Implementing Atomic Actions on Decentralized Data

DAVID P. REED
Massachusetts Institute of Technology

Synchronization of accesses to shared data and recovering the state of such data in the case of failures are really two aspects of the same problem—implementing atomic actions on a related set of data items. In this paper a mechanism that solves both problems simultaneously in a way that is compatible with requirements of decentralized systems is described. In particular, the correct construction and execution of a new atomic action can be accomplished without knowledge of all other atomic actions in the system that might execute concurrently. Further, the mechanisms degrade gracefully if parts of the system fail: only those atomic actions that require resources in failed parts of the system are prevented from executing, and there is no single coordinator that can fail and bring down the whole system.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; H.2.2 [**Database Management**]: Physical Design—*deadlock avoidance, recovery and restart*; H.2.4 [**Database Management**]: Systems—*distributed systems, transaction processing*; H.2.7 [**Database Management**]: Database Administration—*logging and recovery*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: time-domain addressing, two-phase commit, stable storage, nested atomic actions

1. INTRODUCTION

The research reported here was begun with the intention of discovering methods for combining programmed actions on data at multiple decentralized computers into coherent actions forming a part of a distributed application program. The primary concerns were that it be easy to coordinate such combined actions with other concurrent actions accessing the same data, and that it be easy to handle failures in any part of the combined action.

This research was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

Author's address: Laboratory for Computer Science, Massachussetts Institute of Technology, Cambridge, MA 02139.

This paper was originally accepted for publication in *Communications of the ACM.* The CACM department editor responsible for the paper was Anita K. Jones. The author kindly agreed to publish the paper in *ACM Transactions on Computer Systems.*

In the course of the research it became clear that coordinating access to data and recovery from failures were complementary mechanisms aimed at achieving the same goal: providing data and program modules whose behavior is easily specified without consideration for the details of the choice of data representation or the sequence of primitive steps executed that achieve the behavior. This goal is the familiar "information-hiding principle" elucidated by Parnas [17]. Atomic actions make the construction of such modules straightforward: by implementing operations of a module as atomic actions, concurrency and failure can be ignored.

We describe a new method for synchronization and failure recovery that can be easily implemented in a decentralized system. We concentrate here particularly on the application of this method to the implementation of atomic actions. More general applications are described in the author's doctoral dissertation [19].

In the discussion that follows, we first define atomic actions and discuss the problems of implementing atomic actions in a decentralized system. We then describe a way of thinking about objects as sequences of unchangeable versions, *object histories*, that reflect the sequence of changes made to the object over time. Updating an object is thought of as creating a new version, while reading an object is thought of as selecting the proper version and obtaining its value. As part of this discussion, we describe two complementary techniques for coordinating the versions of multiple objects—the *possibility*, which is a group of tentative versions created by updates that can be "simultaneously" added to the object histories, and *pseudotime*, which is a timelike ordering of events used to select the versions of objects that are read and created by a particular program. An example illustrating how the techniques work then follows. Finally, we compare these methods with traditional synchronization and recovery mechanisms.

2. ATOMIC ACTIONS

We define an *atomic action* as a program-specified computation that, although composed of primitive computational steps executed at different times and in different places, cannot be decomposed from the point of view of computations outside the atomic action. During the execution of atomic actions, intermediate states of data objects that arise will never be observed by computations outside the atomic action. During the execution of an atomic action, objects whose values are read by steps of the atomic action can be modified only by other steps belonging to the atomic action during the execution of the atomic action.

Concurrency and failure both threaten to decompose atomic actions. Two concurrent programs accessing a shared variable can interact in such a way that intermediate states of data within one program are observed by the other. If one program modifies the variable while the other is executing, the behavior of the first can be affected. Similarly, a program that halts because of a failure will leave intermediate states of data objects exposed. To say that a program specifies an atomic action means that the program must be atomic even in the face of concurrent execution and failures. We refine our definition by the following two requirements on the set of computational steps in an atomic action A_p specified by program P.

(1) *Concurrency atomicity.* For all executed steps o not in A_p, either o precedes all steps in A_p or o follows all steps in A_p.[1]
(2) *Failure atomicity.* Either all steps in A_p complete, or none of them complete.

To illustrate these concepts, we use a simple example. Consider a bank database, where each account is represented by an object whose value is the balance of the account (this is an extremely simplistic banking system, of course). To transfer money between two accounts, one account's balance must be debited and the other credited. If complex banking transactions are allowed to proceed concurrently, undesirable behavior may occur. For example, two transactions trying to debit the same account may operate incorrectly if both read the value before either writes back its debited value. If an audit operation reads the balances between the debit and credit for some transaction, some money may appear to be missing. Failure can have equally disastrous results, leaving accounts in an inconsistent state.

More abstractly, structuring a system using atomic actions is a key way to assure consistency within the system. Consistency is usually proved inductively: the system is initially consistent (basis) and each operation is shown to take a consistent state into a new consistent state (induction step). By making the operations atomic actions, we can ignore concurrency and failures in the proof, because the equivalence to a total ordering is assured. We can ignore failures because each operation is executed wholly or not at all. Consistency is just one example of a property proved by induction on sequences of atomic actions; we can also ignore concurrency and failure in proving other inductively proved properties if the operations are atomic actions.

Thus, built-in atomic actions simplify the programmer's task in coping with unplanned concurrency and failure. The programmer of an atomic action writes his program without concern that other computations will interfere with data it touches. The specification of an atomic action need describe only the ultimate effect on shared data in terms of the data's initial state and any other inputs to the action, without need to describe any intermediate states assumed by the data, since such intermediate states are not visible to computations outside the atomic action. Thus the implementation of the atomic action can be significantly modified (e.g., doing the credit before the debit, or doing both simultaneously) without propagation of changes to actions outside the atomic action.

Our concept of atomic actions is quite similar to that of Lomet [16] and also to the *sphere of control* described by Davies [5, 6]. If all computations in the system perform all their data accesses as part of atomic actions, then the observable behavior of the system will be the same as a serial schedule, as in the definition of atomic transaction developed by Eswaran et al. [10].

Recovery blocks [18] are also closely related to atomic actions, but recovery blocks have no explicit provision for synchronization. The crucial insight we have

[1] If all steps o are part of some atomic action, then concurrency atomicity is the same as serializability [11]. But we allow here for actions that are not concurrency atomic. An important example is the backup or undo action, which restores the state of a set of objects to that which existed at some point in the past. Such an action should be failure, but not concurrency, atomic.

made is that the atomic action abstraction arises from consideration of the programmer's need to design a program without concerning himself with interactions from concurrent computations. This leads to the idea of hiding intermediate states within the atomic action, and thus implies both the required synchronization and recovery.

