash Recovery in a Distributed Data Storage System", to be published in Comm. of ACM.

Montgomery, W. A., "Robust Concurrency Control for a Distributed Information System", M. I. T. Laboratory for Computer Science Technical Report TR-207 (December 1978).

Saltzer J. H., "Research Problems of Decentralized Systems with Largely Autonomous Nodes", Operating Systems Review 12, 1 (January 1978) pp.43-52.

Comparisons of Notions of Atomicity and Commit

Alan M. Marcum

May 18, 1979

6.845 -- Topics in Computer Systems Research
Professor Jerome H. Saltzer

Comparisons of Notions of Atomicity and Commit

Two central issues in the class discussions of this term have been "atomicity" and "commit." Most of the papers we have read have defined what an "atomic" action is, and what it means to "commit" an action. In this paper, I shall examine the various notions of atomicity and commit, comparing the views of the various authors.

The specific authors to be included in this exposition are Bjork and Davies, Gray, Reed, Montgomery, Takagi, and Lampson and Sturgis. Each author's views will be described in turn, and then a comparison of the various points will be made.

Bjork and Davies

Bjork and Davies laid much of the early groundwork in this area. Consequently, many of their ideas are old and incomplete. Most of the papers on their work are very old (vintage 1972). Yet, even their recent paper (1978) is somewhat incomplete.

The Spheres of Control of Bjork and Davies are aimed primarily at recovery, auditing of updates, and resource allocation, and not so much with consistency and modularity. Their notion of atomicity is vague: they define an atomic process as "processing an operation code at a given level at a given instant." This seems to be merely a refinement to the notion of a procedural abstraction, which allows one level to view the operations of another level as simple, rather than as complex and compound. Their is no explicit notion of indivisibility in this definition of atomic process, except as that implied by the phrase "at a given instant." However, what is an "instant" at one level may be a very long time at another.

The concept of commit is much more sharply defined by Bjork and Davies than is that of atomic process. To commit a transaction is to cause the transaction to relinquish the ability to "unilaterally back out." The ability to "unilaterally back out" is to be able to abort the transaction without asking any other process or transaction if backing out is permissible.

Gray

Gray does not discuss atomic transactions per se. He does discuss consistency, recovery, scheduling, sequencing, and data accessing. His primary concern seems to be maintaining a consistent data base in whatever way he can, where consistency is defined in a quiescent state of the system, when all transactions have completed. To facilitate consistency, Gray defines a "consistency locking protocol," or "two-phase locking protocol," where a transaction holds all its locks until commit-point. He then says that this is more strict than is needed; all that is needed is that no locks be set after any lock is released.

There are cases where the less-strict form of the two-phase locking protocol produces non-atomic transactions. Suppose two transactions, T6 and T7, are running, and they access some of the same data. T6 sets various locks (including one on data object A), computes, releases the lock on A. Meanwhile, T7 has been working, and assume that is has been waiting for T6 to release the lock on A. Upon T6 releasing the lock, T7 reads A, and passes its value outside the system.

At this point -- after T6 releases the lock on A, after T7 reads A and passes its value outside, but before T6 completes -- the system crashes. Because T6 never committed, it will be aborted (UNDOne) rather than completed (REDOne). Because T7 passed the value of A outside the system, it has effectively committed. T7 has seen and acted upon an intermediate result of T6, which is NOT permanent, despite T6 adhering to the two-phase locking protocol.

Commit is the act of making the effects of a transaction permanent. Upon passing the commit point, the transaction is guaranteed to be completed. If there is a system failure between a transaction's commit point and the final completion of the transaction, the recovery manager will ensure that the transaction effectively runs to completion (by applying the REDO log).

Reed

Reed gives a two-part definition of atomic transaction: (i) it either is completely executed or not executed at all, and (ii) none of its intermediate states are visible to other transactions. Both of these ideas are important, although they do interact with one another.

Reed also discusses, "with respect to what are transactions atomic?" A transaction may be atomic respective of other transactions (corresponding to (i) above), or it may be atomic respective of system failures (corresponding to (ii) above), or both. Reed's definition of atomicity includes both other transactions and system failures.

Reed's scheme allows transactions to be atomic of their own volition; they need not depend on other transactions following the same (or some compatible) atomicity protocol. Regardless of the "hostility" of other transactions, if transaction A is supposed to be atomic, it is atomic.

Reed combines the concepts of atomicity and commit into what he calls "atomic commit." Atomic commit causes some set of actions to happen "simultaneously" as far as outside (of the transaction) observers are concerned. When the transaction is finished, it commits its results, and they all become visible at once.

Montgomery

Montgomery's definition of atomic transactions is equivalent to part (i) of Reed's definition: either all or none of the transaction's effects are visible to other transactions. The commit point in Montgomery's scheme is that point at which the polyvalues of the transaction can be evaluated.

Takagi

According to Takagi, an atomic action is indivisible and instantaneous to its users. This seems to correspond very closely with Bjork and Davies's definition of atomicity.

Takagi's definition of commit is very concrete. Upon passing the commit point, a transaction can no longer be backed out, because upon commit, all the recovery information (the UNDO information) is deleted. Once the commit point is reached, the transaction will be run -- eventually -- to completion.

Lampson and Sturgis

Lampson and Sturgis define an atomic transaction with regard to its effects. They assume that all "important" transactions -- those for which atomicity is desired -- perform writes to stable (permanent) storage. The define an atomic transaction as one all of whose writes are performed, or none of whose writes are performed. This definition is one of atomicity with regard to system failure. They state further that, in the presence of crashes, all the writes are completed once any of them is started.

Transactions have two phases: (i) compute, and write an "intentions" list (indicating what will be written to stable storage), and (ii) perform the writes recorded in the intentions list. Commit in the Lampson and Sturgis view is the last operation that occurs during the first phase of a transaction.

Comparison

The older papers do not have a well-defined notion of atomicity; that seems to have developed relatively recently. They do, however, have clear definitions of commit, and usually discuss in detail mechanisms for transaction commit. Atomicity is much better defined in the newer papers. I am not sure when the concept crystallized; it seems to have been before Reed's, though after Gray's, reports. The early version of the Lampson and Sturgis paper describes atomic transactions, and uses those words. Generally, the newer papers do not include much description about commit, commit points, or commit protocols; apparently, commit is now a well-defined concept.

Why have all these authors tried to define these concepts? Whether they defined the concepts or not, they have all had as a goal some form of system consistency in the face of multiple transactions, multiple accessors, conflicting updates and inquires. Even if atomicity was not well defined, consistency was. Usually, consistency was defined relative to some serial order of execution of the parallel transactions.

The papers of these authors show a progression of the theory of atomicity, recoverability, and consistency. From Bjork and Davies in 1972 to Reed and Montgomery in 1978 has been a long time. Lampson and Sturgis alone spent over a year just re-thinking some of their ideas. Still, the theory is not complete.

Modelling of Distributed Transaction Commit Protocols

by
SUNIL SARIN
May 17, 1979

Term Paper

M.I.T. Course 6.845 (Spring 1979)

("Atomicity, Recovery, and Coordination")

Table of Contents

1. INTRODUCTION

In this paper, I consider the problem of *atomically committing* transactions in a distributed system. There are several papers in the literature that address this problem [2, 9, 6, 10, 4, 8]. My intention in this paper is to construct a model for describing the general situation that arises with distributed transactions, such that it is possible to describe most existing algorithms as "special cases" of the model. By presenting a uniform model of this kind, I hope to identify the common ideas that appear in almost all distributed commit protocols and the common problems that many of them face. I also hope to use the model to highlight the important differences among the various protocols.

Sections 2 through 4 of this paper are devoted to motivating and developing the desired model. In Section 5, I examine specific distributed commit protocols from the literature from the viewpoint of this model. Section 6 summarizes the results of this study.

1.1. Assumptions about the Environment

I consider a *distributed system* to consist of a set of information processing *sites* connected together by a *communications network*. A *process* is an ongoing computation or information processing activity that is resident at a fixed *home site*. Individual sites may *fail* at arbitrary times, and when a failed site *recovers* it does so in a fixed "initial" state. There are two kinds of processes resident at a given site: (i) *volatile* processes, which are destroyed forever, not reappearing on recovery, if their home site fails; and (ii) *recoverable* processes, which do not vanish on failure of the home site. When a site recovers, it executes a *recovery procedure* that restores all recoverable processes that existed at the time of failure. Such a recoverable process can then resume its activities at a point dependent on what "state" information (see below) was saved before the failure.

Each site in the distributed system has access to a private *memory* that is not shared with other sites. A site's memory has two components: *volatile* storage, the contents of which are lost on site failure; and *stable* storage, the contents of which are preserved in the face of the site's failure. The concept of stable storage is a familiar one [6, 2]. Stable storage is used to hold important files or data, and also to implement recoverable processes through the recording of process *state information*.

I shall assume that, in addition to providing stable storage, a site also provides the ability to make a collection of writes *atomically* to stable storage; the collection of writes is atomic if it is either performed completely and correctly or not performed at all. Such an atomic write can be implemented using "intentions lists" [5] or any similar mechanism, and it will not be discussed any further.

The only means of communication between sites in the distributed system is via *messages* sent across the communications network. The network is characterized by variable and unpredictable message delays, and individual links in the network are subject to failure.

(Note that the ideas in this paper extend to any system that is physically centralized but logically "distributed", e.g., where there are independent subsystems participating in transactions. Each such subsystem can be considered to be an abstract "node" (or site) in an abstract "network", even though the nodes are physically resident on a single site and there is no physical communications "network".)

1.2. Atomic Transactions: Recovery and Synchronization

A *transaction* is a collection of *actions* (such as reading and writing of data) on the distributed system that is intended to be *atomic*. That is, the partial effects of the collection of actions should not be visible to other transactions. In this paper, I shall concentrate on one particular aspect of transaction atomicity, namely ***recoverability***: either all of a transaction's effects should take place, or none should.

It should be noted that transaction atomicity requires not only recoverability but *synchronization* as well, to ensure that transactions do not observe each other's partial effects. The problems of recoverability and synchronization are not totally decoupled, but many recoverability issues are somewhat independent of synchronization; for example, a synchronization scheme may be "correct" but may allow the possibility of one transaction waiting indefinitely for another transaction to complete or for a site in the system to recover. In this paper I shall treat the issue of recoverability only, ignoring the details of how transactions synchronize, e.g., using *locking* [2], "snapshots" consistent with transaction *timestamps* [9, 10], or waiting for updates from specific transaction "classes" [4]. The only assumption that I shall make about transaction synchronization is that any of a transaction's actions may be forced to *wait* or *abort* for synchronization reasons.

1.3. Scenario

The scenario that I assume for distributed transaction processing is illustrated in Figure 1. Basically, the following agents take part in a distributed transaction:

1. A *user U*, usually a human being at a terminal.

2. A process *C* that acts as transaction *coordinator.*

3. One or more *participant* processes P_1, ..., P_n

The processing of a transaction takes place in the following sequence of steps (similar to the steps described in [5]):

1. The user *U* enters a *start-transaction* request at some site in the system. A coordinator process is allocated to handle the transaction.

2. The user enters a set of *actions*, and the coordinator sets up participant processes P_1, ..., P_n at all necessary sites, to perform the specified actions. (It is irrelevant whether the sites involved are specified explicitly by the user or are chosen by the coordinator in response to a "high-level" request by the user.)

3. The user enters a *commit-transaction* request, and the coordinator *C* attempts to ensure that the transaction commits. If and when the coordinator is certain that all of the actions in the transaction will be "committed", i.e., completed and made permanent, the coordinator sends an *OK* message to the user.

(This above scenario is often slightly modified by having the user combine some or all of this requests to perform steps 1, 2, and 3. For example, the user might enter a single request to start, execute, and commit a transaction, or combine the *start-transaction* request with the requests for actions, and so on. These refinements should have little effect on our discussion.)

The role of the coordinator in this scenario is precisely that, i.e., to coordinate and control the processing of the transaction. All of the actual "work" involved in the transaction (reading and writing of data, computation, etc.) is performed by the participants. (If the transaction requires actions at the home site of the coordinator, the coordinator will create a separate participant process there.) On the other hand, the determination of when a transaction has committed (or aborted) is the sole responsibility of the transaction coordinator.

A distributed transaction may fail to commit for any of several reasons: *synchronization* (the presence of conflicting transactions may make it impossible for a participant to perform the actions requested of it, or may cause actions already performed to be aborted); *failures* of sites or communication links; the expiration of *timeouts*, set to protect against failures or excessive delays; or the receipt, by the coordinator, of an *abort-transaction* request from the user. If a transaction fails to commit, it should *abort* (i.e., it should appear as if none of the transaction's actions ever happened), and the coordinator process should send an *ABORTED* message to the user. In section 2, I shall examine what is required of a distributed commit protocol if it is to ensure that a transaction eventually commits or aborts and is not left in an inconsistent state in which some of its actions have committed and some have aborted.

2. CONSISTENCY AND CORRECTNESS OF DISTRIBUTED COMMIT PROTOCOLS

In this section I focus on the consistency and correctness of the eventual outcome of a distributed transaction and the response to the user, namely that a transaction should eventually either commit or abort, and the user should get a message (*OK* or *ABORTED*) consistent with this outcome. I shall defer till later sections the issues of how the user and/or the transaction coordinator decide that they are satisfied with the actions of the participants and that the transaction should be committed.

2.1. Criteria

The consistency and correctness criteria for a distributed transaction can be stated as follows (from an old version of Lampson and Sturgis' paper [5]):

Correct Response: If the user gets an *OK* message, then all participants either have committed or will commit their actions; if the user gets an *ABORTED* message, then all participants will abort their actions.

Consistency: It is never the case that one participant in a transaction commits its actions and another participant aborts its actions.

We note that these two requirements together are *not* sufficient to ensure that a transaction will eventually commit or abort. (In [5], Lampson and Sturgis added the requirements of *progress* and *termination* to capture this idea.) I shall defer these problems for the moment and examine below what kind of protocol is necessary for consistency and correct response.

2.2. Requirements of a Protocol

Here I present some properties that I believe a distributed transaction commit protocol must have if it is to satisfy the criteria of consistency and correct response. I shall present informal arguments in support of my claim that these properties are essential for distributed transactions in the kind of environment that I assume.

The properties that I believe a protocol must have are quite simple: (i) *recoverable* coordinator and participant processes obeying the *write-ahead-log* protocol; and (ii) the *two-phase* requirement, which in turn requires that participants have the ability to perform *tentative* actions that may be committed or aborted on command of the coordinator. These are described in turn below.

I. *Recoverable Processes.* Clearly, a participant process in a transaction should not vanish unless it has either committed all of its actions on behalf of the transaction or aborted all of them. Similarly, the coordinator should not vanish unless it has ensured that every participant has committed or every participant has aborted (and that the user receives the appropriate response). In order that this be true even in the presence of failures, it must be the case that the coordinator and participant processes are *recoverable* (i.e., not vanish on failure of their respective home sites), and that they each have a *COMMITTED* and an *ABORTED* state (with the obvious associated actions). If a process's home site fails while the process is in its *COMMITTED* or *ABORTED* state, then on recovery of the site the process should complete its actions of committing or aborting.

For the purposes of my "state model" (which is presented in Sections 3 and 4), I assume that each possible state S_C of a recoverable process has associated with it:

- A set of *actions* A which the process performs when in state S_c

- A *failure-state* S_f, which is the state the process should enter on recovery if its home site crashes. S_f may or may not be the same as S_c (it frequently will be), but in any case it must be one of the possible states of the process.

When a process is in state S_c, it may be considered to be executing a procedure S_c*-action* which has the general form:

```
procedure $S_c$-action:
            record state = $S_c$ on stable storage
            do actions $A$
            except when failure: $S_f$-action
    end
```

This syntax, based on CLU-like exception conditions [7] has been borrowed from Takagi [10]. I have extended Takagi's procedure format somewhat to include the explicit recording of the process state on stable storage before any of the actions are performed. This is the *write-ahead-log* protocol (which Gray introduced for updates to data [2]) applied to process states. If the process's site fails, then the recovery procedure will first retrieve the process state S_c from stable storage, and then execute the associated failure action S_f*-action*. If this protocol is not followed, the process may crash after executing some of the actions in A but before its state is recorded; it would then perform some other failure action (the one associated with the process's previous recorded state) on recovery.

Note that if the failure-state corresponding to a given process state S_c is the same state S_c, then the actions in A may be executed repeatedly if the process site fails. If A is "idempotent" (and it often is) then this should be all right. (Lampson and Sturgis [6] would describe such a procedure S_c*-action* as a *restartable action*.)

II. *Two-Phase Requirement.* This requirement is based on the following two observations:

1. The coordinator cannot correctly commit the transaction before it is certain that each participant is in a (recoverable) state in which it is capable of committing its local actions at the request of the coordinator. If this is not followed, the criterion of *correct response* may be violated, since the user may get an *OK* message (indicating that his transaction committed) but a

participant may not be able to commit its local actions. (We note that certain schemes, such as [9, 6] allow the coordinator or user to commit a transaction before it is certain that all of the requests to the participants have been performed and are committable. However, if a coordinator/user does this, then the unacknowledged requests may or may not get committed; thus, the user will not know exactly what the transaction is that is being committed. This is generally undesirable, and I shall not consider it; I assume that the coordinator and user are *well-behaved*.)

2. Until it is certain that a transaction is committed, a participant must be capable of aborting its local actions. If this is not followed, the criterion of *consistency* may be violated: If the participant commits its local actions before learning that the transaction will commit, then a participant elsewhere may have already aborted (or may eventually abort) its own local actions.