3. THE DECENTRALIZED SYSTEM

It is fairly easy to understand the atomic action abstraction and its implications—the trick is to provide the mechanics that allow atomic actions to be implemented. We are particularly concerned with the mechanics of implementing atomic actions in a decentralized distributed computer system. By decentralized computer systems we mean a set of computer nodes each consisting of processor, memory, and permanent storage (disk), connected together by a communications network. Each node can be used as an autonomous computer. The network provides sharing of information between these nodes. It is this sharing of information among programs executing on distributed nodes that must be coordinated.

Communication among nodes is by message passing. The arrival of messages at nodes causes execution of programs that may result in modifying data at that node or retrieving data from that node. In what follows data are modeled as recordlike objects that may contain references to other data objects either on the same site or on other sites. The algorithms that manipulate such objects are assumed to be implemented by programs existing at multiple sites that communicate by message passing.

We assume that individual nodes are capable of providing *atomic stable storage*, that is, storage blocks that do not lose their contents as a result of failure, and for which there are actions that read and write entire storage blocks atomically. By atomic, we mean that a write operation will either successfully write the specified data value or else fail in such a way that the previous data value is left unchanged. No other outcome (e.g., half the data is changed) should be possible. Lampson and Sturgis have argued that it is practical to provide such storage and have suggested a technique for implementing it in [15]. For simplicity of discussion, atomic stable storage is assumed to be the basis for all data structures described here. However, most of the data structures need only be in stable storage, that is, storage that does not lose its contents, but for which failures during a write may leave data incorrect.

The failures we consider all take the form of failing to carry out a requested action. For example, a message may be lost, a processor may stop in the middle of a program, or a node may reject a request because it violates protection constraints. Many failures can be transformed into this class—for example, garbled data messages may be transformed into lost messages by using error detecting codes. We do not consider failures where one action is attempted and results in a quite different, yet valid, action occurring instead.

Six objectives related to the distributed system shaped our solution to the problem of implementing atomic actions. These are as follows:

(1) *Maximize node autonomy, while allowing multisite atomic actions.* The maintenance of individual objects should be the responsibility of the node

containing the object. Consequently, the implementation of synchronization and recovery should be decentralized. However, nodes must work in concert to support the implementation of atomic actions involving objects on multiple nodes (e.g., the case where the source and destination bank accounts are kept on distinct computers).

(2) *Modular composability of atomic actions.* It should be possible to combine separately written programs that manipulate shared data objects into one atomic action, without the need to plan all such combinations in advance. Many schemes for synchronizing access to data fail to satisfy this goal, because mutual exclusion mechanisms, such as critical sections [9], synchronizers [2], and monitors [12], require advance knowledge of all potentially conflicting users of shared data to design the proper synchronization. Such knowledge may not always be available, unless the data accessed by all programs in the distributed system can be known at the time a new atomic action is created—a condition that may be difficult to achieve in practice. For example, in a system where autonomous users can implement new atomic actions, the names of objects accessed by each atomic action would have to be stored in some central registry.

(3) *Support for data-dependent access patterns.* There are important cases where the set of objects to be accessed by an atomic action cannot be predicted in advance. In the bank example such a requirement might arise if one could associate a "reserve account" with one's normal account. If the balance of the normal account is too low to satisfy a debit, the reserve account is debited instead. Then one cannot know what accounts will be accessed without accessing them. Another example occurs if objects may contain pointers to other objects.

(4) *Minimize additional communication.* By doing the necessary synchronization at the time and place where shared objects are stored, one can avoid the need for finding and synchronizing with remote conflicting actions. In a system with shared memory, coordination can be achieved inexpensively by locking objects to be accessed before use, and unlocking after the last use. Such locking is inexpensive because all processes can easily access the locks and because deadlock detection or avoidance can be centralized. In a distributed system, locking requires interactions among nodes that use the objects and therefore substantial communications delays. We thus would like a solution that uses a minimum amount of internode message passing. It is preferable that coordination of processes accessing the same object be done as part of the access operations themselves, and locally to the node containing the object for reasons of performance and autonomy. The best arrangement would be that only the message passing needed to request the actual reads and writes of objects occur. Our scheme achieves this minimum.

(5) *No critical nodes.* A critical node is one whose failure prevents the entire system from being used. Certainly when a node fails, the data it holds become inaccessible. However, it should be possible for computations that do not use data on the failed nodes to proceed.

(6) *Unilateral aborting of remote requests.* Recovery from failures is made difficult in a distributed system by the peculiar nature of communication failures. In particular, when a requestor requires a service from a server that involves modifying data objects stored at the server, certain kinds of communication

failures will leave the requestor in doubt as to whether the server has performed the requested action or not. Further actions by the requestor usually require successful completion of the request at the server to ensure internode consistency. The requestor must wait until the server state can be ascertained, but this can take a very long time. If the requestor holds resources needed by other computations, then such a failure can cause deadlock. We thus would like a mechanism that allows for unilaterally aborting an atomic action, without the need to communicate with all the nodes involved in an atomic action.

4. IMPLEMENTING ATOMIC ACTIONS

In order to understand our approach to synchronization, it is helpful to think about the following noncomputer example. Suppose the personnel officer of a company is charged with determining raises for all employees, by reviewing and updating their personnel folders accordingly. The personnel officer must also submit the total change to higher management for approval. The way he would proceed would be to choose the date at which the raise is to be effective, then process folders one at a time, perhaps going back to alter an earlier choice he had made. The updated salary would not replace the existing salary; instead it would be placed in the folder as a tentative increase, effective as of a certain date, pending approval by the manager. Queries about the salary by the employee or a credit bureau would still show the old salary if processed before the effective date, but if a query were to arrive after the effective date, it would be necessary to check whether the increase was approved. Once the manager received the report and authorized the increase, the salary change would become effective. Thereafter, to clean up the folders, the personnel officer might go to each folder and stamp the new salary "approved," but this only makes answering queries quicker by eliminating the check of whether the increase was approved.

We now elaborate this strategy in terms useful for the computer. To do so, we need to develop three new concepts: (i) use of a timelike ordering, *pseudotime*, to define the interaction between operations on an object (effective dates); (ii) representation of each object as an *object history* of *versions* (the sequence of salaries); and (iii) grouping of tentative versions into sets called *possibilities* that facilitate backward error recovery (the new salaries before authorization). To develop each concept, we describe the properties of each and then a plausible implementation.

We then specify READ and WRITE operations on objects (procedures for accessing folders). Unlike the usual READ and WRITE operations, the behavior of our READ and WRITE is defined in terms of pseudotime, possibilities, and object histories. Once we have defined READ and WRITE, we use them to implement atomic actions.