Since the decision to commit a transaction is centralized at the coordinator, and the only means of communication between the coordinator and the participants is through messages, we can state that: (i) the only way the coordinator can learn that a given participant is capable of committing its local actions is through a message (which I shall call the *DEPENDENT* message, for reasons to be presently made clear) that it receives from the participant; and (ii) the only way that a participant can learn that the transaction has committed is through a *COMMIT* message that it receives from the coordinator. Let us consider the following three events in the execution of a distributed transaction:

Event I:	Participant *P* sends a *DEPENDENT* message to the coordinator *C*, indicating that *P* is capable of committing its local actions.
Event II:	The coordinator *C* sends a *COMMIT* message to participant *P*.
Event III:	Participant *P* receives a *COMMIT* message from *C*.

Clearly, Event II must precede Event III (since one represents the sending of a message and the other represents the receipt of the same message). Also, due to requirement (1) above, Event I must precede Event II, and must therefore precede Event III as well. Now requirement (2) above states that at all times up to Event III, *P* must be capable of aborting its local actions. Since Event I precedes Event III, *P* must be capable of aborting its local actions at Event I as well (when it sends the *DEPENDENT* message indicating its ability to commit). Thus, at Event I, *P* must be capable of "going either way", i.e., committing or aborting its local actions. We shall say that at Event I *P* has performed its local actions on a *tentative* basis, i.e., such that it is possible to either commit or abort them.

From the above, we can conclude that a correct and consistent distributed transaction commit protocol must proceed in at least *two phases*:

1. Each participant performs its local actions on a tentative basis and informs the coordinator thereof.

2. On learning that each participant has performed its local actions on a tentative basis, the coordinator commits the transaction and instructs each participant to commit.

The above notion, of performing actions "tentatively", will be modelled by introducing a recoverable state for participants called *DEPENDENT*. I use the name *DEPENDENT* because once the participant has informed the coordinator of its ability to "go either way", it must await a message from the coordinator as to whether the transaction committed or aborted. The participant cannot abort its local actions of its own choice, since the coordinator may at any time send it a *COMMIT* message and the participant will have made a decision inconsistent with the coordinator; similarly the participant cannot commit its local actions of its own choice. Thus, the participant is "dependent" on the coordinator's instructions.

Since the *DEPENDENT* state implies the ability of the participant to commit its local actions if the coordinator so desires, the participant cannot enter this state unless it has recorded, on stable storage, sufficient information to be able to commit all of its local actions. This requirement can be satisfied using *REDO* logs [2] or "intentions lists" [5]. To describe this recording of commit information in the model, I shall say that a participant *writes its intentions* before it enters the *DEPENDENT* state. (Note also that before and during the *DEPENDENT* state, a participant must have the ability to *abort* its local actions, since it is not yet certain that the transaction will commit. This requirement can be met using "differential files" or *UNDO* logs [2].)

3. BASIC MODEL FOR DISTRIBUTED COMMIT PROTOCOLS

I next present a uniform "state model" for distributed transaction commit protocols. This model will be presented in two parts: (i) a *basic model*, which covers just the consistency and correctness issues raised in Section 2, and (ii) an *extended model*, which covers some additional issues not yet discussed. The basic model is the subject of this section; the extended model will be presented in Section 4.

3.1. State Model

The possible states for the coordinator and for each participant in the basic model are shown in Figure 2. We note the following about this state model:

* Each process has a *COMMITTED* state (abbreviated *C-COM* or *P-COM*) and an *ABORTED* state (abbreviated *C-ABORT* or *P-ABORT*), whose associated actions are:

  - Coordinator:
    C-COM: commit the transaction as a whole; all participants must be made to commit, and the user should get an *OK* message
    C-ABORT: abort the transaction

  - Participant:
    P-COM: local actions will be committed
    P-ABORT: local actions will be aborted

In the event of a failure in its *COMMITTED* or *ABORTED* state, a process will remain in that state. Thus, the actions above must be *idempotent*, e.g., repeated actions on data should have no effect, duplicate messages (such as the OK to the user) should be harmless, etc.

* Each process is initially in its *START* state (*C-START* or *P-START*). In the event of a failure during the *START* state, a process enters its *ABORT* state. We note that this is not strictly necessary for correctness; on recovery, a process could conceivably pick up where it left off. However, in almost every distributed commit protocol a failure in the *START* state causes the automatic aborting of actions performed so far, for reasons of performance. (An exception to this is Reed's scheme [9]. However, in Reed's scheme, every transaction has an associated *timeout* set, and if a site fails it is unlikely that this a timeout will not have expired by the time the failed site recovers.)

∗ The model includes certain *MISSING* states for each process. These are not truly states, but rather *pseudo-states* (or process *situations* in [5]) that correspond to the process being no longer in existence. There are two possible *MISSING* states for a process, depending on what the process did before vanishing:

- *MISSING-OK*: process *committed* before vanishing

- *MISSING-X*: process *aborted* before vanishing

These pseudo-states can in fact be considered to be real "states" of a process if the following constraints are satisfied:

- Once in a *MISSING* state, a process permanently remains in that state.

- Once in a *MISSING* state, a process performs no further actions and sends out no messages.

- If a process *A* is in a *MISSING* state, it is not possible for another process *B* to determine, within the system, which *MISSING* state *A* is in (in fact, it may not even be possible, in certain systems, for *B* to determine that *A* is missing). Thus, an attempt by *B* to communicate with *A* will either never get any reply, or may get a *Process-Missing* reply from *A*'s home site (the *Process-Missing* reply carries no other information, such as whether *A* committed or aborted before vanishing).

∗ The actions that a process takes when in a particular state are highly variable and dependent on the particular transaction execution and commit algorithm. What I have tried to capture in Figure 2 are those actions that are universal or almost so.

∗ The coordinator passes through state *ALL-DEP* ("all participants *DEPENDENT*") before entering state *C-COM*, in which the transaction is committed. The *ALL-DEP* state has been included to allow for the possibility of the coordinator requesting a final go-ahead from the user (which the user might refuse) when all of the participants are *DEPENDENT* and ready to commit. This option is not provided in many protocols, and in these the state *ALL-DEP* does not appear, the coordinator making an immediate transition into state *C-COM* (thus committing the transaction), on receiving *DEPENDENT* messages from all participants.

∗ Figure 2 shows the allowable *transitions* between the possible states of a process.

What is not shown is when these transitions actually occur; this again is highly variable and will be discussed in this paper. There are two important points about these state transitions that should be apparent in Figure 2: (i) a process never goes from its *COMMITTED* state to its *ABORTED* state or vice versa; and (ii) the only possible transitions out of *COMMITTED* and *ABORTED* are into *MISSING-OK* and *MISSING-X*, respectively.

3.2. Basic Two-Phase Commit

The basic state model of the previous section can be used to describe a "two-phase" commit protocol. This appears to be the simplest possible type of distributed transaction commit protocol under our assumptions (about centralized commit decision, etc.), since it incorporates the bare necessities of the "two-phase" requirement presented in Section 2.2. In this section, I shall assume that we are dealing with distributed transactions that have *fixed structure*, i.e., the coordinator can, at the start of the transaction, completely determine the participant structure and actions required to implement the user's request, and this structure cannot be modified as part of the same transaction (i.e., if some participant cannot carry out its actions, the coordinator must abort the transaction). I consider transactions with unpredictable or variable structure in Section 4.

The two-phase commit protocol for fixed-structure transactions is shown in Figure 3. In accordance with to the two-phase requirement of Section 2.2, the processing of a participant in a distributed transaction is performed in two phases:

- *"Phase 1"*, in which the participant is not dependent on the coordinator (e.g., the participant can choose to abort its local actions).

- *"Phase 2"*, in which the participant cannot commit or abort except at the command of the coordinator.

These phases have been indicated in Figure 3. For a transaction with fixed structure, a participant receives a single request (which may be *composite*, i.e., may request a sequence of actions such as data access, update or transfer) from the coordinator; the participant enters "Phase 1" and starts to process the requested actions. When the participant has finished acting on the request, it writes the "intentions" for its actions (ensuring that it can commit the actions if the coordinator so desires), and then enters the *DEPENDENT* (*P-DEP*) state (i.e., "Phase 2"). Having done so, the participant sends the coordinator a *DEPENDENT* message and awaits a *COMMIT* (or *ABORT*) message.

Some points about Figure 3 should be noted for understanding:

- The figure only shows a single participant, but there will in general be several participants each executing an identical protocol. Also, the coordinator executes its protocol against each participant; at the point where it awaits a given message from each participant, it only makes the indicated state transition when it receives the desired message from *all* of the participants.

- The figure only models the *normal* flow of execution, i.e., when all of the transaction's actions can complete and the transaction eventually commits. Issues of when a transaction aborts are treated in Section 3.3.

- The figure does not include "nonessential" messages that are not a critical part of the model (an example of such a message would be a "hurry up and get done" from the coordinator to a participant; such messages are mainly for speedup, and they appear in some algorithms but not others). Also, messages relating to transaction aborts have not been shown in Figure 3, e.g., messages from a participant to the coordinator stating that the former aborted or was otherwise unable to complete its requested local actions.

- In Figure 3, I have assumed a "broadcast" protocol for the manner in which the coordinator is informed that all of the participants are *DEPENDENT* and that the transaction can therefore commit. There are variations possible in this first phase, such as:
  * A *chained* protocol, in which a single *DEPENDENT* message is passed from one participant to the next; when the message reaches the coordinator, it can be certain that all participants are in the *DEPENDENT* state. (Gray refers to this protocol as *nested* two-phase commit in [2].)
  * A *hierarchical* protocol is sometimes used when a participant in a transaction can spawn its own subordinate participants, in a manner transparent to the coordinator (for example, Reed [9] allows "modular construction" of transactions from other transactions whose implementation may not be known). Then, each process is responsible for establishing when all of its subordinates have reached the *DEPENDENT* state (it may use a broadcast or chained protocol, for example); only then can the process itself enter the *DEPENDENT* state and inform its superior. (For transactions whose process structure is not completely known to the coordinator, certain optimizations on the above are possible, e.g., Montgomery's distribution of "completion weights" [8] among the processes allows each process to signal *DEPENDENT* directly to the coordinator, and the coordinator is still able to determine when all participants have completed even though it is not

certain who the participants are.)
For simplicity, I shall ignore variations such as the above; our discussion should not be affected much when applied to commit protocols that use methods other than the broadcast method for determining when a transaction is complete and ready to commit.

3.3. Issues for Discussion and Comparison

In this section, I point out some properties of the basic two-phase commit protocol just described: the degree of *resiliency* of the protocol, the mechanisms by which a transaction can be *aborted*, and the means through which the coordinator and participants can *forget* the outcome of a transaction once it has committed or aborted. These properties (and some more, to be introduced in Section 4) will be referred to in the discussion and comparison of specific protocols, in Section 5.

Resiliency. A protocol that obeys the two-phase requirement, is certainly "resilient" in the sense that a transaction will eventually commit or abort even in the presence of failures. However, in certain phases of the protocol a process *A*, say, is *dependent* on another process *B*, say, in the sense that *A* cannot take unilateral action (e.g., commit or abort) without *B*'s consent. In such a situation, a failure of *B* forces *A* to wait indefinitely for *B*'s recovery. I shall use the term "resiliency" in a very narrow sense, to describe the extent to which a protocol is prone to such waits due to process dependencies.

I examine this issue of process waits in the situation where the outcome of the transaction has not been resolved (to the knowledge of process *A* in the above); I shall consider the situation after the resolution of the transaction's outcome as a separate issue presently. For the basic two-phase commit protocol, we observe the following regarding the dependence of the coordinator and participants on each other:

I. *Participant waits.* As already stated in the state model, if P is in its *DEPENDENT* state it must await the coordinator's instructions regarding the outcome of the transaction. This means that the participant cannot unilaterally abort its local actions, which it might desire to do if, for example, some other transaction makes a conflicting request. Such conflicting transactions must instead be held up (unless some special mechanism, e.g., [10, 8], allows such transactions to proceed). This is an undesirable effect, but it is one that seems to be intrinsic to the problem of committing distributed transactions. One of the strategies commonly used to reduce the time "window" during which the participant is dependent on the coordinator is to use a "three-phase" commit, which I describe in the next section. There are alternative mechanisms which try to *decentralize* the decision (either completely decentralizing it, or doing so only during a critical time window) as to whether the transaction should commit or abort, thus making the participants not dependent on any single site; some of these are:

1. If a participant detects a failure of the coordinator, it *inquires* of the other participants as to whether or not they committed their actions. This scheme was described in an older version of the SDD-1 reliability mechanisms [3].

2. A chain of *backup coordinators* is used; if the coordinator currently in control of the transaction fails, the next one in the chain takes over. This scheme is used in SDD-1 [4].

3. *Multiple coordinators* are used, with a *voting* scheme to decide the outcome of the transaction. This scheme is described in Reed [9].

These mechanisms seem to be fairly general in the sense that they could be applied to commit protocols in environments other than the ones for which they were originally designed. The mechanisms can naturally be expected increase resiliency as well as overhead; however, I am not attempting to model these mechanisms in this paper.

II. *Coordinator waits.* Before it can decide to commit the transaction, the coordinator *C* is "dependent" on each participant *P* in the sense that *C* cannot commit the transaction until it receives a *DEPENDENT* message from *P* stating that *P* will be able to commit its local actions if *C* desires. (Of course, the coordinator can abort the transaction at any time if it so chooses; this it might do if a participant crashes. The coordinator might also be able to *restructure* the transaction, but I do not consider this possibility until Section 4.) Thus, while awaiting *DEPENDENT* messages from the participants, the coordinator has no choice but to continue waiting or to time out and abort. This dependence on the coordinator on the participants is an intrinsic property of almost all the distributed commit procedures I am examining, with the important exception of SDD-1 [4]. In SDD-1, data management sites cannot refuse instructions to update their data; this allows a coordinator to "spool" update messages for data managers that are down, and to then go ahead and commit the transaction. (I shall discuss this further when I examine the SDD-1 commit protocol in Section 5.)

Transaction Aborts. There are several reasons why a distributed transaction may abort: concurrency control and potential deadlock, site crashes, timeouts triggered by excessive delays, autonomous decisions by a participant or by the coordinator or user. I shall not examine these reasons in any detail, but rather concentrate on the issue of how a decision of one process to abort causes the transaction as a whole to be aborted. There are essentially two issues to be considered here: (i) communicating a participant's abort decision to the coordinator; and (ii) communicating the coordinator's abort decision to all of the participants. These are considered in turn below.

I. *Local participant aborts.* As noted earlier, the participant can only abort its local actions before it enters the *DEPENDENT* state. However, before entering the *DEPENDENT* state, a participant's local actions may be aborted for any of several reasons: a crash of the participant; synchronization reasons, to maintain transaction atomicity or to avoid a real or potential deadlock; or a timeout on an expected message from the coordinator. If we continue to assume a fixed transaction structure, then a local abort at any one participant should cause the transaction as a whole to abort, i.e., the coordinator must enter the *C-ABORT* state at some point. I examine here the issue of how the coordinator is informed of a participant's aborting. In general, this is done *implicitly*, by the coordinator setting a timeout and aborting the transaction if not all participants respond (see below). However, this implicit mechanism is often speeded up by *explicit* messages from the participant to the coordinator indicating the former's inability to satisfy the latter's requests. Of course, these speedups are useless when the participant crashes (since the coordinator may wait indefinitely for the participant to recover and send a message), so some implicit mechanism at the coordinator is always required. (For reasons associated with "forgetting" transactions, which I examine presently, it may be desirable for a participant to remain in a recoverable *P-ABORT* state until it has sent an *ABORTED* message to the coordinator.)

II. *Aborts initiated by the coordinator.* Before it enters the *COMMITTED* (*C-COM*) state, the coordinator may decide to abort the transaction for any of several reasons: a participant signals its inability to complete a request; a timeout, if the *DEPENDENT* messages from the participants don't come in for too long a time; the user autonomously decides to abort and sends an *abort-transaction* request; or the coordinator crashes. Once the coordinator decides to abort the transaction (or has to abort after a crash, because it is in state *C-ABORT* on recovery), its decision must somehow be propagated to all of the participants. In general, this propagation must be *explicit*, i.e., *ABORT* messages must be sent to the participants, because some participant(s) may already be in the *DEPENDENT* state and will be awaiting the coordinator's instructions. Only in certain special situations can this be avoided. For example, in the old version of Lampson and Sturgis' algorithm [5], the coordinator can simply disappear if it is aborting; a participant in its *DEPENDENT* state that does not get a *COMMIT* message and finds the coordinator "missing" will infer that the transaction must have aborted, and will also abort. Also, in "three-phase" commit, which I describe in Section 4, the coordinator can vanish if it aborts before sending any *GET-DEPENDENT* messages to the participants, because it can be certain at that point that no participant is as yet *DEPENDENT*.

"Forgetting" a Transaction. Here I consider the situation after a transaction has been committed or aborted, i.e., after the coordinator has entered the *C-COM* or *C-ABORT* state, repsectively. The participants must be instructed to commit or abort their local actions, and this can be done in two ways:

1. *Active*: the coordinator sends a *COMMIT* or *ABORT* message to each participant. (This is illustrated in Figure 3.)

2. *Passive*: each participant inquires of the coordinator whether the transaction was committed or aborted. (This inquiry, which is not modelled in Figure 3, is usually triggered by a conflicting request from some other transaction, or may happen when the participant recovers from a crash.)

Either method can be used by itself (and the "active" method is frequently used as such), or both can be used in some combination (e.g., in [9]). The issue that I look at here is when a process (coordinator or participant), after determining the outcome of a transaction and performing the associated processing, can *forget* the transaction, i.e., can vanish and erase all memory of the outcome of the transaction. For example, if only the "active" method above is used, then the coordinator must persistenly send out a *COMMIT* (or *ABORT*) message to each participant until the participant gives a positive acknowledgement that is has received and acted on the message; if not, it may happen that some participant never receives the message, e.g., if it is down at the time the coordinator sends out the messages. This fact is illustrated in Figure 3, where the coordinator waits for *COMMIT-ACKs* to arrive from all of the participants.