4.1 Object Versions

We think of each object as a sequence of *versions*.[2] Each WRITE to an object creates a new version, installing it into the sequence of versions, which represents

[2] Our use of object versions is similar to the treatment of synchronization in [21], though our mechanism using known histories and pseudotime is quite different. Also closely related is the practice of using version numbering for modifications to files, as in TENEX [4], though such operating systems provide no mechanism for interfile consistency.

the history of the object as known so far—the *object history.* Once created, a version's value does not change. The pseudotime of the write is recorded with the version and is used to order the version properly with respect to other versions. To READ from an object at pseudotime p, p is used as described below to select the proper version of the object to return.

To simplify the following discussion, we assume that the complete sequence of versions belonging to an object history is stored. We show later how to improve the situation so that for most objects only the current version is actually stored. For now, we ask that the reader suspend disbelief on this point until we have described the basic implementation of objects.

4.2 Backward Error Recovery

Before continuing the discussion of how READS and WRITES manipulate the object history, we must describe the way failure recovery works. If a failure prevents an atomic action from being completed, any WRITE the atomic action had done to shared data should be aborted to satisfy the requirement that no intermediate states of atomic actions are visible outside the atomic action. For this reason, new object versions are written in two steps. First, each WRITE operation creates a tentative version, called a *token,* in the appropriate object history. Thereafter, if the atomic action is committed, all the tokens it has created are converted to permanent versions, but if the atomic action is aborted, all the tokens it has created are removed from their object histories.

A token can be thought of as a version whose existence is conditional upon the later completion of the atomic action that created it. We call the set of tokens created by an atomic action a *possibility,* since it represents a new state of the system whose existence is conditioned by the successful completion of the creating atomic action. At the beginning of an atomic action, a new possibility is created. Each WRITE done by the atomic action creates a new token and adds that token to the possibility. When the atomic action completes successfully, it *commits* the possibility, so that all of its tokens become versions.

Possibilities have two important properties. First, the commit step is atomic; even though an arbitrary number of tokens may belong to the possibility, and those tokens may be at many nodes, there should be no way for a failure to cause only part of the commit step to be completed, changing only some tokens into versions. Second, we wish to ensure that, if the atomic action never completes, all of the tokens will be erased. We detect that an atomic action will not complete by using a time-out, so that if the atomic action goes into a loop, for example, we can still ensure that the tokens will eventually be erased or converted into versions.

Until the time the atomic action is either committed or aborted, the final state of the tokens created by the atomic action is in doubt. While the status is in doubt, computations that belong to the atomic action may need to read the value of its tokens, but computations outside the atomic action must be prevented from reading the tokens until the atomic action is completed, lest they obtain values that will be erased by a subsequent failure of the atomic action.

As an illustration of why tokens must be visible to the atomic action that creates them, but not visible outside that action until the action is completed, consider our bank example again. Suppose that we wish to do two transfers out of the same account as one atomic action. One way to do this is to call on the

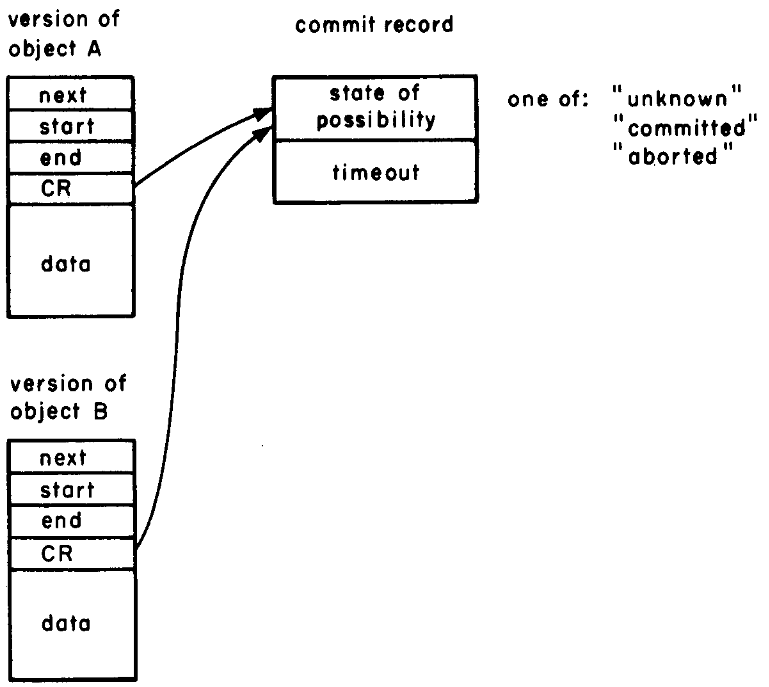

Fig. 1. Versions of several objects sharing a commit record.

transfer action described earlier twice, once to do each tranfer. The source account will then be debited twice, with the second debit action reading the reduced balance resulting from the first debit action. This reduced balance will be represented as a token in our model, visible to the second part of the double transfer action, but not visible to other computations yet, since a failure may occur before the second credit is completed, necessitating aborting the whole double transfer.

4.2.1 *Implementing Possibilities.* An implementation for possibilities works as follows. The state of a possibility is represented by a record in atomic stable storage called a *commit record,* which records the state of the possibility and the time of its time-out (see Figure 1). When initally created, a commit record is in the "unknown" state. Committing the possibility is done by changing the state atomically to the "committed" state. Aborting the possibility is accomplished by changing the state to "aborted." If the state is still "unknown" after the time-out elapses, the next time the state is examined, that activity will change the commit record's state to "aborted."

Every WRITE or READ operation uses a parameter (see Section 4.4) that is an identification for the commit record for the atomic action containing that WRITE or READ. Each version (whether token or not) contains a commit record pointer, stored by the WRITE operation that created it. When a program wants to read a version, this commit record is first compared with the commit record associated with the READ. If the same commit record is used for the version and the READ, then the READ is part of the same atomic action as the WRITE that created the version; so it does not matter if the commit record is still in the "unknown" state. If different commit records are used, then the READ operation checks the state of the version's commit record. If the version's commit record is "committed," then the READ operation returns the version's value. If the commit record is found to be "aborted," then the READ operation erases the version and another version is selected (see Section 4.4 for discussion of how READs select the version read). Finally, if the commit record is in the state "unknown," the READ operation waits; this wait is bounded by the time-out on the commit record.

4.3 Pseudotime

We usually describe the effect of reads and writes to an object in terms of the *time ordering* of the reads and writes. Thus, a READ returns a value determined by the latest WRITE that precedes the READ. In order to achieve synchronization between actions that manipulate shared data, we must then provide means for controlling the real-time ordering of READs and WRITEs. In a decentralized system, because of communications failures and delays, correctly enforcing relative time orderings between actions is difficult and slow.