If we view the situation from the point of view of a participant, however, we see that Figure 3 is not quite correct. In particular, it is not really correct for a participant to vanish after committing its local actions and sending a *COMMIT-ACK* to the coordinator (these can be done in any order or in parallel without having any effect on the argument), because it will not be certain that the coordinator received the *COMMIT-ACK*. For example, the coordinator may have crashed after sending out the *COMMIT* but before receiving the *COMMIT-ACK*. On recovery, the coordinator will send another *COMMIT* but the participant may have vanished in the meantime and will not reply; the coordinator will then be in the unhappy situation of forever sending messages with no hope of ever getting the desired reply. (Even if a "passive" strategy is used, and the coordinator doesn't persistenly send *COMMIT* messages, it will in any case have to stay around forever and be prepared to respond to an inquiry that will never come.)

To solve the above problem, one might require that the participant not vanish until it is certain that the coordinator has received the *COMMIT-ACK*. This idea, however, merely passes the buck to the coordinator, who must now acknowledge the *COMMIT-ACK* and cannot vanish until it is certain that the participant has received this acknowledgement! This argument can be extended ad infinutum, and we can conclude that there is *no fixed-length protocol* of message exchanges (acknowledgements and acknowledgements of acknowledgements) that will allow both the coordinator and the participant to correctly vanish. That this is in fact true can be proved very easily: Suppose there is such a protocol M_1, M_2, ..., M_n, say, of length n. Let A be the process that sends the next-to-last message M_{n-1}; it must receive the last message M_n from the other process B, say, before it can vanish. Now suppose A sends out M_{n-1} and then crashes; B, meanwhile, receives M_{n-1}, sends back M_n and then vanishes without waiting for an acknowledgement (since M_n is the last message in the protocol). Now A is currently down, and therefore will not receive message M_n; on recovery, therefore, A will wait forever for M_n to arrive, and will never be able to vanish. The protocol therefore cannot be correct.

From the above, we can expect that neither process can ever "forget" a transaction (the process itself may be allowed to disappear, but is home site must forever maintain a record of the outcome of the transaction). This seems to be true in general (see [6]). Most distributed commit algorithms have a "post-commit" protocol that involves just two messages: *COMMIT* and *COMMIT-ACK*. With a scheme like this, the coordinator can forget the transaction once it has received *COMMIT-ACKs* from all of the participants, but a participant can never forget the transaction. This is so because the coordinator may crash after sending a *COMMIT* and would therefore not receive the participant's *COMMIT-ACK* in response; on recovery of the coordinator, then, the participant will receive a duplicate *COMMIT* message. Thus, the participant must always be prepared to respond to a duplicate *COMMIT* from the coordinator. The methods that are used to ensure this are the following:

1. The participant either remembers the transaction forever, or maintains sufficient information to be able to respond to a duplicate *COMMIT* or *ABORT* message. The latter approach is the one frequently taken, using monotonically increasing transaction sequence numbers or "timestamps". The schemes of SDD-1 [4] and Reed [9] are examples of this approach.

2. The *implicit* approach. Here the participant is allowed to vanish immediately once it has sent the *COMMIT-ACK* to the coordinator and committed its local actions. The coordinator in its turn persistently sends *COMMITs* to the participant until either it receives a *COMMIT-ACK* in reply or it learns that the participant has vanished. I call this method "implicit" because the coordinator must infer that the participant correctly completed its actions and vanished. This method was used, for example, in Lampson and Sturgis' old algorithm [5], but does not seem to be much in favor currently. This is probably because the implicit approach has the following problems: (i) the use of the implicit approach requires that the home site of a vanished process be capable of generating a *Process-Missing* reply in response to a message sent to that process, and this capability is not always provided; (ii) in order to properly generate a *Process-Missing* reply, a site must maintain some memory of processes, in which case the approach becomes little different from approach (1) above - the participant's home site may as well be prepared to send the required *COMMIT-ACK* when it receives a duplicate *COMMIT*.

3. *Guaranteed delivery and duplicate suppression*, at the network level. Here, the underlying network ensures that message delivery is robust and duplicate-free, and the participant process never has to worry about receiving a duplicate *COMMIT* message from the coordinator. This is the

approach Montgomery takes with his "robust sequenced processes" [8]. (We note that the same problem, of having to "remember" all messages, or at least the last "sequence number", has to be solved at the network level if that is the level at which duplicate messages are eliminated.)

4. GENERALIZATIONS OF THE MODEL

The state model of Section 3 covers only the basic requirements of consistency and correctness. However, many distributed transaction commit protocols incorporate more detail than the "basic" two-phase commit. In this section, I present an "extended" state model which attempts to capture such detail. This extended model will be used to identify a broader range of commit protocols, and also to bring up some additional issues that arise, in particular unpredictable and variable transaction structures and early release of uncommitted transaction results.

4.1. Extended State Model

Here, the "basic" state model of Section 2 is extended slightly (by refining the states *START* and *DEPENDENT* of the processes before transaction commit/abort) to carry more information about the progress of the transaction. In particular, I introduce a *DONE* state for participant processes. A participant will be in its *DONE* state when it is certain that there will be no further local actions (other than committing or aborting actions already "performed", on a tentative basis) that it must perform on behalf of the transaction. The purpose of introducing this concept is to model the following:

1. Delaying the dependence of the participants on the coordinator. In its *DEPENDENT* state, a participant must await instructions (to commit or abort) from the coordinator, which means that the participant may have to wait an indefinite amount of time, e.g., if the coordinator crashes. While this effect cannot be avoided altogether, its effects can be reduced if a participant does not enter the *DEPENDENT* state immediately when its local actions are complete. This procedure is followed in "three-phase" commit protocols (of which Lampson and Sturgis' old algorithm [5] is an example), which we examine presently.

2. Early release of transaction results. Certain schemes (such as Takagi's [10] and Montgomery's [8]) allow the outputs of not yet committed transactions to be supplied to other transactions, usually to allow increased concurrency. For such schemes, the *DONE* state can be used to indicate the earliest point in the transaction's local progress at which a participant can release the

transaction's results. (One could conceive of a participant releasing a transaction's results even before learning that its local actions are *DONE*, but this is unlikely to be often fruitful since the results will be "dirty" and subject to further modification by the transaction at any time.)

The extended state model is presented in Figure 4. The state of a participant process before it learns that the transaction is committed or aborted can be described in terms of two dimensions:

- Whether or not the participant's local actions on behalf of the transaction are *DONE*.
- Whether or not the participant is *DEPENDENT* on the coordinator, in the sense that it must await the coordinator's instructions on whether to commit or abort the transaction's local actions.

The state of a participant can thus be described as an "ordered pair" of values along these two dimensions. In Figure 4, I have used the value "-", in the first and second elements of the pair, to indicate that the participant is not *DONE* or not *DEPENDENT*, respectively. The state of the coordinator can similarly be described along two dimensions: whether or not it knows that all participants are *DONE*, and whether or not it knows that all participants are *DEPENDENT*.

We note that the extended state model of Figure 4 is intended to be very general; not all of the states that appear in the model actually manifest themselves in all of the commit protocols that I am examining. We shall see, for example, that in "three-phase" commit, every participant's *DONE* state precedes its *DEPENDENT* state; the state *<-,P-DEP>* (*DEPENDENT* but not yet *DONE*) does not appear in this kind of protocol. However, there are other possible protocols, involving "conversational" transactions (Section 4.3), in which a participant can be *DEPENDENT* but not yet *DONE*.

4.2. Three-Phase Commit Protocol

In *three-phase* commit (Figure 5), the coordinator and participants go through an extra round of message exchanges before the participants enter the *DEPENDENT* state and await the coordinator's instructions. The request for processing that the coordinator sends to a participant (I am still assuming a "fixed" transaction structure here) asks for a *DONE* message to be sent in reply on completion of the participant's local processing.

Only when it is certain that all participants are *DONE* does the coordinator request them to enter the *DEPENDENT* state. The three "phases" that a participant goes through, prior to committing or aborting the transaction, are the following:

- *"Phase 0"*, in state *P-START* (which is the same state as <-,-> in Figure 4). This phase ends when the participant completes the requested local processing, on a "volatile" basis (i.e., such that the participant can abort them at any time), and enters "Phase 1".

- *"Phase 1"*, in state *<P-DONE,->*. The participant has completed its local processing but is not yet dependent on the coordinator. (This means, for example, that the participant can choose to abort its local actions if it so desires.) In this phase, the participant sends the coordinator a *DONE* message and waits for a *GET-DEPENDENT* message (which the participant can refuse if it has unilaterally aborted); on receipt of this message, the participant records its "intentions" (to ensure that it can commit if the coordinator says so) and enters "Phase 2". (Note that the participant might choose to start recording its intentions while awaiting the *GET-DEPENDENT*, for faster response; on receiving the *GET-DEPENDENT*, the participant only has to save its "state variable" on stable storage.)

- *"Phase 2"*, in state *<P-DONE,P-DEP>*. This is the same as Phase 2 of "two-phase" commit: the participant sends a *DEPENDENT* message and must await the coordinator's instructions to commit or abort.

The progression of the coordinator through its states is similar. Note that in Figure 5, I have made room for a "final user *OK*" when all participants are *DONE*; if the user is to be given a final chance to retract his transaction, this seems to be the most appropriate point, before any participant is *DEPENDENT* on the coordinator. Then, the coordinator's *ALL-DEP* state (*<ALL-DONE,ALL-DEP>* actually) coincides with state *C-COM*, since it is reasonable for the coordinator to go ahead and commit the transaction once it learns that all of the participants are *DEPENDENT*. (Allowing the user a final chance to abort at this point, after all participants are *DEPENDENT*, is conceivable but seems pointless.)

The canonical example of a three-phase commit protocol is that of Lampson and Sturgis' old algorithm [5]. (The transaction commit procedure in Takagi [10] is also three-phase, as I shall show in Section 5.) The effects of introducing the extra phase in the protocol are the following:

1. Increased Resiliency. By delaying its entry into the DEPENDENT state, a participant reduces the time window during which it must await the

coordinator's instructions. This effect is most prominent when one participant P_i, say, completes it local actions much earlier than another P_j, say (which often occurs in practise, e.g., if P_j's processing depends on the completion of P_i's). Then P_i will enter the *DONE* state but will not be *DEPENDENT* until the coordinator is certain that P_j is also *DONE*; if the coordinator crashes in the meantime (or if there is a conflicting request at P_i and no *GET-DEPENDENT* message is received in a very long time), P_i will be able to unilaterally abort its local actions. The participant P_i would not have this option in two-phase commit, since it would immediately enter the *DEPENDENT* state on completion of its local actions.

2. Additional delays, and the cost of sending extra messages, are incurred; this illustrates the tradeoff that one commonly observes, between resiliency and overhead.

3. A less important effect than the above is the following: Before entering the *ALL-DONE* state, the coordinator can be certain that no participant is yet in the *DEPENDENT* state (since no participant becomes *DEPENDENT* without the explicit instruction of the coordinator). Then, if the coordinator wishes to abort the transaction before it has sent any *GET-DEPENDENT* messages, it can simply vanish without having to send explicit *ABORT* messages to the participants (who will eventually abort their local actions of their own accord). (This reduced dependence of the coordinator on the participants, before the *ALL-DONE* state, might also allow the coordinator more freedom to "restructure" the transaction, which I discuss presently.)

4.3. Conversational Transactions

Here I consider a more general kind of transaction structure, in which the transaction is described not by a single composite request, entered by the user all at once, but rather by an arbitrary sequence of actions requested interactively by the user. At some point during the interaction, the user will request that his actions be committed (or aborted) as an atomic transaction. The participants in such a transaction receive requests for actions one at a time, and at any given point may not know what further actions will be requested under the same transaction.

I shall assume that the user in such a conversational transaction is ***well-behaved***; that is, the user does not request a transaction commit until he has received acknowledgements that all of his requests have in fact been acted on. (If the user is not well-behaved, then the outstanding unacknowledged requests may or may not get performed depending on the timing of specific messages; I shall not consider this possibility.)

The range of possible commit protocols for conversational transactions is somewhat wider than for transactions with fixed structure. I have attempted to give a broad classification of these protocols (with my own "names" for the protocols) below. (The state transitions for these classes of protocols are shown in Figure 6.)

1. *Strongly Dependent.* Here a participant immediately performs each requested action in *DEPENDENT* mode. Thus, the participant does not know at any given time whether or not it is *DONE*, but must at any time be prepared to commit (or abort) the actions that it has performed so far. (Note that the *P-DONE* state is absent in such a protocol. In fact, the participant only knows that it is *DONE* when it receives instructions to *COMMIT*; thus, state *P-COM* implies *P-DONE*.) This is the simplest possible protocol for committing conversational transactions (it has but two "phases"), but it is not extremely resilient since a participant is immediately *DEPENDENT* on the coordinator once it performs a requested action. (The basic scheme of Reed [9] and Lampson and Sturgis' new algorithm [6] are of this form; these authors, however, do allow for more resilient protocols to be implemented by the user or application).

2. *Standard.* In this protocol, a participant performs all requested actions on a volatile basis at first, and only enters the *DEPENDENT* state when the coordinator sends it an explicit *GET-DEPENDENT* message (at which point the participant also knows that it is *DONE*). This protocol does provide increased resiliency, since a participant only enters the *DEPENDENT* state when the coordinator is certain that all actions to be performed on behalf of the transaction are *DONE*; until then, the participant has the freedom to unilaterally abort. This protocol can be called "two-and-a-half" phase, since the coordinator goes through three phases (initial, *ALL-DONE*, and finally *ALL-DEP/COMMIT*) but a participant goes through only two (initial, and *<P-DONE,P-DEP>*).

3. *Extended.* This is almost the same as "standard", except that a participant's *DONE* and *DEPENDENT* states are separated. This protocol is truly three-phase, since both coordinator and participant go through three phases. In terms of resiliency, the extended protocol has very little more to offer than the standard protocol above. The extended protocol is useful, however, in situations where a transaction's results can be released before they are committed. The scheme of Takagi [10] is an example of such an extended protocol.

(In all of the above, the coordinator enters state *ALL-DONE* when the user declares that there are no further actions to be performed under the transaction; a "well-behaved" coordinator will also not enter the *ALL-DONE* state until it has received acknowledgements of all the requests made under the transaction. Note that when the coordinator is *ALL-DONE*, the participants themselves may not be aware that they are *DONE*, e.g., in the "strongly dependent" protocol.) In Section 5, I shall look at some example distributed commit protocols that fit the categories described above.

4.4. Dynamic Transaction Restructuring

Here I consider the possibility of the coordinator (or the user) *restructuring* a transaction, i.e., retracting some actions and starting some other ones (such as performing the desired actions on an alternate copy of the data in question). There are various reasons why this might be done: some participant signals that it is unable to perform the actions requested of it, or that it has aborted its local actions; some participant is down or is very slow in responding to the requests sent to it; the user may change his mind about an action that he requested. When the coordinator wishes to restructure a transaction, it must ensure that the undesired participant actions get properly aborted.

The transaction restructuring problem can therefore be reduced to that of the coordinator *selectively aborting* a given participant *P*, say, while trying to commit a restructured transaction consisting of the actions of the other participants; it is this simplified problem that I examine here. The safest procedure for the coordinator to follow would be to persistently send an *ABORT* message to participant *P* until an acknowledgement is received, and only then to commit the transaction (by entering the *C-COM* state and sending *COMMIT* messages to the other participants). This method has a major drawback, namely that the transaction must wait an indefinite amount of time if the undesired participant *P* is down. The approaches that can be taken to remedy this are the following:

1. *Disallow* any such restructuring. This, essentially, is the approach taken in [9, 8, 6]. In Reed's scheme [9], for example, once a "token" has been created under a given transaction (a "possibility" represented by a "commit record"), it is not possible for that individual token alone to be aborted; it is only possible for the transaction as a whole, with all of its tokens, to be aborted (or committed). (Note that certain kinds of restructuring are indeed possible under Reed's scheme, if tokens are first created under "dependent

possibilities"; I shall examine such strategies in Section 5.)

2. Allow *limited* restructuring, only during the time interval that the coordinator is certain that no participant has performed its actions in *DEPENDENT* mode. This is only possible under "three-phase" commit protocols (in two-phase commit, the coordinator has no control over when a participant becomes *DEPENDENT*). Thus, in three-phase commit (Figure 5), the coordinator and/or the user can choose to restructure the transaction any time before the *GET-DEPENDENT* messages are sent out to the participants, i.e, before state *ALL-DONE*. (Using this strategy, the coordinator need not even send an *ABORT* message to a participant that it wishes to abort; since the participant cannot be in the *DEPENDENT* state, its local actions must be volatile and will eventually be aborted anyway.)

3. Allow *arbitrary* restructuring at any time, so long as the coordinator does not send both a *COMMIT* and an *ABORT* to the same participant. This strategy is not followed in any of the schemes I am examining, but it is conceivable. For example, in Takagi's scheme [10] one could conceive of the coordinator having a "*commit* P_i" and an "*abort* P_j" in the transaction "commit cache". Such a scheme is dangerous, however. It would not work, for example, if the participants could query a "commit record" that contained no information other than whether the transaction as a whole committed or aborted; thus if participant P_j (who should abort) does not receive the *ABORT* message above, it may later get a "commit" response to its inquiry. (Similar problems arise if participants can inquire of each other as to the outcome of the transaction.) In order for the strategy to work correctly, the coordinator would have to retain enough information about the transaction to be able to give different responses to inquiries from different participants. This hardly seems worth the trouble, since the benefits that might derive from allowing arbitrary transaction restructuring are questionable.