In our approach to synchronization, we have replaced the real-time ordering with ordering by a timelike, totally ordered set called *pseudotime.* Each READ and WRITE operation is assigned a particular pseudotime, and READs and WRITEs to shared objects behave such that a READ returns the value written by the latest (in the pseudotime ordering) WRITE that precedes (again, in pseudotime) the READ. The thing that makes pseudotime ordering easier to deal with than real-time ordering is that we are able to devise a decentralized way of reserving ranges of pseudotime values that does not require communication among the participants.

Pseudotime must also have two other properties. First, if two steps of a computation are ordered such that step A must occur before step B (e.g., if B waits for A to finish), then the pseudotime for step A must precede (in the pseudotime ordering) that of step B. This property ensures that sequential programs still behave in the same way as they do in a system where real-time orderings define the meaning of reads and writes to shared data.

The second property loosely links the rate of increase of pseudotime to real time. Basically, we would like the pseudotime ordering of two events to correspond to their real-time ordering whenever the events occur far enough apart in real time. Thus we do not care what pseudotime ordering is assigned to nearly simultaneous events that are not ordered parts of the same computation. We do care, though, that two events that are observed to be ordered from outside the system be ordered in pseudotime in the same way.

The pseudotime of an event is generated from a *pseudotemporal environment.* Each computation has, as part of its execution state, a pseudotemporal environment (PTE) which generates the pseudotimes used by steps of that computation in reading and writing shared data. We describe only the PTEs of atomic actions here.

To guarantee that other atomic actions do not see intermediate states of objects modified by other atomic actions, we require that for any two atomic actions, A and B, either all steps of A must precede (in pseudotime) all steps of

B, or all steps of *B* must precede all steps of *A*. We achieve this by constraining the sets of pseudotimes that are used in each atomic action so that either all pseudotimes used in *A* precede all those used in *B*, or all those used in *B* precede all those used in *A*.

There are two important operations on pseudotemporal environments and pseudotimes. To create a new pseudotemporal environment for an atomic action, one executes the following:

$$\text{pte} \leftarrow \text{CreateAtomicPTE}(\,).$$

The resulting object is a generator of pseudotimes. To generate the next pseudotime from a PTE, one executes the following:

$$\text{pt} \leftarrow \text{NewPT}(\text{pte}).$$

The result is a pseudotime greater than all pseudotimes previously generated using PTE.

4.3.1. *Implementing Pseudotime.* We briefly describe an implementation of pseudotime and PTEs that is suitable for a distributed system. The implementation uses approximately synchronized real-time clocks at each node of a distributed system. Clocks can be synchronized easily to within microseconds using the WWV radio time standard broadcast by the National Bureau of Standards. Lamport's clock synchronization mechanism would also suffice [14]. These clocks are used to create time stamps that are unique. To the value read from the clock, a unique site identifier is concatenated as the low-order bits. Thus, even though two sites need not communicate, it is guaranteed that the sets of time stamps they generate are disjoint.

A pseudotemporal environment for an atomic action is represented as a two-component structure consisting of a time stamp obtained at its creation, and a time stamp read as part of the last selection of a pseudotime from the PTE. For simplicity, assume that each of these quantities is specified as an *N*-bit integer.

A new pseudotime is selected from the pseudotemporal environment by getting a time stamp and prefixing it with the time stamp of creation of the PTE (see Figure 2). The time stamp read must be greater than the time stamp of all prior selections; if not, either the new selection must wait, or the real-time clock must be set forward. The real-time clock value read also replaces the time stamp last selected in the PTE.

Comparison of pseudotimes is done by treating them as binary fractions, where the leftmost digit is the high-order bit of the creation time stamp of the source PTE. As a result of this definition, the pseudotimes in one PTE always are less than any pseudotimes selected from a PTE created later. If two PTEs are created "simultaneously" at different sites, pseudotimes from each will be ordered by the order induced by the site identifiers that make time stamps unique in their low-order digits.

Thus far, we have discussed the PTEs and pseudotimes associated with atomic actions. Data accesses made by programs outside of atomic actions nonetheless use pseudotimes. Such pseudotimes are derived from a simpler kind of PTE that consists only of a cell containing the time of last selection. A new pseudotime is selected by just reading a time stamp and storing it in the PTE. Treating these

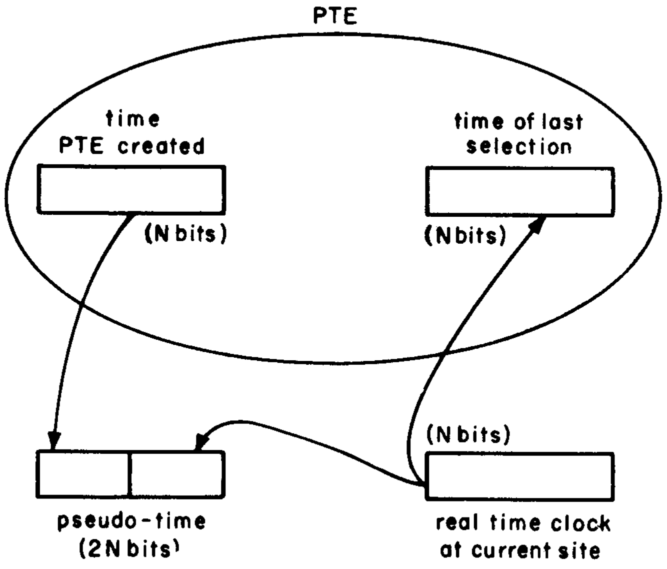

Fig. 2. Selecting a new pseudotime from pseudotemporal environment for an atomic action.

N-bit time stamps as binary fractions orders them correctly with respect to the (longer) time stamps belonging to atomic actions.

The use of time stamps for synchronization was originally developed by Johnson and Thomas [13]. Later work by Thomas [23] and Bernstein et al. [3] has carried this approach further. We were inspired by these approaches, though they did not use the time stamps in the way we have described.

4.4 Coordinating Object Reading and Writing

We now define the READ and WRITE operations in terms of the above concepts. The READ operation selects the proper version of the object, ensures that it is readable, by the requesting computation, and returns the value. The WRITE operation creates a token in the object history. Additional parameters to the READ and WRITE operations specify the pseudotime and possibility to be used in each operation. In a practical system, these additional parameters would be provided implicitly by the system; we give them explicitly to simplify our notation. Thus, we have the operations

$$v \leftarrow \mathrm{READ}(o, p, q)$$

and

$$\mathrm{WRITE}(o, v, p, q)$$

where v is the value, o is the object identifier, p is the pseudotime, and q is the possibility.

Before discussing the implementation of READ and WRITE operations, it is important to emphasize that we place *no* constraint on the real-time order of processing of READ and WRITE operations on an object. Thus a READ of an object at pseudotime p may arrive at and be processed by the node containing the object before a WRITE whose pseudotime is less than p. In a distributed system, delays in transmitting messages may be unbounded as a result of network congestion or transient failures, resulting in such out-of-order arrivals. Since nodes are not ominiscient, they will process READ soon after arrival, although there may be some point in holding a READ for a small amount of time in case the WRITE arrives slightly late. If a WRITE with a pseudotime less than an already executed READ arrives at a site, it cannot be executed, for that would cause the value previously returned by the READ to be incorrect. Consequently, such a late-arriving WRITE will be rejected.