We note that the subject of discussion here is one that most sources in the literature do not mention explicitly. However, it appears that some form of transaction restructuring is often useful, and what I have tried to do here is indicate the available options and the problems associated with each. It appears that "arbitrary" restructuring of the kind desribed above is hardly worth the trouble, but that "limited" restructuring seems reasonable and not dangerous. Limited restructuring can be implemented, for example, using Reed's "dependent possibilities" [9].

4.5. Early Data Release

Here I consider the issue of when a participant can release the results of a transaction to other transactions. The discussion will be couched in terms of data objects that the transaction updates and that other transactions may want to read (or further update). The possible approaches that can be taken are the following:

1. Don't release the results until certain that the transaction has committed (or aborted).

2. Allow "early" release of a transaction's results in a controlled fashion, only when the participant is certain that the transaction will perform no further actions on the data object in question.

3. Allow "early" release of a transaction's results at any time.

Strategy (1) has been followed in almost all schemes until recently [2, 9, 6, 4]. Under this strategy, uncommitted transaction results are never released, so that problems of *cascaded backout* are avoided; the strategy is simple, and warrants no further discussion. Early data release, however, can provide increased concurrency, as pointed out by Takagi [10] and by Davies [1]. In order to maintain consistency (transaction atomicity) in the face of early data release, additional system overhead has to be incurred, e.g., to keep track of dependencies among uncommitted transactions. In this connection, strategy (3) is unlikely to be productive, since if a transaction's results are released before it is established that they are "done", then the results will be "dirty" and subject to modification at any time; transactions that read such dirty data will most likely have to abort. (At best, the other transactions may be able to "recompute" if there is some way that they could be "informed" of changes to their inputs. This possibility is briefly alluded to by Davies, but no specifics are given; clearly, the system overhead required is even greater than with strategy (2).)

I shall therefore exclude strategy (3) from our consideration, and examine early release of transaction results under strategy (2), i.e., when the participant process is certain that the transaction's actions on the data object are complete. The concept of early data release can be captured in the model, using the *DONE* participant state. (In actual fact, it would be more correct to associate a *DONE* state with each object that the transaction acts on; this is what Takagi's scheme does in effect. However, the idea is the same, whether we consider the participant's composite actions on behalf of the transaction to be *DONE* or whether we consider the transaction's actions on individual objects to be *DONE*.) The schemes of Takagi [10] and Montgomery [8] are the only ones I am aware of that provide early data release of the kind being considered here, and they represent two different approaches to the problem:

1. *Conditional release* of uncommitted results (Takagi). Here, a transaction T_2 is allowed to read the uncommitted outputs of transaction T_1 under the condition that if T_1 eventually aborts, T_2 will also abort.

2. *Polyvalues* (Montgomery). Here, if transaction T_1 is *DONE* (with respect to a given data object or participant process), transaction T_2 will be given a *pair* of values, representing the inputs that it would get if T_1 committed and T_1 aborted, respectively. T_2 can then compute the results that it would output under all possible situations (it may itself install polyvalues into the database, but in general it would not complete unless its outputs to the real world turned out to be the same under all possible situations). This scheme allows T_2 to proceed without fear of "cascaded backout", because its own completion is not dependent on T_1 eventually committing.

Of themselves, the above mechanisms are not of interest for this paper. However, they do have an impact on the transaction commit protocol, and I shall look at this issue when I examine Takagi's and Montgomery's schemes in Section 5.

5. EXAMPLE DISTRIBUTED COMMIT PROCEDURES

In this section, I consider several distributed commit protocols described in the literature, namely those of [2, 9, 6, 10, 4, 8]. I shall try to see how these schemes fit into my general model for such protocols, and examine some of the issues raised earlier, such as "forgetting" transactions, resiliency, and, where applicable, "early data release".

5.1. Gray

Gray's description of "two-phase commit" [2] is sketchy at best. For example, he does not address the issue of how the coordinator determines that it is satisfied with the actions of the participants (i.e., that the participants are *DONE*), and when a coordinator starts requesting "votes" from the participants. Also, Gray does not completely address the issue of when a transaction can be "forgotten". He does say that the coordinator persistenly sends *COMMITs* (or *ABORTs*) until it has received acknowledgements from all of the participants, at which point it can vanish. However, Gray does not look at the problem from the participant's viewpoint, i.e., the fact that a participant cannot vanish after committing and sending an acknowledgement to the coordinator, since the coordinator may be down and may later send a duplicate *COMMIT* (or *ABORT*). As was pointed out in Section 3.3, the participant must be able to respond to such a duplicate *COMMIT*. (The participant's home site might be able to do so by examining its "log", and this would be all right even though it is slow; Gray does not discuss this possibility at all.)

If we ignore the above problems, we see that the basic features of Gray's scheme do fit easily into the model:

- When the participant "forces" (i.e., writes on stable storage) its *"AGREE"* record, it enters what I have called the *DEPENDENT* state, since it must now obey the coordinator's instructions to commit or abort.

- The coordinator commits the transaction when it forces its *"PHASE12-COMMIT"* record; it enters the *C-COM* state at this point. As noted in the "two-phase" requirement (Section 2.2), the coordinator can only do this when it is certain that all participants have forced their *"AGREE"* records, i.e., are in the *DEPENDENT* state.

As noted above, the description of Gray's scheme is lacking in detail, so we shall not discuss it any further.

5.2. Reed

The most striking feature about Reed's scheme [9] is its flexibility: The user or application can choose whatever kind of protocol it wishes to follow for deciding when to commit or abort a transaction, and the system ensures that undesirable effects don't happen no matter what the user does. Thus, for example, the system sets a timeout on every transaction ("possibility", actually), which protects against undesirable user process behavior such as looping forever, disappearing without issuing a commit or abort instruction, etc. In Reed's scheme, the setting up of coordinator and participant processes, and the means by which they communicate and cooperate, is completely invisible to the system. However, the system takes the responsibility for ensuring that the requests of the user processes are correctly and consistently processed; thus, the system allocates a "commit record" for a transaction, and is responsible for propagating the outcome of the transaction to all of its actions and for determining when the transaction can be forgotten (i.e., when the commit record can be reclaimed).

In examining Reed's scheme, I shall first see how the "kernel" of the scheme fits the model, and then examine how some of the features (mainly "dependent possibilities") can be used to implement the classes of protocols that I have described. (I shall for the most part ignore the "pseudo-temporal environment" mechanism, since its purpose is for transaction synchronization and it has little to do with recoverability.) Using just the basic features of Reed's scheme, a user wishing to execute a transaction would run under a "possibility" that has a "commit record", and would create just a single process, the coordinator, to execute the transaction. (I assume that the coordinator and the commit record are located at the same site, for simplicity.) The coordinator would send requests for actions on objects to the "object managers" of the objects in question. (These object managers, who may be resident on various sites, are effectively the "participants" in the transaction.) If a particular request involved updating an object, a "token" would be created that would refer to the commit record associated with the transaction (or possibility). At some point, the coordinator would execute a complete or abort request, which would be reflected in the state of the commit record; the tokens created by the transaction would then become valid "versions" (or aborted versions). This naive use of Reed's mechanism corresponds to the *"strongly dependent"* commit protocol (Figure 6):

- Once a token is created under a given transaction, it is effectively in the *DEPENDENT* state, since the token can only be committed or aborted on instruction of the transaction's commit record. The token cannot be selectively aborted by a user process (or for concurrency control reasons, or as a result of failures other than a timeout on the commit record); it only gets aborted if the transaction as a whole aborts.

- The concept of a *DONE* state (for an object that the transaction acts on) does not appear; a participant (object manager) only knows there will be no further actions on the object when it learns that the transaction has committed (or aborted).

- If the coordinator is not "well-behaved" (i.e., executes a complete request before gettting acknowledgements of all token requests), then the unacknowledged requests may or may not get performed.

Reed's mechanisms can be used in a much more disciplined way than the above, through the use of *dependent possibilities.* In particular, I propose the following setup for a distributed transaction: The coordinator sets up a participant process at each site that will be involved in the transaction, and each such participant executes under a separate dependent possibility (i.e., a possibility that is dependent on the possibility under which the transaction as a whole is running). The coordinator and the participants follow some kind of protocol to decide when: (i) a given participant should commit its dependent possibility, thus making its actions (tokens created, etc.) directly dependent on the parent possibility; (ii) the coordinator should commit the transaction, by committing the parent possibility. In Reed's scheme the user is free to use any kind of protocol, meaningful or not, for making the above decisions; I show below how this structure can be used to implement the classes of protocols described in Sections 3 and 4. If we assume a fixed transaction structure (in which the coordinator sends each participant a single, perhaps composite, request, and waits for a notice of completion), the two- and three-phase commit protocols can be implemented as follows:

1. *Two-Phase Commit.* Here, each participant executes a procedure of the following form:

```
possi := possibility$dependent()
            were possi do   ; under a dependent possibility
                  ; perform actions requested
                  ; (create-token, etc.)
                  end
            possibility$complete(possi) ; enter DEPENDENT state
            send(coordinator,"DEPENDENT")      ; inform coordinator
```

2. *Three-Phase Commit.* The participant procedure is extended somewhat:

```
possi := possibility$dependent()
            were possi do   ; under a dependent possibility
                  ; perform actions requested
                  ; (create-token, etc.)
                  end             ; enter DONE state
            send(coordinator,"DONE")           ; inform coordinator
                  ; await reply GET-DEPENDENT, on receiving:
            possibility$complete(possi) ; enter DEPENDENT state
            send(coordinator,"DEPENDENT")      ; inform coordinator
```

We note that in the above protocols, there is a clear separation between the responsibilities of the user (or application) and those of the system. Thus, the concept of when a given participant is *DONE* is entirely an artifact of the user, and is never visible to the system. On the other hand, when a process (presumably the coordinator) wishes to commit the parent possibility of the transaction, it simply executes the command **possibility$complete**; the system takes care of committing all of the actions taken (tokens created) under the possibility, wherever they may be located. (This is why, in the above procedures, I have not shown the final phase of the transaction, when the coordinator enters the *C-COM* state and informs the participants thereof.)

The above protocols were shown for transactions with fixed structure only. For conversational transactions, the various types of protocols described in Section 4.3 can be implemented in a very similar way.

Dynamic Transaction Restructuring. Certain kinds of transaction restructuring can be done if "dependent possibilities" are used (as in the above protocols). For example, while a participant in the three-phase commit protocol above is awaiting a *GET-DEPENDENT* message from the coordinator, it may instead receive an instruction to abort its local dependent possibility. Note, however, that arbitrary transaction restructuring, that involves the *"selective aborting"* of individual actions, is not possible once the actions in question become directly dependent on the parent possibility of the transaction.

"Forgetting" Transactions. The "commit record" associated with a transaction (possibility) can be reclaimed once it has established that all tokens dependent on it have "encached" the final outcome of the transaction; the commit record sends a *state* message to the object manager in charge of the token, and receives a *no-ref* message in response. As indicated in Section 3.3, the latter message may be lost, and the object manager may receive duplicate *state* messages from the commit record; to ensure that the latter can be reclaimed, the object manager will always acknowledge with a *no-ref* reply, even if the token in question has been aborted or erased because it is out of date. Thus, the object managers "remember" sufficient information to enable them to respond to duplicate commit or abort instructions from the commit record.

Increasing Resiliency. Reed allows for a possibility to be implemented using *multiple* commit records at several sites, with a *voting* scheme to decide the final outcome (complete or abort) of the possibility. This scheme has the effect of making participant processes (object managers that have had "tokens" created under the possibilitiy) not dependent on any single site. This increased resiliency is, naturally, offset by the increased overhead involved with collecting the votes, etc., and reclaiming the commit records once the final outcome has been decided.

5.3. Lampson and Sturgis

There are really two schemes that have been put forward by this pair of authors [5, 6]. The more recent one [6] seems to be the more general one and is the one I shall focus on; however, I shall also examine some of the features of the older algorithm [5]. The context that Lampson and Sturgis assume is one in which all transactions are *conversational*, i.e., the user enters requests for actions one by one, and at some point decides that he is satisfied and requests that his transaction be committed.

The basic scheme of [6] is very similar to Reed's basic scheme (i.e., without "dependent possibilities"). Thus, each *Write* action executed by a participant (server) process becomes "strongly dependent" (since the "intentions" get written immediately and cannot be erased except on command of the coordinator). This *"strongly-dependent"* protocol does not provide much resiliency, as was pointed out in Section 4.3, and Lampson and Sturgis propose some variations on their basic scheme. In particular, they suggest that a participant could defer acting on *Write* requests (and writing the associated intentions) until the coordinator sends a *"GetReady"* request. This begins to resemble the "three-phase" protocol, but isn't quite the same. In particular, Lampson and Sturgis do not seem to allow the execution of *Write* actions on a "volatile" basis, without being dependent on the coordinator; thus, the concept of a participant being *DONE* but not *DEPENDENT* is not apparent. It is at this point that the schemes of Lampson and Sturgis and Reed differ; Lampson and Sturgis do not provide the full power that can be obtained with Reed's "dependent possibilities".

We note that the older protocol of Lampson and Sturgis [5] is truly three-phase: a participant does go through a *DONE* state (or *"Prepared"*, in Lampson and Sturgis' terminology) in which it is not yet *DEPENDENT* on the coordinator (i.e., not *"Ready"*). (In the *DONE* state, the participant does not write its intentions until the coordinator sends a *"GetReady"* (*GET-DEPENDENT* in my model) message, but this could be improved somewhat by having the participant write the intentions while awaiting the coordinator's *GetReady* message, on receipt of which all the participant has to do is set its state to *"Ready"* (*DEPENDENT*).)

"Forgetting" Transactions. This issue does not seem to be explicitly addressed by Lampson and Sturgis; there seems to be an implication in [6] that neither the coordinator nor the participants can ever "forget" a transaction. (Note that the actual processes involved in the transaction need not stay around forever; it is necessary, however, for their home sites to forever retain a record of the transaction.) However, we note that the older version of Lampson and Sturgis' scheme [5] does allow both the coordinator and the participants to forget a transaction. This is done using the "implicit" approach that was mentioned in Section 3.3:

- If after committing, the coordinator finds a given participant missing, it can assume that the participant must have committed before vanishing, and the coordinator itself can therefore vanish.
- To abort the transaction, the coordinator can just vanish without sending any *ABORT* messages. A participant that finds the coordinator missing will then abort its local actions, since it can assume that if the coordinator vanished without persistently sending *COMMIT* messages, it must have aborted the transaction.

The ability of the coordinator and participants to do the above depends heavily on the idea of the sender of a message obtaining a *"Process-Missing"* reply from the home site of a vanished destination process; if the sites in the system were not equipped to give such a response, the above ideas would not be applicable. (Note that a *Process-Missing* reply is in effect some kind of communication from the vanished process, and the sending process makes certain inferences from this communication that allows it to take the actions described above. The situation is very different when a sending process finds the destination site down or does not receive any reply for a very long time; in such a situation, the sending process cannot assume that the destination process vanished, and therefore cannot take the actions described above. Thus, a participant in the *DEPENDENT* state is in fact dependent on the coordinator, and cannot unilaterally abort; it must await a message from the coordinator or a *Process-Missing* indication from the coordinator's home site.)

5.4. Takagi

Takagi's distributed transaction management scheme [10] is characterized by the idea of "early release" of uncommitted transaction results. I shall not discuss the details of this idea, except insofar as it affects the transacton commit procedure. To see how Takagi's scheme fits the model, let us consider a participant process in a transaction, in particular a data manager responsible for an entity (data item, data object) that the transaction updates. The participant goes through three phases with respect to the transaction:

- *Phase 0*: When the transaction has accessed the entity but has not updated it, i.e., the entity version for the transaction is *"dirty"*.

- *Phase 1*: When the transaction has updated the entity, i.e., the corresponding version is *"dependent"*.

- *Phase 2*: When the entity version is *"prepared"* for commitment.

It should be clear that this corresponds almost exactly to the "extended three-phase" commit protocol for conversational transactions (Figure 6); Takagi's states *"dirty"*, *"dependent"*, and *"prepared"* correspond to the *START*, *DONE*, and *DEPENDENT* states, respectively, in my model. (Note that the term "dependent" in used with different meanings in the two schemes - Takagi's *"dependent"* refers to the point where the transaction's output can be released to other transactions, while my *DEPENDENT* refers to the point where the participant must obey the coordinator's instructions to commit or abort.)

(The correspondence between Takagi's protocol and my extended three-phase protocol (Figure 6) may not be immediately obvious. In particular, there is no *GET-DONE* message from the coordinator to the participant in Takagi's scheme, that tells the participant that the transaction will make no further changes to the objects accessed. However, the *DONE* state of the participant is implicit in Takagi's assumptions about the transactions in the system, namely that a transaction's accesses to a given entity will either consist of a single "read-only" access, or a "read-for-update" followed by a "write" access. Thus, when the participant receives a "write" request from the transaction coordinator, there is an implicit *GET-DONE* in the request that allows the participant to infer that the transaction will not update the entity any further and that the new entity version can therefore be released to other transactions. I believe that Takagi's restrictions are not really fundamental to his idea of early data release and control of cascaded backout; it should be possible to relax his restrictions and allow a transaction to make multiple accesses to the same entity. Then, Takagi's ideas could be used in exactly the same way if the transaction coordinator "piggy-backed" a *GET-DONE* request with the last access that the transaction made to an entity (if the coordinator does not yet know whether or not a given access is the last one, it could send a separate *GET-DONE* later). A participant process would then know when the transaction's outputs are not subject to further change and are therefore ready for release to other transactions.)