In order to keep track of the reads that have been executed, each version records the maximum pseudotime of the READs that have accessed that version. A late-arriving WRITE is detected by the fact that its pseudotime p is in the interval between the pseudotime of the WRITE creating a version and the maximum READ pseudotime of that version.

To execute the write command WRITE(o, v, p, q), the system first checks to see if there is already a version that exists at the pseudotime p. If so, then an error is signaled. Otherwise, a new token is created and sorted into the object history according to the pseudotime p. The pseudotime p and possibility q are recorded in the version.

To execute the operation $v \leftarrow$ READ(o, p, q), we must first determine which version is the proper one to access. The READ operation must use the version created by the WRITE with largest pseudotime that does not exceed p. To determine if we can return the value of the version, we first check whether the possibility recorded by the WRITE is the same as q. If so, the value of the version can be returned; if not we check to see if the version is committed, by testing the state of the possibility recorded when it was written. If the state is "unknown," then the READ operation waits until the state becomes either "aborted" or "committed." If the state is "committed," then the value of the version can be returned as the result of the READ. If the state is changed to "aborted," then the version is removed from the object history and the previous value is returned. Before the value is returned, the maximum read pseudotime is updated to p if necessary.

For illustrative purposes, we can represent object histories as a sorted linked list of versions, as in Figure 3. Each version contains a value (DATA), the pseudotime of the WRITE that created it (PTW), the maximum read pseudotime (PTR), and the identifier of the possibility that created it (a pointer to a commit record, CR). The list is sorted in order of decreasing pseudotimes. Figure 3 also shows how we WRITE(o, v, p, q) by adding a token. This token becomes a version if and when the state of q is changed to "committed," or it will be deleted if and when q's state is changed to "aborted."

5. EXECUTING AN ATOMIC ACTION

An atomic action consists of a prologue, an epilogue, and the sequence of steps that carry out the desired actions. The prologue consists of constructing a new pseudotemporal environment for the atomic action and a new commit record. The epilogue consists of changing the state of the commit record from "unknown" to "committed." If the commit record is already in the aborted state, due to a

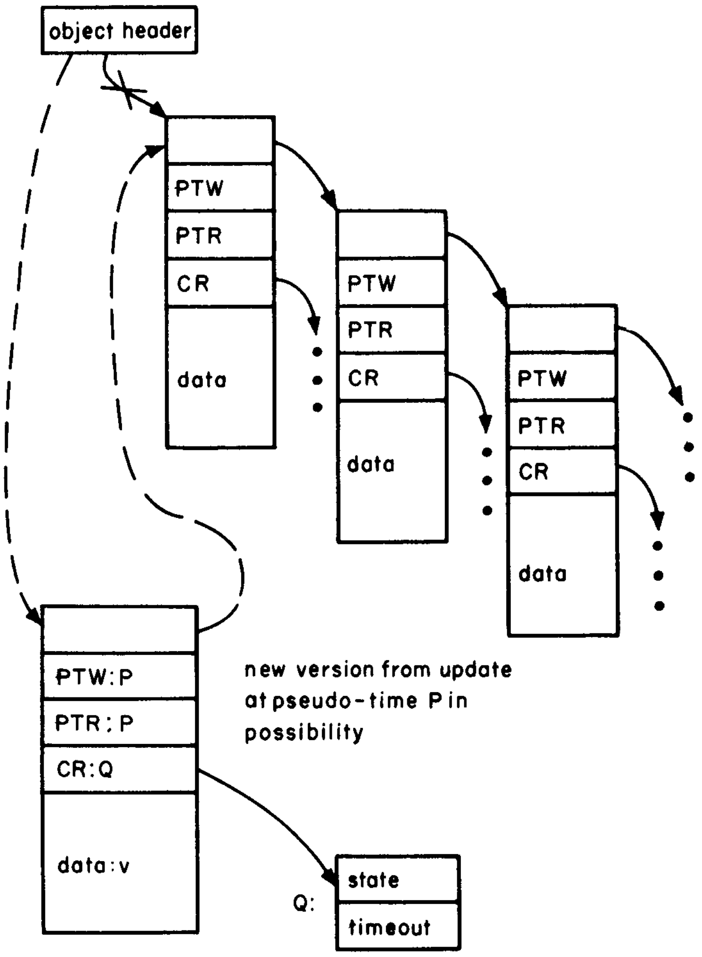

Fig. 3. Object history.

time-out or a unilateral attempt to abort the action, the commit record's state is not changed and the fact that the atomic action has failed is reflected to the invoker. The epilogue must be executed only when it is known that all steps of the atomic action have finished correctly. Thus, positive acknowledgments must be received from all remote nodes that execute parts of the atomic action.[3] If the epilogue is not executed (as when the node that would execute it crashes), the time-out on the commit record will eventually cause the tokens created by the failing atomic action to be erased.

All reads and writes executed by the atomic action are done by executing READ and WRITE operations with pseudotimes obtained from the atomic action's PTE and with the commit record constructed in the prologue. The PTE

[3] Actually, not all steps have to succeed. The atomic action may provide for this in the alternative steps.

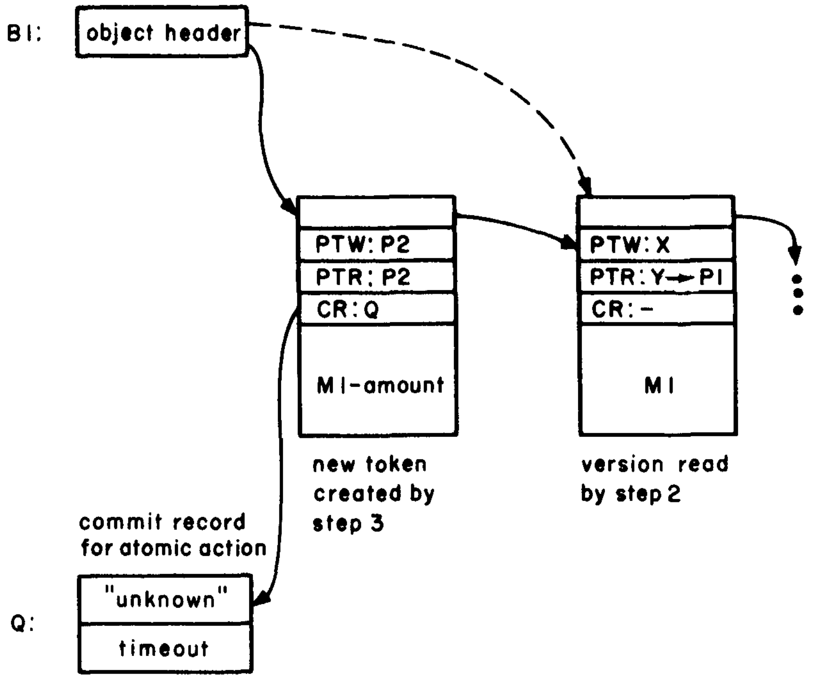

Fig. 4. Object history of *B*1 after step 3 of transfer.

provides the atomic action with exclusive access to a range of pseudotimes that will not be used by other programs. Thus a version written as part of the atomic action will be observable by other computations only if it is the final version of that object written by the atomic action; versions with pseudotimes earlier than the final one cannot be returned as a result of a READ with a pseudotime later than those in the atomic action's PTE.