Early Data Release. Takagi's scheme of early data release, and the cascaded backout that goes with it, places an important constraint on the commit protocol. In particular, since the *DEPENDENT* state implies that the participant will obey only the coordinator's instructions (to commit or abort), a participant in a transaction cannot enter the *DEPENDENT* state (i.e., make the entity version *"prepared"*) so long as the transaction is subject to cascaded backout. Thus, if transaction T_2 reads the results of transaction T_1, then T_2 cannot enter the *DEPENDENT* state until T_1 commits. In Takagi's scheme, this is reflected in the procedure that a participant follows when it receives a *"prepare"* message (*GET-DEPENDENT* in my model) from the coordinator of transaction T_2. The participant does not enter the *"prepared"* state (*DEPENDENT* in my model), and inform the coordinator thereof, until it has learned that the transactions on which T_2 depends, T_1 in this example, have committed.

5.5. SDD-1

The main feature of the SDD-1 design [4] is that its model of transaction execution is very different from that which I have been assuming so far. In particular, the "participant" sites involved in reading a transaction's inputs and computing its outputs need not be the same as the sites at which the outputs (updates to data) will actually be installed. Transaction execution thus takes place in two disjoint stages: (1) ***reading the*** inputs (which involves synchronization) and computing the outputs of the transaction, and (2) *committing the outputs*, i.e., installing updates to data and delivering requested data to the user. The first stage essentially corresponds to "Phase 0" of the three-phase commit protocol; when this stage is over, the transaction is *DONE* in the sense that all of its results are known.

The second stage of transaction execution, committing the results, is the one that I examine here. This stage essentially covers phases "1" and "2" of the three-phase protocol: An initial *UPDATE* message informs a participant (data manager) of the values for data update, and is like my *GET-DEPENDENT* since the participant must then await the final *COMMIT* (or *ABORT*) message. However, there is a major difference here, in the assumptions about the environment, between the SDD-1 commit protocol and most others. In particular, data managers holding data to be updated by a transaction (the "participants" in the transaction commit) cannot refuse to install the updates (i.e., cannot "unilaterally abort"). This allows the transaction coordinator to "force" a given data manager into the *DEPENDENT* state (which it does when it sends the initial *UPDATE* message), so that the participant must await a *COMMIT* or *ABORT* message. Even if the participant is down while the coordinator is attempting to commit, the initial *UPDATE* message for the participant can be "spooled"; the participant on recovery must first read all of its spooled messages, at which point it will be dependent on the coordinator. Thus, in the SDD-1 commit protocol, the coordinator is not dependent on the participants' cooperation in order to commit a transaction's results. (Note that this only applies to the *commit* procedure, *after* the transaction has read its inputs and computed its results. *Before* the commit procedure begins, a transaction is dependent on the availability of its input data and on satisfying its synchronization requirements.)

Increased Resiliency. To take care of coordinator crashes while a participant is in the *DEPENDENT* state, the basic commit protocol is extended in SDD-1 by using *backup* coordinators and adding an extra phase of message exchanges (which ensures agreement between the current coordinator and all of the backups). (In an older version of the SDD-1 commit protocol [3], a participant that found the coordinator down would *inquire* of the other participants whether or not they had committed their results. This solution was ultimately rejected because the mechanisms necessary for achieving agreement among the various participants (especially when the only participant(s) that committed has also crashed and therefore will not respond to inquiries) are somewhat more complex and involve longer delays than the use of "backup" coordinators.)

5.6. Montgomery

The scheme of Montgomery [8] uses a hierarchical network structure to provide transaction atomicity with reduced locking reqirements. The ideas that are of interest to us here are the use of *polyvalues* and the associated distributed commit protocol. I shall focus first on what Montgomery calls *predictable* transactions, or, in my terms, transactions with *fixed structure*. In Montgomery's commit protocol, each participant (or at least the ones that update data) goes through two phases of processing:

1. A *"lock phase"*, during which the participant can abort its local actions. This phase ends when the participant completes its local processing and sends a *"ready"* message to the coordinator.

2. A *"wait phase"*, during which the participant must await a *"complete"* or *"abort"* message from the coordinator regarding the outcome of the transaction. During the wait phase, the participant knows that it will not have to perform any further processing on behalf of the transaction.

It should be clear that this procedure corresponds to the "two-phase" protocol for transactions with fixed structure (illustrated in Figure 3): The "lock phase" is "Phase 1", before the participant's *DEPENDENT* state, while the "wait phase" is "Phase 2", when the participant is *DEPENDENT* on the coordinator (and also knows that it is *DONE*).

Early Data Release. The feature that distinguishes Mongomery's procedure from most other distributed transaction management schemes is that a participant can release a transaction's results before learning (from the coordinator) whether the transaction committed or aborted. This "early release" takes the form of *polyvalues*, which describe the possible values of a data object under each possible combination of outcomes (commit or abort) of the pending transactions on which the polyvalue depends. During the *DEPENDENT* state, a participant also knows that its actions on behalf of the transaction are *DONE*, and it therefore knows what values the data updated by the transaction will have if the transaction eventually commits; since the participant retains the old values of the data, it can construct polyvalues which other transactions can read.

Then, when a transaction T_2, say, reads the polyvalue output of transaction T_1, its own completion is not dependent on T_1's finally completing or aborting. Thus, the problem that appeared in Takagi's scheme [10], that T_2 could not enter the *DEPENDENT* state until T_1 committed (because of the possibility of cascaded backout), does not appear in Montgomery's scheme.

Regarding early data release and the *DONE* state, Montgomery's treatment of *unpredictable* transactions is fairly interesting. Montgomery uses *"completion weights"* to allow a transaction coordinator to determine when an unpredictable transaction's actions are complete (i.e., all participants are *DEPENDENT*) and can be committed, even though the coordinator did not know in advance exactly who the participant processes were. Now in an unpredictable transaction, a participant process immediately enters the *DEPENDENT* state when it performs an action (such as an update), but never knows for certain that it is *DONE* until it receives a commit (or abort) message from the coordinator. (The protocol is thus "strongly dependent" (Figure 6).) Now Montgomery allows a participant in such a transaction to release the results of the transaction (in the form of polyvalues) at any time, which means that the transaction's outputs are released before the participant is certain that it is *DONE*. I claimed in Section 4.5 that this is generally a bad idea (since the results are "dirty"), but Montgomery has an interesting way of getting around the problem. The participant releasing the polyvalue has no idea whether or not it is *DONE*, but the coordinator will at some point know whether or not it requires further processing from the participant that release the polyvalue; at this point, there are two possibilities:

1. The coordinator does *not* require further processing from the participant. Then, the participant is in fact *DONE*, and the "completion weight" scheme will allow the coordinator to commit the transaction (barring failures and other undesirable events elsewhere).

2. The coordinator *does* require further processing from the participant, in which case it will send further requests to the participant. However, the participant, having released the outputs of the transaction, will refuse any such requests, and the coordinator will ultimately have to abort the transaction since it cannot accumulate the required completion weight. (The fact that the transaction will eventually abort does not affect any other transactions that read its output data, since these transactions will also have been supplied the "old" value of this data as part of the polyvalue.)

In effect, what the participant does when it releases data before knowing that it is *DONE* is to "freeze" the data into the *DONE* state; then, if the coordinator finds this unsatisfactory, it is forced to abort the transaction.

"Forgetting" Transactions. We note that the use of polyvalues makes the problem of forgetting a transaction somewhat more complicated, since not only must the participants in a transaction be informed of the transaction's outcome, but so must also the processes (data managers) that were given polyvalues that refer to the transaction. Montgomery's solution to this problem is fairly simple: In the same way that the coordinator is responsible for informing each participant of a transaction's outcome, each data manager is responsible for informing each other data manager to which it supplied polyvalues. This simply distributes the responsibility for propagating the outcome of a transaction. (The problem described in Section 3.3, of a process receiving duplicate commit or abort messages for a given transaction, is handled in Montgomery's scheme at the *network* level, by providing *robust sequenced processes* with no duplicate messages.)

6. SUMMARY

I have attempted in this paper to present some kind of uniform model for describing distributed transaction commit protocols. The model was used to identify certain *classes* of commit protocols (such as "two-phase" and "three-phase"), with different resiliency and other properties, and individual schemes from the literature were matched with the identified classes. In most cases, it was found that the scheme being examined fitted a single protocol class. However, the scheme of Reed [9] was shown to provide considerable flexibility by allowing the user to enforce any class of protocol for determining when to commit a transaction, with Reed's system providing the underlying support needed to ensure that the final decision to commit or abort was correctly and consistently enforced. (The scheme of Lampson and Sturgis [6] also provides this support, but it does not allow for the broad range of protocols that Reed's scheme does.)

I examined several issues that arise in connection with distributed transactions, in particular: the *resiliency* of the various protocol classes, and existing methods for improving resiliency; how dynamic transaction *restructuring* might be performed and the problems associated with it. I also examined how differing assumptions about the environment affected individual protocols, for example *early release* of data in [10, 8], and the inability of data managers to refuse update commands in SDD-1 [4].

Most importantly, I attempted to formalize certain recurring themes that appear in the vast literature on the subject, namely:

1. The *two-phase* requirement for distributed transaction commit protocols: a semi-formal argument was presented indicating why this requirement is essential. Also, the basic *resiliency* problem that arises in consequence of the two-phase requirement was identified: Once a participant has signalled to the coordinator its ability to commit (i.e., that it is in the *DEPENDENT* state), it must await the coordinator's instructions to commit or abort, and is therefore susceptible to a failure of the coordinator's home site. Some proposed methods for alleviating this problem were briefly mentioned.

2. *"Forgetting" transactions*: It was established that it is not in general possible for both the coordinator and the participants to erase all memory of a transaction once it has been committed or aborted. With a few exceptions (such as [9]), this problem has not received much attention in the literature; the kinds of techniques commonly used to deal with it were briefly described.

FIGURES

Figure 1: SCENARIO FOR DISTRIBUTED TRANSATION PROCESSING.

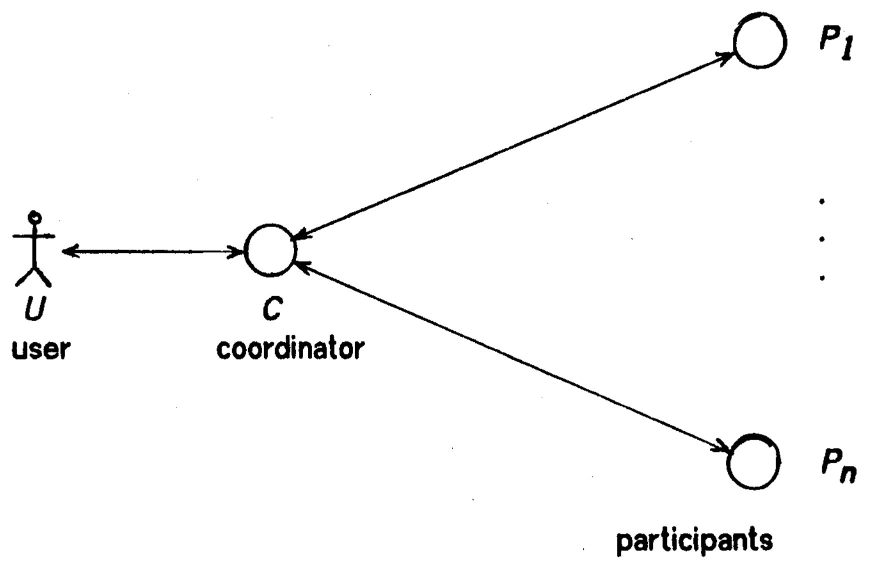


Figure 2: BASIC STATE MODEL.

Participant:

STATE	ACTIONS
P-START (failure-state *P-ABORT*)	varies
P-DEP	send *DEPENDENT* message; await *COMMIT/ABORT*
P-COM	commit local actions; inform coordinator
P-ABORT	abort local actions; inform coordinator
P-MISSING-OK	none
P-MISSING-X	none

allowable transitions:

$$\begin{tikzcd}
\textit{P-START} \arrow[r] \arrow[dr] & \textit{P-DEP} \arrow[r] \arrow[d] & \textit{P-COM} \arrow[r] & \textit{P-MISSING-OK} \\
 & \textit{P-ABORT} \arrow[rr] & & \textit{P-MISSING-X}
\end{tikzcd}$$

Coordinator:

STATE	ACTIONS
C-START (failure-state *C-ABORT*)	varies
ALL-DEP (failure-state *C-ABORT*)	get final user *OK*
C-COM	send *COMMITs* to participants, send *OK* to user
C-ABORT	send *ABORTs*, etc.
C-MISSING-OK	none
C-MISSING-X	none

allowable transitions:

$$\begin{tikzcd}
\textit{C-START} \arrow[r] \arrow[dr] & \textit{ALL-DEP} \arrow[r] \arrow[d] & \textit{C-COM} \arrow[r] & \textit{C-MISSING-OK} \\
 & \textit{C-ABORT} \arrow[rr] & & \textit{C-MISSING-X}
\end{tikzcd}$$

Figure 3: TWO-PHASE COMMIT PROTOCOL.

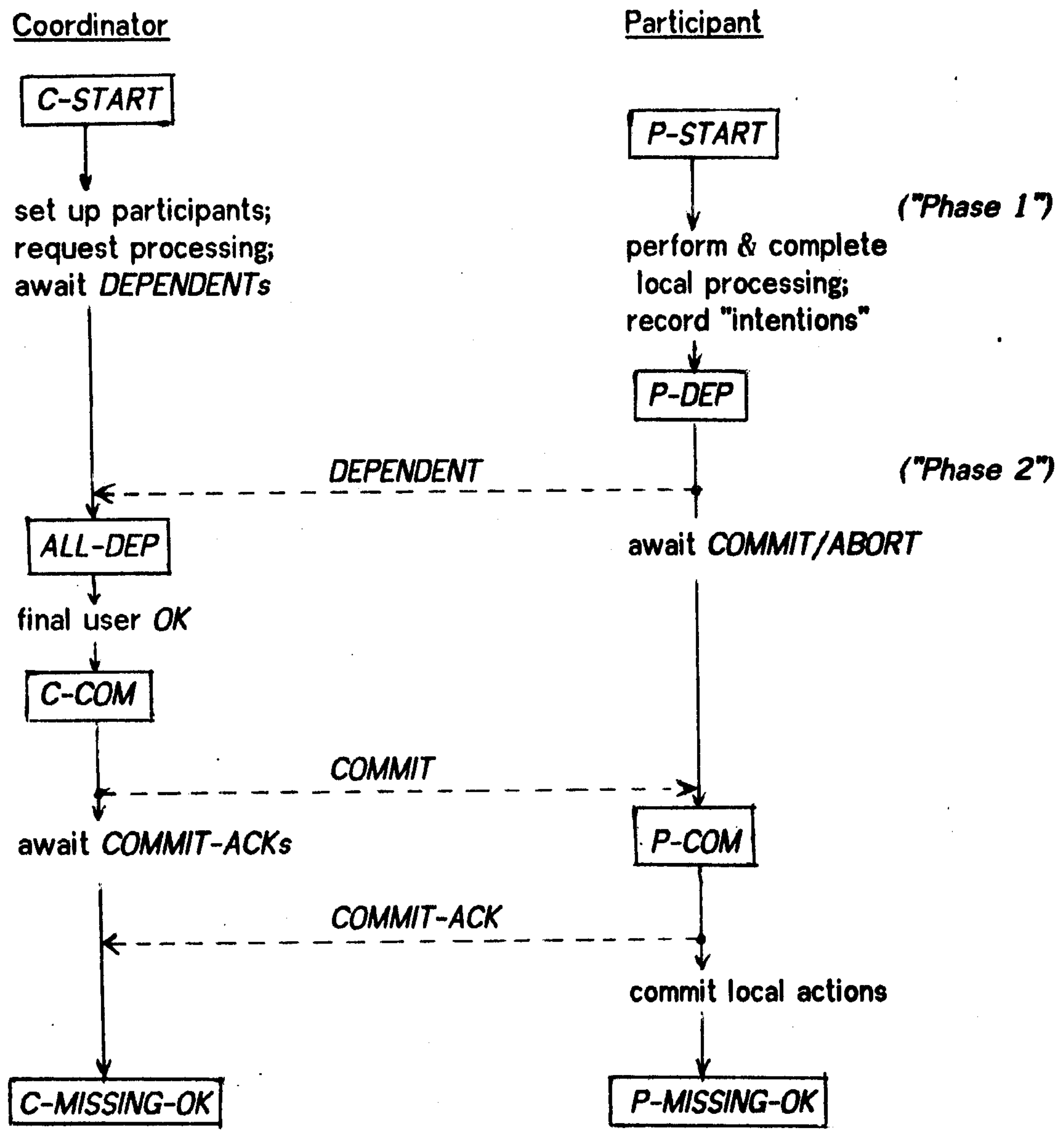


Figure 4: EXTENDED STATE MODEL.

Participant:

STATE	FAILURE-STATE
<-,-> (or *P-START*)	*P-ABORT*
<*P-DONE*,->	*P-ABORT*
<-,*P-DEP*>	<-,*P-DEP*>
<*P-DONE*,*P-DEP*>	<*P-DONE*,*P-DEP*>

P-COM,*P-ABORT*,*P-MISSING-OK*,
P-MISSING-X: see Figure 2

Coordinator:

STATE	FAILURE-STATE
<-,-> (or *C-START*)	*C-ABORT*
<*ALL-DONE*,->	*C-ABORT*
<-,*ALL-DEP*>	*C-ABORT*
<*ALL-DONE*,*ALL-DEP*>	*C-ABORT*

C-COM,*C-ABORT*,*C-MISSING-OK*,
C-MISSING-X: see Figure 2

Figure 5: THREE-PHASE COMMIT PROTOCOL.

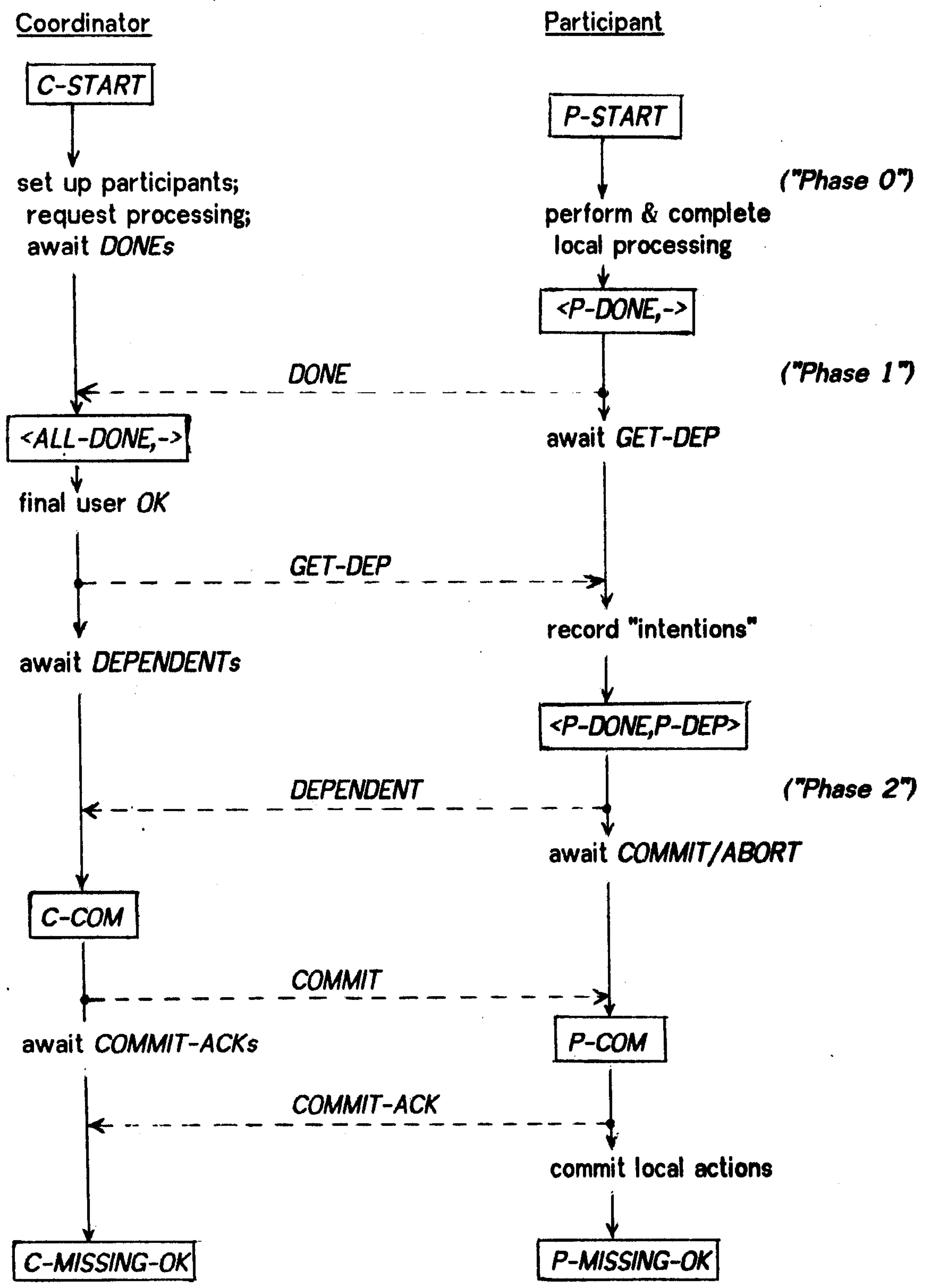


Figure 6: COMMIT PROTOCOLS FOR CONVERSATIONAL TRANSACTIONS.

1. *Strongly Dependent*:

participant:

```
          P-START (<-,->) ⟶ perform request ⟶ ...
   ... ⟶ <-,P-DEP>      ⟶ receive COMMIT ⟶ ...
   ... ⟶ P-COM          ⟶ commit local actions
```

coordinator:

```
          C-START (<-,->) ⟶ user requests commit (assuming ALL-DONE) ⟶ ...
   ... ⟶ C-COM          ⟶ send COMMITs
```

2. *Standard*:

participant:

```
          P-START         ⟶ perform requests, receive GET-DEP ⟶ ...
   ... ⟶ <P-DONE,P-DEP> ⟶ receive COMMIT ⟶ ...
   ... ⟶ P-COM
```

coordinator:

```
          C-START         ⟶ get ACKs of requests ⟶ ...
   ... ⟶ <ALL-DONE,->   ⟶ user requests commit, send GET-DEPs,
                             receive DEPENDENTs ⟶ ...
   ... ⟶ C-COM          ⟶ send COMMITs
```

3. *Extended*:

participant:

```
          P-START         ⟶ perform requests, receive GET-DONE ⟶ ...
   ... ⟶ <P-DONE,->     ⟶ send DONE, receive GET-DEP ⟶ ...
   ... ⟶ <P-DONE,P-DEP> ⟶ receive COMMIT ⟶ ...
   ... ⟶ P-COM
```

coordinator:

```
          C-START         ⟶ get ACKs of requests, user OK;
                             send GET-DONEs, receive DONEs ⟶ ...
   ... ⟶ <ALL-DONE,->   ⟶ send GET-DEPs, receive DEPENDENTs ⟶ ...
   ... ⟶ C-COM          ⟶ send COMMITs
```

References

[1] Davies, C. T.
Recovery Semantics for a DB/DC System.
In *Proc. ACM National Conference, 1973.*

[2] Gray, J. N.
Notes on Data Base Operating Systems.
In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, Springer-Verlag Lecture Notes on Computer Science, Vol. 60, 1978.

[3] Hammer, M., and Shipman, D. W.
An Overview of Reliability Mechanisms for a Distributed Data Base System.
In *Proc. IEEE COMPCON Spring 1978.*

[4] Hammer, M., and Shipman, D. W.
Resiliency Mechanisms in SDD-1: A System for Distributed Databases.
Computer Corporation of America Technical Report in preparation.

[5] Lampson, B., and Sturgis, H.
Crash Recovery in a Distributed Data Storage System.
Xerox Palo Alto Research Center, November 1976.

[6] Lampson, B., and Sturgis, H.
Crash Recovery in a Distributed Data Storage System.
to appear in *Communications of the ACM.*

[7] Liskov, B. H., and Snyder, A.
Structured Exception Handling.
M.I.T. Laboratory for Computer Science Computation Structures Group Memo 155, December 1977.

[8] Montgomery, W. A.
Robust Concurrency Control for a Distributed Information System.
M.I.T. Laboratory for Computer Science Technical Report TR-207, December 1978.

[9] Reed, D. P.
Naming and Synchronization in a Decentralized Computer System.
M.I.T. Laboratory for Computer Science Technical Report TR-205, September 1978.

[10] Takagi, A.
Concurrent and Reliable Updates of Distributed Databases.
M.I.T. Laboratory for Computer Science Computer Systems Research RFC-167, November 1978.

Crash Resistance and Recovery in NAMOS

by Andrew Mendelsohn

Professor Saltzer
6.845
May 22, 1979

1. Introduction

During the past semester, the Computer Systems Research Seminar has examined a number of papers describing mechanisms for obtaining atomic transactions in a distributed system. The majority of these papers focus most of their attention on the consistency and information hiding aspects of their mechanism, while giving only a very high-level, hand-waving sort of argument concerning the performance of their mechanism in the face of failures. Lampson and Sturgis [1] authored the only paper that described its system primarily from the viewpoint of failure resistance and recovery and that provided a rigorous description of its behavior in the face of a fairly well-defined set of failures. In addition, what really sets their paper apart from all others, is that the effects of failure are specified and coped with at <u>all levels</u> of their system.

In this paper, we will attempt to describe Reed's system [2] using the methodology developed by Lampson and Sturgis. More than most, Reed is fairly careful in addressing the issues of failure; however, the thesis' sheer volume tends to cause the tidbits of information relevant to failure recovery to be scattered throughout the thesis. No coherent argument for the crash resistance of the *full* system is given. In this paper, we will attempt to construct such an argument by pulling together all the relevant information provided by Reed and filling in any loose ends that we may discover.

After completing this exercise, we will attempt to compare the lattice of abstraction required by Reed with that needed by Lampson and Sturgis. In particular, we would like to examine the validity of the statement Lampson gave in our seminar to the effect that the notion of *compatible actions* is lurking behind the correctness of all distributed systems and that other workers in the field have not mentioned this concept, simply because they have not thought deeply enough about the problem. Finally, we will comment on the relationship between Reed's and Lampson and Sturgis' atomic commit and transaction mechanisms.

2. The Physical System

Reed's physical system consists of a set of autonomous nodes (each of which behaves like a single system in itself) connected by some communications medium. The few details he gives us are very much in line with the physical system model given by Lampson and Sturgis, and hence for rigor we will assume their model as the bottom level of Reed's system. In their model, each node may have a data storage system and a processor. The processor consists of a collection of processes and some shared state. The processor may access the data storage system in units of a page. Additionally, the processors may exchange blocks of data over a communications system. Finally, the failure modes of the processor, disk, and communications system are elaborately specified. The reader is referred to [1] for further details of their model.

3. The Object-oriented System

This level of Reed's system is rather different from Lampson's stable system. Lampson allows individual nodes to randomly access pages and invoke actions at other nodes. This is too unconstrained for Reed's world of autonomous nodes and hence a more structured object-oriented model is used.

3.1 Stable Objects

The goal of the stable object abstraction is to provide stable, long term storage of objects at a single node. These objects are required for the implementation of commit records, known histories, and version values in the NAMOS level of the system. For the sake of completeness, we will outline a straightforward implementation of objects that can be built on top of Lampson and Sturgis' stable storage. [1].

We will support the usual notion of an object where each object has some fixed type, and all operations on the object are performed through invocations of its type manager. Each object has a *name* which consists of its type and an index into a *universal object context*. The context is simply an array stored in a known location in stable storage. Each array entry contains an address of a page in stable storage that is the root of the indexed object's representation. A root page can be generalized to a rooted tree of pages if need be, but we will ignore this added complication in what follows. Each root page contains an array of names of lower-level objects that forms the representation of the top-level object.

This level of the system makes no general guarantees about the atomicity of operations on the objects it provides. Side-effect-free operations are by nature atomic as are operations that require only one stable put to a "visible" object (as opposed to a temporary object only reachable from volatile storage). Fortunately, it turns out that the NAMOS level only has two simple update operations to objects that are required to be atomic: a change in state of a commit record and a deletion or insertion of a version into a known history. In both these cases, the updates can be done with a single atomic stable put to a visible object.

There is one final nicety needed here. Operations on objects that create and possibly return new objects may leave around unreachable stable pages and unused stable objects (the invisible objects of above) if a failure occurs during the course of their execution. This is no real problem, though, as the unused objects can be garbage collected by scanning the NAMOS-level known histories and all objects reachable from the histories. The unused pages can in turn be garbage collected by scanning the stable page representations of all reachable objects. There is no problem of multi-node garbage collection since, objects at this level can not be referenced from outside their home node.

3.2 Monitored Stable Objects

This level of the system also provides monitors to mediate operations to shared, mutable objects. Each monitor has a lock in volatile storage. Monitor operations that are atomic will be denoted by a heading of the form

<opname> = atomic proc(....) .

To avoid deadlock, we disallow nested monitor calls (i.e. no process can hold more than one lock), and any wait done during a monitor call forces the waiting process to temporarily release the monitor lock.

In his thesis, Reed suggests a simple locking scheme for providing atomic actions on mutable objects. In this scheme, the object to be modified is locked and then copied to whatever level necessary. All required changes are made to the copy. Finally, the copy is atomically substituted for the original as the lock is released. This operation, though, will not always be atomic under the object model: if more than one of the lower level objects contained in the locked object must be mutated, then a crash occurring before all the objects are mutated will leave some changed and some not. Since the atomic stable page put seems to be adequate for our purposes, we will not go into this any further.

Our approach also differs somewhat from that of Reed in our use of the term stable storage. Reed's notion of stable storage admits the possibility that in the event of a crash during an update, the state of the updated storage may be detectably lost. Since his weaker form of stable storage is required to be reliable in the long-term, under Lampson and Sturgis' physical storage model, it can only be implemented by some form of replication. Hence, we can see no benefit from using this type of stable storage.

3.3 Message System

Only minimal message passing facilities are needed for the implementation of the next level's internode communication protocols. The message system will provide a simple form of remote procedure call capability, but unlike Lampson and Sturgis' system, remote procedure calls need not be used to the exclusion of other possible protocols.

The form of remote procedures required is very weak and is more appropriately described as a capability for reliably sending request messages and then matching them with their responses. This can be easily implemented by tagging messages requiring such matching with unique identifiers. The UID's must of course be unique in spite of processor crashes, but any message system data structures required for matching messages may be volatile.

When we wish to denote a request/reply message pairing in our programs, we will use a special form of syntax. For the requestor, we will write

```
send <msgname>(<arglist>)
<msg> := awaitreply()
```

This syntax also indicates that the send may be retransmitted if the reply does not arrive promptly.

We will assume all remote request servers are embodied as special *message handler* processes. These processes will be denoted by the following canonical form:

```
foo = message_handler
        while true do
                case awaitmessage of
        <msgtype1> => ...  sendreply <msg>(<arglist>) ...
        <msgtype2> => ...               :
          :                             :
        end
end
```

There is no requirement that the message handler respond to every request it receives, nor is there any retransmission mechanism for replies. Lost replies will be recovered from by retransmission of the request.

4. NAMOS

In this section we show how to implement pseudo time, known histories, and possiblilities. Following the format of [1], we will give informal CLU-like code that implements these objects. We diverge from real CLU in that 1)we use monitors and scalar types, 2)we consider variant records (oneof's) to be mutable, and 3)we allow message system calls of the form mentioned in the last section. Our code is somewhat informal in that we do not define clusters (type managers) for all lower level objects used, but instead operate directly on the objects' representations.

4.1 Pseudo Time

Pseudo time is straightforward to implement, and as long as it is monotonically increasing with time, it cannot affect system correctness. It suffices to say that it requires a stable, monotonically increasing clock source.

A pseudo temporal environment is likewise trivial to implement, and so there is no point giving code describing its operations. A pseudo temporal environment may be stored in a cell in volatile storage.

From the viewpoint of crash resisitance and recovery, pseudo time and pseudo temporal environments are uninteresting. Pseudo temporal environments are volatile and pseudo time is recovered by resysnchronization with some outside source.

4.2 Known Histories

In appendix I, we give code that implements the linked-list representation of known histories suggested by Reed. A known history is represented by a single header object which points to a linked list of version objects. Each known history is also protected by a monitor. The update operations do, at times, require more than one stable put to modify a known history; however, the known history object is always left in a correct and consistent state after each such put. Hence all known history operations are impervious to processor failure.

A few minor points about the implementation should be noted. When a process enters a known history monitor but finds that it must wait for the completion of a token before it can proceed, it must release the monitor and wait. Therefore, when the process is awakened after the token is aborted or completed, there is no guarantee that any of the process' local state that refers to the known history is of any use. So the process must start over from the beginning of the monitor again.

To avoid a similar problem, new tokens are created without waiting for the go-ahead (a *permit-create* message) from their commit records.

Known history (objectref) monitors are invokable only from from the version reference manager at the known history's home node. The version reference manager, however, can be invoked by remote requests to perform lookups and defines on versions. We will not give code for the version reference manager here, but will briefly outline how it can be implemented.

The version reference manager processes requests to perform lookups and defines on versions by spawning processes that call the try to lookup and try to define entries of the appropriate known history monitor. The manager also processes *state* messages, which are requests to encache the state of a commit record in a token, by calling the appropriate set state monitor entry and then sending a *no-ref* message back to the requesting commit record if the state is successfully encached. Since the only kind of object reference meaningful across nodes is a version reference, a value returned by a lookup request or passed by a define request must be purged of all local object references before it can be sent between nodes. The standard solution to this problem is to encode all local objects in some standard intermediate representation before transmitting them across nodes. The matching decode operation is then performed at the receiving node. Version references will, of course, pass through the encode and decode operations intact.

4.3 Possibilities

An implementation of possibilities is given in appendix II. We use a single monitor for all commit records of a given node. The commit records are stored in a stable set. In order to show more clearly the message passing coordinated by this monitor, we have taken the liberty of using the message handler syntax (see section 3.3), rather than the usual procedure-oriented syntax, to describe its activities.

We have implemented Reed's optimized form of single site commit record. All tokens that send *permit_create* messages to the commit record are remembered in a volatile set at the commit record site. When the commit record completes or aborts, it sends encache *state* messages to all tokens in this set. When the commit record receives *no_ref* responses from all the commit records in the set, it is deleted from its node's stable commit record set.

An inspection of the code for possibilities indicates that all monitor operations leave each commit record in a consistent state in spite of failures. This assumes that we are using a set representation such as Lampson and Sturgis' stable sets, where the operations insert, delete, and retrieve are atomic.

Since we are mainly interested in comparing Reed's scheme with Lampson and Sturgis', we will not bother looking into multi-site commit records or dependent possibilities in this paper.