5.1 Example Atomic Action

Let us consider the case of the transfer between two bank account balances, B1 and B2, in more detail. The atomic action is executed as six steps, as follows. If any of steps 2–5 fail, the whole action is aborted by not executing step 6. The time-out on the possibility will eventually erase all tokens written, but if a failure is detected, the atomic action may choose explicitly to set the possibility's state to "aborted" to speed things up.

1. Create a new possibility Q and a new pseudotemporal environment E.
2. Select pseudotime P1 from E, and READ(B1, P1, Q, balance).
3. Select P2 from E, and WRITE(B1, P2, Q, balance-amount).
4. Select P3 from E, and READ(B2, P3, Q, balance).
5. Select P4 from E, and WRITE(B2, P4, Q, balance + amount).
6. Try to change the state of Q from "unknown" to "committed."

Because of the order of the steps, we know that P1 < P2 < P3 < P4, and that all pseudotimes used in other computations are either less than P1 or greater than P4. As a result, step 2 will read a version of B1 and step 3 will create the succeeding version. Figure 4 shows the state of B1's object history after step 3.

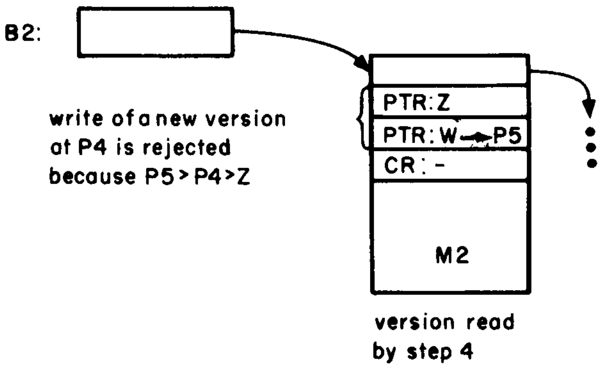
B2:
write of a new version at P4 is rejected because P5 > P4 > Z
PTR:Z
PTR:W→P5
CR: -
M2
version read by step 4

Fig. 5. Object history of *B*2 after step 4 of transfer and a READ at *P*5.

Similarly, step 4 will read a version of B2, and step 5 will create the succeeding version. Note that another program executing a read on B1 whose pseudotime is later than P2 will wait until step 6 is completed or the atomic action is aborted.

Problems may occur, however, if some other computation executes a READ on B1 or B2 with a pseudotime greater than P4. Consider the case of a READ(B2, P5, . . .) that is processed before the above transfer executes step 5. Then the value returned by the read at P5 will be the value of a version created before P1, and that version's maximum read pseudotime will be set to P5. The object history for B2 at this point is shown in Figure 5. When step 4 is executed, it will return the same value in its READ, but the write in step 5 will be rejected. The transfer atomic action will then fail, either explicitly changing Q's state to "aborted," or just by doing nothing and letting the commit record time out.

This case of READs aborting WRITEs should not be too surprising. It is analogous to aborting an atomic action because of a detected deadlock in a scheme that uses locking to achieve synchronization. The likelihood of frequent aborts due to this cause grows with the frequency with which separate atomic actions attempt to use the same object nearly simultaneously. We believe that in a decentralized system, such conflicts will be relatively rare, and so the overall performance degradation will be acceptable. It seems to be a price that must be paid in order to allow atomic actions to be constructed without knowledge of the other users of data they modify.

6. PRAGMATIC IMPLEMENTATION ISSUES

To implement these ideas effectively and efficiently, there are several issues that must be addressed. Here we outline approaches to some of these.

6.1 Representation of Versions

With two exceptions, versions are never modified once created. If we can eliminate the need for any modifications, we can benefit (a) in performance because fewer writes to stable storage will be needed and (b) by allowing versions to be implemented in a write-once stable medium such as an optical disk. As part of the SWALLOW project [1, 20, 22], we have solved these problems. To eliminate the need for modifying the chain pointers when adding a token, we restrict the implementation to adding new versions at the front of the list (so only more current versions can be added). To eliminate the need for modifying the maximum READ pseudotime, we observe that if new versions can be added only on the front of the list, then we need to store a maximum READ pseudotime only for the most recent version. The other maximum READ pseudotimes will be replaced by an implied value just less than the WRITE pseudotime of the next version. The maximum READ pseudotime for the latest version can be stored in the object header, or in a separate table maintained in faster storage that maps each object header to its maximum READ pseudotime. This latter strategy was finally chosen for SWALLOW. The table can be compressed by restricting the length of time, δ, that an atomic action can run before writing. Then the table need only store the maximum pseudotimes of current versions created less than δ time units ago. For all other objects, the maximum READ pseudotime will be $T - \delta$ where T is the current real time.

6.2 Impact of Distribution on Availability

The trick we used to ensure failure atomicity is to make the commit action for all tokens in a possibility atomic by representing the state of the possibility in one place—the commit record. There are two drawbacks to this trick. First, if data objects are stored on multiple nodes, reading a version will often involve delay due to accessing the commit record. However, once a read of a version discovers that the commit record is in the committed state, that information can be recorded with the version, so that later reads need not check the commit record state. Once all versions have recorded the state of the commit record (either turning from tokens into ordinary versions, or being erased), the commit record can be deleted. Second, the node containing the commit record becomes a critical resource, since its unavailability will prevent the state of many objects from being ascertained. This problem can be alleviated by distributing the commit record's state on multiple nodes, using a voting strategy [19].

6.3 Pruning Object Histories

For most practical systems, our implementation so far suffers from a serious problem. Since all versions of an object are stored forever, the total storage used by the system will increase at a rate proportional to the update traffic in the system. Consequently, we would like to be able to throw away old versions of the objects in the system. We can do this pruning of versions without much additional mechanism, however.

Before explaining how the pruning works, though, it is worthwhile to consider why one might *not* want to throw away old object versions. One reason is that the object histories provide exactly the information needed to go back to an earlier state of the system and restart it. One need merely pick a pseudotime that does not correspond to a point in the middle of an atomic action (in terms of the implementation sketched above, any N-bit time stamp is such a pseudotime), and use it to READ all the objects in the system. Since the objects are read in between atomic actions, the system state thus read will be consistent. This kind of checkpointing and backup to previous states is usually added on as a separate mechanism to systems without explicit mechanisms to support atomic actions. As a result, checkpoints in such systems usually can contain inconsistent data.

Another use for maintaining old versions is to support very large, slow, read-only atomic actions. If a read-only atomic action is executed in a system that maintains only one version of each object (e.g., as in a system that uses locks for synchronization), then it will either delay all computations that use the data it reads until it completes, or it will not be able to complete because writes to data it needs will make it impossible to access a consistent system state. If old versions are kept around, a very slow read-only atomic action will always be able to find the versions that it needs, even though writes at larger pseudotimes are executed, creating new versions. Such large read-only transactions do occur in practice, usually as the result of a need to provide an unanticipated kind of summary report about information maintained in a large database. Finally, we note that if a naturally write-once storage technology (such as laser-written optical storage) is used, retaining old versions is unavoidable (so we can "make a virtue out of necessity").

Assuming that we want to prune old versions, the only problem is to ensure that we do not allow writes to take place that would create new, different versions in the same range of pseudotime where an old one exists. We must also make sure that at least the most current committed version of each object is retained.

A simple and practical pruning mechanism works as follows: whenever a version has a pseudotime less than the most recent committed version, it is eligible for pruning. We only prune versions whose PTR field is less than a threshold equal to $T - d$ where T is the current real time, as obtained from the node's N-bit real-time clock, and d is a positive constant. The significance of d is that the only objects for which noncurrent versions are retained are those that were written by atomic actions that started within the last d time units. If d is zero, then only the most recent committed version of each object and any tokens created by atomic actions in progress are kept. Essentially this is equivalent to a locking strategy. Setting d to be greater than zero retains some old versions, so that late-arriving reads (from large read-only actions) can proceed. A good value for d can be chosen by noting the likely amount of time between the start of an atomic action and the time its last read arrives at the node containing the object read.

The amount of storage used for objects if this pruning algorithm is used is roughly the sum of two quantities—the total amount of storage used for the "current" versions of all the objects, plus the storage needed to hold any other versions of objects that were created by writes in the last d time units. The second quantity is roughly constant, assuming a constant update traffic, and is proportional to the overall system update traffic. Since most large databases do not have a high update traffic (i.e., a very small part of the database is actually modified during any short time period), the overhead is likely to be small.

If pruning occurs, the algorithms for READ and WRITE described earlier need to be modified so that READs and WRITEs whose pseudotime is older than $T - d$ are rejected. Such rejections add another way in which atomic actions abort.

6.4 Extensions

A major goal is that atomic actions be modularly composable operations. That is, one can implement atomic actions so that new atomic actions can be constructed out of previously existing atomic actions without either (a) modifying the preexisting implementations or (b) requiring that the new actions know what objects the preexisting atomic actions access. Locking mechanisms for providing synchronization or recovery for atomic actions make it difficult thus to compose atomic actions because of the need to have at least one instant of time where all data touched by an atomic action are locked. Composing atomic actions in a system based on locking thus requires extending the time during which an object is locked.

Implementing composable atomic actions requires extending pseudotemporal environments and possibilities. PTEs are extended so that a nested atomic action has exclusive access to a contiguous subset of the pseudotimes in its containing atomic action's PTE. Similarly, possibilities are given a hierarchically nested structure. The details of these extensions can be found in the author's dissertation [19].

A commit record can be deleted after the last token referring to it is either aborted or committed. This deletion requires an augmented protocol presented in [20].

In a distributed system, where communication is costly, it may improve performance to encache the state of an object at a site other than its home site. Versions of objects provide a useful unit of encachement, and strategies for distributing new versions of encaching sites can use the fact that the (object name, pseudotime) pair uniquely identifies the version.

If READs and WRITEs to objects in a distributed system are requested by messages, the mechanisms outlined in this paper work correctly, even if the communications system reorders the messages, duplicates them, or loses them. The reason for this is that the pseudotime and possibility required for each READ and WRITE provide enough identification to order each READ or WRITE request's effect on the object, and to ensure that a request is idempotent (may be executed repeatedly, with the same effect as if executed once).

7. CONCLUSIONS

In this paper, we have concentrated our attention on one aspect of synchronization—control of simultaneous access to shared data objects. It has been traditional to treat shared data synchronization with the same ideas and mechanisms as other problems of synchronization, such as disk queue scheduling and processor multiplexing, even though synchronization of access to data is a very simple and important case. The power of synchronization mechanisms has been measured by determining what "synchronization problems" they can and cannot solve, where such problems often have little to do with the important case of concurrent access to data.

As we have seen, by treating data synchronization alone, we need not be so concerned about the *timing* of programs accessing data, but rather we can concern ourselves with the more relevant requirement that the program access the correct states of the data. The division of synchronization into two classes, data access synchronization and process (timing) synchronization, seems to be a useful and powerful division.

Our view that a data object really stands for a sequence of states and that accesses (reads and writes) to the object are operations on that sequence is rather powerful. Pseudotime can be thought of as a naming mechanism for successive states of all objects in the system. Because we implement programs with that naming mechanism, programs accessing shared objects can be defined without need to consider their timing explicitly. Since timing of programs is one of the attributes of program execution over which the designer has little control (especially in distributed systems), reducing the importance of timing in understanding program execution simplifies the design task.

Pseudotimes and possibilities provide a "language" for describing the desired coordination of actions. In a distributed system, a remote request specified in high-level terms ("perform this set of account transfers") can be qualified with a pseudotemporal environment and possibility to ensure atomicity. This qualification can be specified without the requestor knowing what steps are carried out by the server. Thus one can ensure atomicity without compromising the ability to build abstract, modular interfaces. To work correctly, the modules that implement abstract requests need only pass the PTE and possibility through to the underlying READ and WRITE operations. If the basic operations are other than READ and WRITE, then assuming that these other basic operations are designed so that their state changes correspond to the pseudotime ordering and commit/abort with the possibility, one can still build atomic actions. Thus our scheme fits naturally with modular programming styles.

Our two-phase implementation of atomic actions is a close relative of the *two-phase commit* described by Gray [11] and Lampson and Sturgis [15]. In particular, the state of the commit record is analogous to the state of the coordinator process of Lampson and Sturgis. They also divide writes into two steps, creating a tentative value to be written and then doing the write. The major difference is that, in our scheme, the uncommitted state of the token prevents other atomic actions from reading it. In their schemes, uncommitted data are not stored as part of the object, which leads to two difficulties. First, additional synchronization is needed in their schemes, so they use write-locks. Second, writes are stored into the object only after the entire atomic action is committed, preventing the possibility of storing results as they are generated, in parallel with the computation. This parallel storing of results is a potential performance improvement that results from our scheme.