4.4 Transactions

Finally atomic transactions are obtainable with the following:

```
    in pte$transaction do
            possi := possibility$create(7)
            were possi do
                    <statements>
            end
            possibility$complete(possi)
    end
```

This code guarantees to make <statements> atomic with respect to both failures and other transactions, assuming that no other possibilities or pseudo temporal environments are employed within <statements>.

5. Observations

Now that we have gone through with this rather tedious construction, what can we learn from it? For a start, we can now construct a lattice of abstractions for the system with some degree of confidence that we have not left out anything of importance.

5.1 Lattice of Abstractions

We have constructed an abstraction lattice for Reed's system and it is pictured in appendix III. A casual glance at the lattice tells us that Reed's system is of about the same order of complexity as that of Lampson and Sturgis. A little closer inspection reveals that other than the physical level of the two systems, the only abstractions used in both systems are stable storage, monitors (Reed's thesis uses locks), unique identifiers, and stable sets (for a single node).

Moving from the bottom up, the first major divergence between the two systems occurs in the abstraction built on top of physical communications. Reed believes that the physical distribution of the system can never be successfully hidden, and that this fact of life must inevitably be visible all the way up to the user interface. In addition, based on the "end-to-end argument," he sees little point in providing more than basic message passing in implementing the upper levels of his system. On the other hand, Lampson and Sturgis' goal is to build an abstract system that looks to be centralized and non-distributed. Out of this basic philosophical difference, comes Lampson and Sturgis' remote procedure calls and Reed's primitive message system.

Lampson's remote procedures syntactically look exactly like normal procedure calls; however, true to Reed's beliefs, the fact that the remote calls are really implemented on top of a distributed system warps their semantics. Only remote procedures that are *restartable* act semantically like normal procedure calls. All inter-node communication in Lampson's system (above the base level) is done through remote procedure calls.

Reed, however, implemented his message system so that the decision of what message passing protocol is to be used is left to the next level of his system. The next level has the option of using a remote-procedure-call-like mechanism if need be; however, it may also construct arbitrary protocols to accomplish a given task. Application dependent knowledge can be used in place of low-level mechanisms (e.g., duplicate message suppression).

As a final comment on communications, Reed's message system does not seem to require any "initial connection extablishment" between a pair of processors after one of the pair crashes. This contradicts the belief of Lampson and Sturgis stated in [1] that in effect says that such a protocol is necessary for a distributed system to work properly.

A second major difference between the two lattices is that Lampson and Sturgis build up their lattice by constructing bigger and bigger atomic actions, while Reed does not. Rather, Reed's construction (at least under our implementation) goes from atomic stable pages to non-atomic stable objects, to non-atomic monitored stable objects, to atomic commit records, non-atomic known histories, and atomic stable sets, and finally to atomic transactions. Reed's system is building bigger and bigger operations on the shared data that maintain data consistency in the face of failure, rather than atomicity.

A third difference between the two lattices is the absence of the process save/restart capability in Reed's system. The only place this is needed to ensure correctness of Lampson and Sturgis' system is in the implementation of their two-phase commit protocol. It seems to be used primarily for efficiency reasons, as a passive scheme similar to Reed's basic method would certainly suffice. More will be said about this later.

Another motivation for Lampson and Sturgis' "save process state" capability is that it should be extremely useful at the application level of their system. This same statement is not as obviously true in Reed's system, as a transaction restarted in mid-stream stands a good chance of being forced to abort because of an out-of-date pseudo temporal environment.

Finally, processor crashes and communication unreliability make the notion of restartable actions necessary in both systems. Since all shared data in Reed's system is protected by monitors and the monitor operations are always guaranteed to leave their objects in a consistent state in spite of processor crashes, it appears that one need not worry about *compatibility* of actions as in Lampson and Sturgis' system. Since there is no restarting of "saved" processes after processor crashes, arguments about the correctness of Reed's system seem to only require that the integrity of its data structures is maintained between the crashes.

5.2 Crash Recovery in Namos

The processor crash recovery algorithm in Namos is as follows:

1. Cleanup stable storage as in [1].

2. (optional) Garbage collect unused local objects and stable pages unreachable from used local objects.

3. Rebuild all necessary volatile data structures (e.g. message system data structures, monitor locks).

4. Initialize standard system processes and procedures (e.g. message system processes, known history monitors, commit record monitors).

5. Resynchronize the pseudo time clock (e.g., with WWV).

6. Start accepting remote requests.

5.3 Atomic Transaction and Commit Mechanisms

The closer one looks at the atomic transaction mechanisms provided by Reed, and Lampson and Sturgis, the more they look the same. If Lampson and Sturgis used pseudo-time to order their transactions (something similar to Takagi), rather than ordering them by "first come, first serve," and if Reed restricted each known history to at most one old version and one token, then their two transaction schemes would be equivalent (commit schemes being equal). The commit schemes themselves are also remarkably similar. This has two major implications:

1. It should be straightforward to add "dependent" (a la Reed) intentions lists to Lampson and Sturgis' scheme to get a distributed locking mechanism that allows modular compositon of operations.

2. Reed's commit scheme can be used in place of Lampson and Sturgis' thereby eliminating those mysterious entities called compatible actions.

For practical purposes, however, we might in fact want to instead use Lampson and Sturgis' atomic commit mechanism in Reed's system, since it is much more efficient in terms of messages and delay than Reed's mechanism (at least for the single site commit record case).

Using the terminology of Reed's system, the Lampson and Sturgis mechanism translates to the following:

A commit record *create* request is sent by the transaction to some possibly remote node. The node creates a stable set that will be used to keep a record of all nodes (not all tokens as done by Reed) which contain tokens that depend on the commit record. The commit record state is set to *waiting*, and the commit record's name is returned to the requestor. At any given node N, when the first attempt is made to create a token, T, at that node that is dependent on a given commit record, C, a stable set is created at N that will be used to hold the names of all tokens dependent on the commit record. The token, T, is then put in the stable set and a message is sent to the commit record's home, instructing the commit record, C, to put the node, N, in its set of nodes. Further token requests sent to node N that are dependent on C only require insertion of the token into the node N's set of tokens dependent on C. The commit record is committed/aborted by sending a message to its home node. Upon receipt at the node, the commit record's state is atomically set. Commit messages are then sent to all nodes containing tokens dependent on the commit record. When these messages are all acknowledged, the stable set of nodes may be deleted. At the nodes with dependent tokens, the commit state is encached in stable storage and all dependent tokens are committed/aborted.

As can be seen from this description, Lampson and Sturgis' commit mechanism is basically an optimized version of Reed's. It cuts down on the number of required messages by 1)allowing the commit record to only notify a given node containing dependent tokens once of its state and by 2)allowing a node with possibly many dependent tokens to only notify the commit record once of their collective existence. The second clause implies that once a node knows about a given commit record, future requests to create tokens dependent on that record need not query the commit record.

6. Conclusion

Our intent in writing this paper was to develop a complete picture of the lower-level abstractions required by Reed's system. We were especially attuned to issues concerning crash resistance. By going through much of the nitty-gritty details of actually programming a subset of his full system, we feel that we have uncovered most of the relevant facts in this area. We were also able to determine a complete lattice of abstractions for Reed's system and to provide a simple procedure for providing processor crash recovery. Finally, we compared various aspects of Reed's and Lampson's systems and noted some differences and strong resemblances between the two systems.

7. Appendix

Appendix I: Known History Monitor

```
commit_state = (waiting, aborted, completed)

version = record[val: any, start, end: pseudot,
            possi: possibility,
            state: commit_state,
            next: histlink]

% objectref is a reference to a known history
objectref = monitor is create, delete, try_to_define, try_to_lookup, set_state;

      histlink = oneof[next: version, empty: null]

      rep = record[tcreate, tdelete: pseudot
               link: histlink]

      create = proc(pt: pseudot) returns(cvt)
            % disallow creation pending a possibility - seems ridiculous
            return (rep$(tcreate: pt,  tdelete: -1,
                     link: histlink$(empty: nil)))
            end create

      delete = proc(or: cvt, pt: pseudot) signals(bad_delete)
            % trivial, a bunch of special case checks
            end delete

      try_to_define = proc(vr: versionref, val: any, possi: possibility)
                  signals(non_existent_state,redefinition,forgotten_state)
            % Try to create a token
            % Assume versions never deleted so "forgotten_state" impossible
            prevlink: histlink
            or: objectref, pt: pseudot := versionref$decompose(vr)
restart:    prevlink := find_place_in_history(vr)
                  except when non_existent_state: signal non_existent_state
            tagcase prevlink
                  tag next(v: version) % v gets version pointed to by prevlink
                        if v.state = waiting then
                              % check for duplicate request
                              if v.start=pt and v.val=val and v.possi=possi
                                    then return end.
                              send test(possi,vr)
                              state(p: possibility,vref: versionref,
                                    s: commit_state) := awaitreply()
                               set_state(vr, s)
                               send no_ref(possi,vr)  % no wait send
                               goto restart
                          % else state is completed
                          elseif v.end >= pt then signal redefinition end
                          % else fall through to create token
                     tag empty:  % fall through to create token
                end
                % atomically insert token
                prevlink.next := version$(val: val, start: pt, end: -1,
                                    state: waiting, next: copy1(prevlink))
                send create_ref(possi, vr)
                permit_create(p: possibility, vr: versionref, ok: boolean)
                     := awaitreply()
                if ~ok then set_state(vr, aborted); send no_ref(possi,vr)
                          signal bad_possibility end
                end try_to_define

           try_to_lookup = proc(vr: versionref, possi: possibility) returns(any)
                          signals(nonexistent_state,forgotten_state)
                % similar to but simpler than try_to_define
           end try_to_define

           find_place_in_history = proc(vr: versionref) returns(histlink)
                               signals(non_existent_state)
                % Returns the link which points to the version whose start and
                % end times bracket pt or whose start time comes closest to but not
                % not after pt
                or1: objectref, pt: pseudot := versionref$decompose(vr)
                or: rep := down(or1)
                if pt < or.tcreate or (or.tdelete ~= -1 and or.tdelete < pt) then
                     signal non_existent_state end
                link: histlink := or.link
                while true do
                     tagcase link
                          tag next (v: version):
                               if v.state = aborted then splice_out(link)
                               elseif v.start <= pt then return(link)  end
                               else link := v.next
                          tag empty:  return(link)
                     end
                end
                end find_place_in_history

           set_state = proc(vr: versionref, s: commit_state) returns(commit_state)
                or: objectref, pt: pseudot := versionref$decompose(vr)
                link: histlink := find_place_in_history(vr)
                except when non_existent_state: return(aborted)
            tagcase link
                tag next (v: version):
                    if v.state = waiting then
                        v.state := s
                        if s = aborted then splice_out(link)  end
                    end
                    return(v.state)
                tag empty: return(aborted)
            end
            end set_state

        splice_out = atomic proc(previous: histlink)
            previous.next := previous.next.next
            end splice_out

end objectref
```

Appendix II: Implementation of Commit Records

```
possibility = monitor is create, create_ref, test, complete, abort, no_ref;

commit_record = record[UID: bignum, state: commit_state,
                       timeout: time, tokenset: set[versionref]]

possibility = bignum

%initialize local state
        commitset: set[commit_record]':= set[commit_record]$create()
% We write <operation>(commitset, possi) as a shorthand for set operations

% Message dispatch loop.
% A message dispatch terminates on a continue statement or when control passes
% to the end of its body.
while true do
        case awaitmessage of

create(abort_time: time) =>
        possi: possibility := uniqueID()
        insert(commitset, commit_record$(UID: possi,
                state: waiting, timeout: abort_time, tokenset: set[versionref]$create()))

        sendreply ok(possi, true)

create_ref(possi: possibility, vr: versionref) =>
        cr: commit_record := retrieve(commitset,possi)
                except when not_found:
                 sendreply permit_create(possi,vr,false); continue  end
        if cr.state=waiting then
                if cr.timeout > time() then
                        insert(cr.tokenset,vr)
                        sendreply permit_create(possi,vr,true)
                        continue
                else sendreply permit_create(possi,vr,false)
                        abort(possi); continue
                end
        end
        sendreply permit_create(possi,vr,false)

test(possi: possibility, vr: versionref) =>
        cr: commit_record := retrieve(commit_set,possi)
                except when not_found:
                 sendreply state(possi, vr, aborted); continue  end
        if cr.state ~= waiting then sendreply state(possi, vr, cr.state)
        elseif cr.timeout <= time() then
```

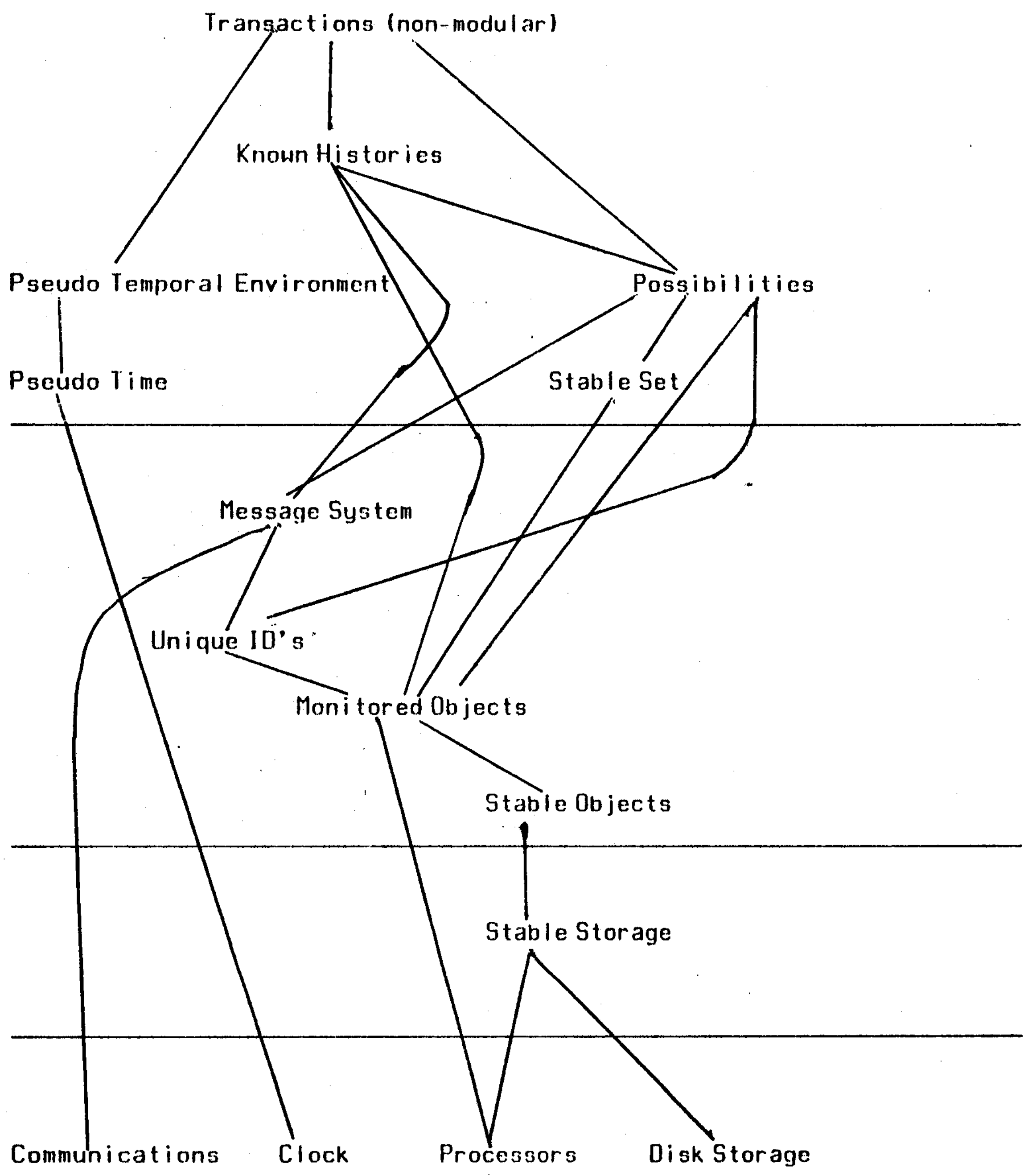
```
                sendreply state(possi,vr,aborted)
                abort(possi); continue
        % else don't reply at all
        end

complete(possi: possibility) =>
        cr: commit_record := retrieve(commit_set,possi)
                except when not_found:
                        sendreply status(false); continue end
        if cr.state ~= waiting then sendreply status(cr.state=completed)
        elseif cr.timeout <= time() then
                sendreply status(false)
                abort(possi); continue
        else cr.state := completed %atomic operation
                sendreply status(true)
                % encache state
                for vr: versionref in set[versionref]$elements(cr.tokenset) do
                        send state(possi,vr,completed)
                end
        end

abort(possi: possibility) =>
        % analogous to complete except commit record may be removed from
        % commit_set without waiting for its state to be encached in all
        % dependent tokens
        end abort

no_ref(possi: possibility, vr: versionref) =>
        cr: commit_record := retrieve(commit_set,possi)
                except when not_found: exit %i.e. do nothing
        delete(cr.tokenset, vr)
        if empty(cr.tokenset) then delete(commit_set, possi)
        end no_ref
end possibility
```

Appendix III: A Lattice of Abstractions for Reed's System



REFERENCES

[1] Lampson, B., and Sturgis, H. Crash recovery in a distributed data storage system. Working paper, Xerox PARC, April 27, 1979.

[2] Reed, D.P. Naming and synchronization in a decentralized computer system. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1978.

REPLICATION ADDED TO THE HIERARCHICAL PROCESS MODEL

Jeannette M. Wing

6.845 Atomicity, Recovery, and Coordination

May 1979

1. Motivation and Review

Replication of databases both within one site and at several sites in a distributed system provides enhanced reliability in Montgomery's hierarchical process model [Montgomery]. I will discuss how upon adding replication, the model works under normal conditions and under certain failure modes. In particular, replication requires additional complexity in synchronizing updates and handling network partitioning.