It is interesting to note that our object semantic model is somewhere between the traditional von Neumann machine semantics based on changeable memory locations and the more recent "side-effect-free" machine semantics best illustrated by data-flow machine architecture [7]. Although our objects can be updated, they are built on a substructure consisting of immutable *object versions* that correspond to the structured objects available in a data-flow machine. The immutability of object versions leads to the same advantage that is accrued from immutability in a data-flow architecture: the timing of concurrent programs is not important to the behavior of the program. However, by supporting an update semantics on top of the immutable versions, we support a user view of the system as an extensive memory with state-changing operations, a view that seems to be better for interuser sharing. Thus, we may have achieved the "best of both worlds."

It is worthwhile noting the relationship between the idea of maintaining object histories and two other ideas. An important notion in managing recovery of databases is the idea of a log [10]. Normally, a log consists of a time-ordered list of all old versions of data objects. Our object histories might be thought of as a "distributed log." The object history is also closely related to the history arrays of Dennis and Van Horn [8], which were used to describe the essential orderings among concurrent programs accessing shared data. In this context it is worthwhile to note that the idea of synchronization using object histories and pseudotimes allows us to increase parallelism, though that is not our primary goal. This is because the essential orderings are captured in the relationships between pseudotime values, rather than in the flow of control between steps of the programs.

In a system designed to be used in building modular abstract operations, both the synchronization and recovery mechanisms must be designed to preserve the degree of abstraction of the module interface. Either inadequate synchronization or inadequate recovery from failures could result in compromising the abstraction, and therefore both mechanisms must be present and correct to provide such abstractions. We have shown both a synchronization mechanism and a mechanism that provides limited backward error recovery that work well together in building atomic actions, a kind of abstract operation. We believe that such mechanisms must be designed to work together; the traditional approach of implementing reliability measures and synchronization measures independently would not work in the distributed computing environment.

ACKNOWLEDGMENTS

Jerry Saltzer read and criticized numerous drafts of this paper, and earlier performed the same task for the dissertation on which it is based. His help and encouragement have been primarily responsible for whatever lucidity this discussion has shown. Jim Gray of IBM San Jose Research provided the initial inspiration for this research during discussions we had in the summer of 1975; since then, he has been kind enough to share his fund of database "folklore" in discussing these ideas. Dave Clark, Liba Svobodova, Barbara Liskov, Irene Greif, Butler Lampson, Karen Sollins, and Anita Jones have also significantly shaped these ideas and their presentation.

REFERENCES

1. ARENS, G. Recovery of the swallow repository. S.M. thesis, Dep. of Electrical Engineering, M.I.T., Cambridge, Mass., Jan. 1981; Tech. Rep. TR-252, M.I.T. Lab. for Computer Science, Cambridge, Mass., Jan. 1981.
2. ATKINSON, R., AND HEWITT, C. Synchronization in Actor systems. In *Conf. Rec. 4th Symp. Principles of Programming Languages* (Los Angeles, Jan. 17-19), ACM, New York, 1977, pp. 267-280.
3. BERNSTEIN, P.A., SHIPMAN, D.W., ROTHNIE, J.B., AND GOODMAN, N. The concurrency control mechanism of SDD-1: A system for distributed databases (the general case). Tech. Rep. CCA-77-09, Computer Corporation of America, Cambridge, Mass., Dec. 1977.
4. BOBROW, D.G., BURCHFIEL, J.D., MURPHY, D.L., AND TOMLINSON, R.S. TENEX, a paged time sharing system for the PDP-10. *Commun. ACM 15*, 3 (March 1972), 135-143.
5. DAVIES, C.T. Recovery semantics for a DB/DC system. In *Proc. 1973 ACM Ann. Conf.* (Atlanta, Ga., Aug. 27-29), ACM, New York, 1973, pp. 136-141.
6. DAVIES, C.T. Data processing spheres of control. *IBM Syst. J. 17* (1978).
7. DENNIS, J. First version of a data flow procedure language. Tech. Memo TM-61, M.I.T. Laboratory for Computer Science, Cambridge, Mass., May 1975.
8. DENNIS, J.B., AND VAN HORN, E.C. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (March 1966), 143-155.

9. Dijkstra, E.W. Cooperating sequential processes. In *Programming Languages,* F. Genuys, Ed., Academic Press, New York, 1968.
10. Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19,* 11 (Nov. 1976), 624-633.
11. Gray, J.N. Notes on database operating systems. In *Operating Systems: An Advanced Course,* vol. 60, Lecture Notes in Computer Science, Springer-Verlag, New York, 1978, pp. 393-481.
12. Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques* C.A.R. Hoare and R.H. Perrott, Eds., Academic Press, New York, 1973.
13. Johnson, P.R., and Thomas, R.H. The maintenance of duplicate databases. ARPANET NWG/ RFC 677, DARPANET, Defense Advanced Research Agency, Washington, D.C., Jan. 1975.
14. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (July 1978), 558-565.
15. Lampson, B., and Sturgis, H. Crash recovery in a distributed data storage system. To appear.
16. Lomet, D.B. Process structuring, synchronization, and recovery using atomic actions. In *Proc. ACM Conf. Language Design for Reliable Software, SIGPLAN Notices* (ACM) *12,* 3 (March 1977), 128-137.
17. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15,* 12 (Dec. 1972), 1053-1058.
18. Randell, B. System structure for software fault tolerance. *IEEE Trans. Softw. Eng. SE-1,* 2 (June 1975), 220-232.
19. Reed, D.P. Naming and synchronization in a decentralized computer system. Ph.D. Dissertation, Dep. Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., Sept. 1978; Tech. Rep. TR-205, M.I.T. Laboratory for Computer Science, Cambridge, Mass., Sept. 1978.
20. Reed, D.P., and Svobodova, L. SWALLOW: A distributed data storage system for a local network. In *Local Networks for Computer Communications,* A. West and P. Janson, Eds., North-Holland, Amsterdam, 1981, pp. 355-373.
21. Stearns, R., et al. Concurrency control for database systems. In *Proc. IEEE Symp. Foundations of Computer Science,* (October), IEEE, New York, 1976, pp. 19-32.
22. Svobodova, L. Management of object histories in the SWALLOW repository. Tech. Rep. TR-243, M.I.T. Laboratory for Computer Science, Cambridge, Mass., July 1980.
23. Thomas, R.H. A solution to the update problem for multiple copy databases which use distributed control. BBN Rep. 3340, Balt, Beranek, Newman, Cambridge, Mass., July 1976.

Received November 1979; revised August 1982; accepted August 1982