Two kinds of databases exist in the process model: the application databases that data managers access and update, and the process databases[1] which store the process state and message queues of a process. Maintaining duplicate application databases offers the advantages of increased reliability, increased accessibility, more responsive data access, and load sharing. And, maintaining duplicate process databases leads to a more robust implementation of the concurrency control mechanism.

Montgomery introduces the hierarchical process model to implement atomic broadcasting. All processes in the system are organized in a hierarchy. We find at the root and nodes, message forwarders that relay messages to the appropriate destination ports, and data managers that hold the data items and can also act as message forwarders. We find at the leaves of the hierarchy, transaction processes and data managers.

In section 2, I present two ways of adding redundancy to Montgomery's process model. In section 3, I discuss various failure modes for both models. In section 4, I summarize how replication fits into Montgomery's scheme.

---

1. Montgomery introduces *process databases* in discussing robust sequenced processes. He replicates the local state of a process in implementing simple processes so that a process step is an atomic update to all copies of the process state.

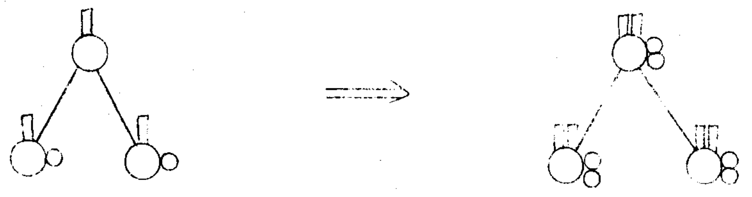
2. Replication

To achieve a fully redundant hierarchy we can replicate the nodes of the hierarchy. Or, we might wish to only achieve partial redundancy. We discuss two methods of replicating nodes for a fully redundant hierarchy and briefly discuss the advantages of achieving partial redundancy.

2.1 The fully redundant case: replicating only the databases

In this method, we expand the hierarchy to have at each node a single process maintaining multiple copies of its process and ,if a data manager, application databases. Figure 1 illustrates this where a rectangle represents a process database and a small circle, an application database.

Fig. 1.



A node with only a process database represents a message forwarder. If the process is a data manager then not only would the process state and the message queues be replicated but also the application databases. By maintaining duplicate copies of the process database we are, in essence, simulating atomic stable storage. This is consistent with Montgomery's suggestion of the need to store the process in atomic stable storage.

Each process now must provide more functions. A message forwarder no longer simply relays incoming messages to the appropriate destination ports but it must also maintain its copies of the process database. As in many multiple copy protocols [Alsberg, Grapa, Kaneko, Stonebraker], the process serves as a primary and coordinates updates to its databases. So in adding to the process the responsibilities of managing the synchronization of messages among the multiple message queues and of managing the consistency of the database copies, we add complexity to the process' set of process step specifications.

2.2 The fully redundant case: replicating the entire node

In this second method of expanding the hierarchy, we replicate each node in its entirety. A natural and consistent way of organizing these replicated nodes would be in a hierarchy (Figure 2). Note that now the structure of the hierarchy changes as a function of the branching factor at each node in the original hierarchy and the number of duplicate copies of a process we make.

Fig. 2.

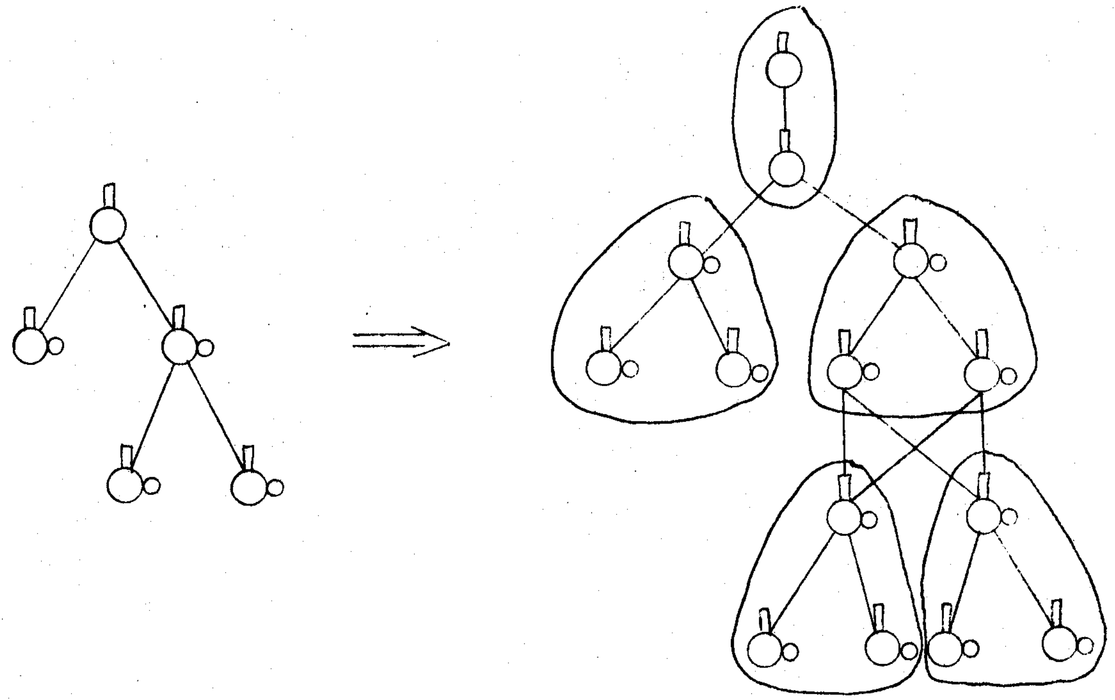


Figure 2 shows that the root node only replicated once so that the branching factor of two stays the same. We replicate each of the root's children, however, twice, so that its branching factor, in effect, doubles and the tree structure becomes a directed acyclic graph (dag). In this case, we violate Montgomery's process model since a process may no longer have a unique parent. But the effect is the same since the parents are essentially identical copies of each other. What difference this makes depends on the network topology. Some processes will physically reside at the same site so the change in the hierarchical process structure may not require a change in the physical layout of the network.

We could alternatively maintain the tree structure of the original hierarchy. We could have multiple nodes of the same replicated processes. That is, two processes may be parents of two distinct nodes where each of these nodes contains replications of the same process. But this introduces even more complexity to the update problem since copies of one process are not grouped together and the number of nodes would grow exponentially. Also, we could organize the replicated nodes in a linear chain and still maintain the original hierarchy. But this requires sending request messages (via forwarding) through all these copies besides sending the coordination messages (for maintaining consistent copies). Burdening all copies with extra message forwarding is unnecessary and we could not exploit the advantages of, say, a majority consensus algorithm. I will not pursue these alternate models.

The root of each sub-hierarchy (circled in blue in Figure 2) acts as the primary in coordinating updates. The hierarchical structure lends itself to using a protocol dependent on a primary plus backups (master/slave protocol) using distributed control protocols (timestamps and majority consensus) [Johnson, Thomas].

The advantages of this second model over the first are that:

1. The second is resilient to a single process failure whereas the first is not.

2. We add less functional complexity to the process' set of step specifications in the second model that in the first since control is distributed among the replicated processes and not centralized at one.

In the second method a message forwarder that is a primary receives a message and informs its copies (backups) that a message was received. Under normal conditions, the processes should all agree (or the primary can tally votes) on the content of the output and the destination ports. The primary tells one of the backups to forward the message. If this backup is only an ancestor and not a parent of the process that is to receive the forwarded message, this backup then fowards the message the message to one of its child, presumably another backup. This continues until we reach a backup that is a parent of the receiving process. The primary could tell all of its copies to forward the message in which case the destination ports would have to ignore duplicate messages. (This is unwise since we do not take advantage of the hierarchical structure of each sub-hierarchy. We might as well have organized the replicated processes in a linear chain.) So optimally only one, and at worst, log$n$ backups should forward the message to the destination ports where n is the number of copies of the process in the sub-hierarchy. We achieve communication between the primary (root node of a sub-hierarchy) and the backups as in the non-replicated process model by using input and output message queues. We use a second sequence numbering scheme to distinguish messages among copies from messages among distinct processes.

Note that a data manager can respond to queries immediately. Only for updates do additional messages have to be sent among the primary and backups.

2.3 The partially redundant case

In practice we might choose to replicate only the top few levels of a hierarchy. This not only requires fewer redundant copies of databases but also reflects that much traffic may occur at the higher levels because of the message forwarding mechanism. Because of Montgomery's assumption of locality of reference the benefits of reliability and accessibility due to replication are amplified if each site in the physical network maintains copies of the higher levels of the hierarchy. Besides the local traffic we expect many messages to be forwared upwards in the hierarchy from sending processes which are not directly related to the receiving processes but related only through an ancestor. If coordination of message forwarding among replicated nodes contributes only a negligible amount to overall processing then we increase the reliability of message forwarding and the accessibility (for queries) of data via data managers.

3. Failure Detection and Recovery

I will describe the effect of replication on process failure and network partitioning. Network partitioning raises severe problems when databases are replicated in comparison to those problems raised when several sites maintain single dedicated databases per site or when a database is distributed among several sites. I further discuss this in section 3.2.

Recall that Montgomery limits the effect of lost, duplicate, and damaged messages by using robust sequencing, atomic stable storage, replication of local states, and error detecting codes.

3.1 Process failure

If a process fails in the first model suggested for replication then it is treated like a process that has removed itself from the hierarchy. A parent message forwarder of this dead process must be able to detect it, reclaim its message queues, and inform the ancestors of the message forwarder of the dead process. Then the parent can choose to adopt the dead process' children. At this point any application databases at the dead process are inaccessible.

If a process fails in the second model suggested for replication then we alter the structure of the hierarchy. First we treat the dead process as above. If the dead process is the root, then we elect a copy to be the new root of the sub-hierarchy (that is, the hierarchy of a process and its copies). This can change a dag to a tree. In Figure 2, if one of three copies (say of the leftmost sub-hierarchy) dies, one of the remaining two is or becomes the parent of the other. When a process p becomes live and wishes to rejoin the structure the process needs only to send a "request for adoption" message to some message forwarder f. That message forwarder f checks to see if other copies of the process p exist and if so, f sends the current root of that sub-hierarchy a message to adopt p. If no copies of p exist then f can add it to the hierarchy. We delete and add these processes using the same protocol as for point-to-point communication.

The obvious advantage of the second model is that we can still access application databases via live copies of data manager processes. In this sense we are resilient up to a certain n number of process failures where n is the smallest number of copies of any one process in the entire hierarchy. If the copies are physically separate we might expect to be resilient beyond n process failures if not all of the first n failures are copies of the same process.

3.2 Communication failures

Network partitioning poses problems that are particular to replicated databases. Basically, partitions must journalize requests and somehow merge these journals when they rejoin. In a distributed system where there is no replication but where each site hosts dedicated databases, when the network partitions, those sites that can communicate with each other can access each other's databases. In a system where there is sharing or where a database is split among many sites, when the network partitions, users can query and update that portion of the database within the same partition. We do not need to journalize requests because no inconsistencies can arise. Upon rejoining partitions in any of these cases we merely increase accessibility of these databases to the rest of the network.

If the physical communication network is hierarchical or is amenable to a hierarchical structure then a corresponding process model is homomorphic to the actual network topology. Knowing the topology helps in determining what actions to take when the network partitions. The following discussion assumes a hierarchical network topology.

If a communication failure occurs between two processes in the first model suggested for replication then the network divides into two hierarchies. Both partitions can continue to service a request for which there exists a message forwarder within the partition that is the ancestor of all the destination ports of the request message. This type of communication failure does not affect replication of the process and the application database in this model as in the second model.

The second model poses trickier problems because we no longer manipulate a strict hierarchy (tree) but a dag. Thus, a communication failure between two distinct processes or between two copies of the same process does not necessarily imply we have network partitioning.

First I will consider the complete isolation of one part of a hierarchy from another. This may require more than one communication link failure.

> 1. If the root of the isolated part is acting as the primary of a sub-hierarchy then we treat this case as for the first model. (Each partition services what it can within its partition.)
>
> 2. If the root is not, then it is one of the copies of a process in the other partition of the hierarchy. In this case we have the classic network partitioning problem of journalizing requests and merging journals upon rejoining partitions.

Next I will consider just a single communication failure. There are two cases to consider:

> 1. Failure between two distinct processes. There are two subcases.
>
> > a. The failure occurs between a process and its *unique* parent [Figure 3]. In this case we treat the failure as described for the first model because the network partitions into two hierarchies.

b. The failure occurs between a process and *one* of its parents [Figure 4]. In this case the child process *p* can still be reached by one of its other live parents. When the link is connected again the dead parent will see the current version of the child process *p*.

Fig. 3.

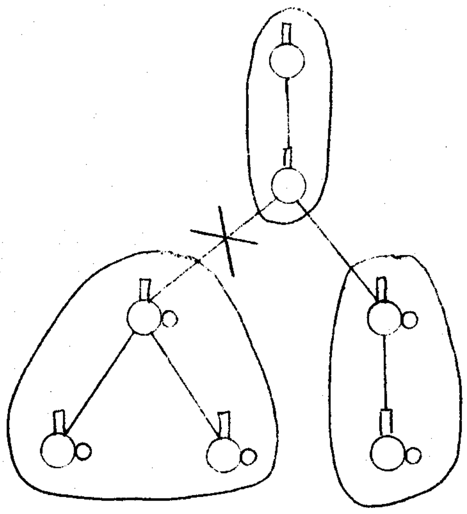
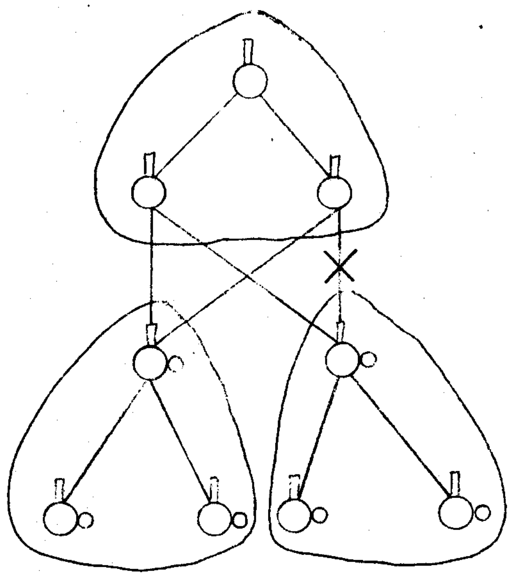


Fig. 4.

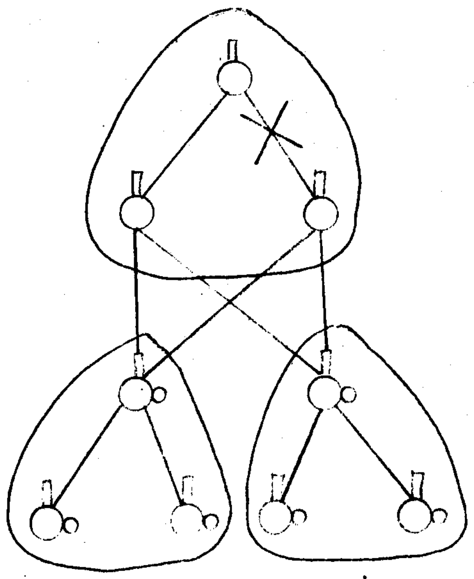


2. Failure between two copies of the same processes. Again, two cases.

a. The failure occurs such that the network partitions into two hierarchies. We treat this case as in (a) above.

b. The failure occurs such that the network does not partition [Figure 5]. This case depicts a side effect that arises in dags. All processes are accessible but the root process shown in Figure 5 might not be aware of any changes that its right child *r* may make initiated by either of *r*'s children. How innocuous this is depends on the application.

Fig. 5.



Network partitioning and single communication link failures raise problems when we add replication to Montgomery's process model mainly because of the change in the structure of the process model graph.

4. Summary and Conclusions

I described two ways of adding replication to Montgomery's process model. For one, we replicate locally the process and the application databases; for the other, we replicate the entire process node which may change a strict hierarchy (tree) into a dag.

For each model I discussed what happens when a process fails, when the network partitions, and when single communication links fail.

I favor the second model of replicating entire process nodes because it provides greater reliability, that is, resiliency to a single process failure, than the first model. It does, however, introduce problems with communication link failure that do not appear in the first model.

By adding replication to Montgomery's scheme, we add reliability and improve the robustness of his concurrency control mechanism at the expense of additional message passing for maintaining consistency among the copies.

References

[Alsberg] Alsberg, P.A., and J.D. Day, "A Principle for Resilient Sharing of Distributed Resources," Second National Conference on Software Engineering, 76CH1125-4C, pp.562-570.

[Grapa] Grapa,E., "Characterizations of a Distributed Data Base System," Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois-R-76-831, October 1976.

[Johnson] Johnson, P.R. and R.H. Thomas, "The Maintenance of Duplicate Databases," ARPANET NWG/RFC #677, January 1975.

[Kaneko] Kaneko, A., et al, "Logical Clock Synchronization Method for Duplicated Database Control," Nippon Electric Company Central Research Laboratories, LR-3854, September 1978.

[Montgomery] Montgomery, W.A., "Robust Concurrency Control for a Distributed Information System," MIT/LCS/TR-207, December 1978.

[Stonebraker] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," Proc. Third Berkeley Workshop, August 1978.

[Thomas] Thomas, R.H.., "A Solution to the Update Problem for Multiple Copy Data Bases Which Use Distributed Control," BBN Report #3340, July 1976.