M.I.T. Laboratory for Computer Science May 3, 1979

Computer Systems Research Division Request for Comments No. 172

Using Naming for Synchronizing Access to Decentralized Data

by David P. Reed

Attached is a draft copy of a paper I submitted to the 7th Symposium on Operating Systems Principles.

USING NAMING FOR SYNCHRONIZING ACCESS TO DECENTRALIZED DATA

by

David P. Reed

Massachusetts Institute of Technology

Laboratory for Computer Science

May 1, 1979

USING NAMING FOR SYNCHRONIZING ACCESS TO DECENTRALIZED DATA

Abstract

In this paper, a new approach to the synchronization of accesses to shared data objects is developed. Traditional approaches to the synchronization of access to shared data by concurrently running computations have relied on mutual exclusion--the ability of one computation to delay the execution of other computations that might access or change shared data accessed by that computation. Our approach is quite different. We regard an object that is modifiable as a sequence of immutable versions; each version is the state of the object after an update is made to the object. Synchronization can then be treated as a mechanism for naming versions to be read and for defining where in the sequence of versions the version resulting from some update should be placed. In systems based on mutual exclusion, the timing of accesses selects the versions accessed. In the system developed here, versions have two-component names consisting of the name of an object and a pseudo-time, the name of the system state to which the version belongs. By giving programs control over the pseudo-time in which an access is made, synchronization of accesses to multiple objects is simplified.

Our approach is intended to be used in an environment where unreliable components, such as communication lines and processors, and autonomous control of resources occasionally cause certain objects to become inaccessible, perhaps in the middle of an atomic transaction. Computations may also suddenly halt (perhaps as the result of a system crash) never to be restarted. Our approach provides facilities for recovering from such sudden failures, grouping updates into sets called possibilities, such that failure of any update belonging to a possibility prevents all of the other updates in that possibility. The pseudo-time naming mechanism also provides a useful tool for restoring a consistent state of the system after a failure resulting in irrecoverable loss of information or a user mistake resulting in an inconsistent state.

Keywords: data synchronization, locking, recovery, distributed systems, data bases, atomic actions.

Introduction

The research reported here was begun with the intention of discovering methods for combining programmed actions on data at multiple distributed 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.

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 of 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 [15]. Atomic actions form one such class of modules. In this paper, we describe a new method for synchronization and failure recovery that works well in a distributed 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 [17], from which this treatment was derived.

Decentralized Computer Systems

By decentralized computer systems we mean a set of computer nodes consisting of processor, memory, and permanent storage (disk), connected together by a communications network. Each node can be, and often is, used as a powerful stand-alone 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 programs to be executed that may result in modifying data at that node or retrieving data from that node. In what follows data is modelled as record-like objects that may contain references to other data objects either on the same site or on other sites. The algorithms that manipulate such data are composed of programs existing at multiple sites that communicate by message passing. Coordination is required because there may be several computations that access the same objects in the system at any one time, initiated by independent users of the system. As such, the concurrency that concerns us is the unplanned kind, rather than the kind of concurrency designed into an individual computation to speed it up.

Coordination of concurrent processes is difficult in a distributed system because of communication delays and modularity. In a centralized system with shared memory, coordination can be achieved inexpensively by locking the data to be accessed while the computation uses it. 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 between the users of the data and therefore substantial communications delays. Furthermore, deadlock detection is impractical because it requires global knowledge of all computations and their locks. Deadlock avoidance is impractical because a module of a distributed computation that uses modules on other nodes may not have knowledge of the data accessed or the order of access at those nodes.

Recovery from failures is made difficult in a distributed system by the peculiar nature of communication failures. In particular, when node A requires a service from node B that involves modifying data objects stored at node B, certain kinds of communication failures will leave node A in doubt as to whether node B has performed the requested action or not. The requesting computation at node A has only one option at this point, since further actions by node A are usually contingent upon successful completion of the request at B to insure consistency between various parts of the system. Node A must block until node B's state can be ascertained, but this may take a very long time. If node A holds resources needed by other computations, then such a failure can cause deadlock.

In a monolithic distributed data base, such failures may be tolerable, since each node and communication link is maintained to a high standard of availability. In a system where nodes are autonomously managed, such failures are more likely to happen, and more likely to be of long duration. For example, after node A sends its request, but before B responds, node B (a desktop computer) may be powered off for lunch.

Atomic Actions

The goal of this research is to support the construction of atomic actions. An atomic action is an operation on data whose effects on data are completely specified by the algorithm executed by the atomic action. In particular, though the atomic action may access (read or update) many pieces of data, each many times, as part of its execution, the effect of the atomic action can be described as a relation between the initial state of all of the

Your description of atomic actions is too abstract.

data items it touches and their final states when the atomic action is finished.

Atomic actions require both synchronization and recovery mechanisms in their accesses to data. Synchronization is required to ensure that no other computations within the system can observe an intermediate state of the data objects accessed. If an intermediate state of an object could be observed outside the atomic action then the behavior of the atomic action could not be specified solely in terms of a relation between initial and final states of the objects accessed. Synchronization is required to ensure also that no other computation can modify any data object used by the atomic action during its execution. That is, the atomic action's program can be written without any consideration of interference from concurrent access to the data it accesses. Recovery mechanisms are required to ensure that if a failure occurs, preventing completion of an atomic action, that the intermediate state of the data resulting from partial completion of an atomic action is not exposed to observation by other computations.

This is a non-obvious implication [illegible]

Our concept of atomic actions is quite similar to that of Lomet [14] and also similar to the sphere of control described by Davies [6,7]. If all computations in the systems 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. [8].

The simplest implementation of atomic actions is to delay all other computations in the system for the duration of the atomic action. This is often inefficient in a single processor system, but in a distributed system connected by a network, it may be impossible, because of communications failures.

It is sufficient, however, to guarantee that an atomic action has exclusive access to the data it actually reads or updates. Locking is often used to achieve this, by associating a lock (or mutual exclusion semaphore) with each data object that will be used by a computation before that computation can access the data. Locking introduces the possibility of deadlock, the detection of which may be quite difficult in a distributed system, while classic deadlock avoidance techniques cannot cope with transactions whose data accesses are unknown, due to the presence of information-hiding mechanisms that hide the representations of objects, or due to the use of pointer or accesses otherwise predicated on values obtained earlier in an atomic action's execution.

The essence of locking is to seize exclusive access to a group of objects _for a period of time_. Thus, the proper behavior of an atomic action is controlled _indirectly_, by ensuring that the timing of its steps is properly coordinated with the timing of other computations. The basis of the locking approach to implementing atomic actions is that there is one instant or interval during the atomic action at which all locks are simultaneously held. That interval must either precede or follow the corresponding interval of any potentially interferring atomic action.

In contrast, the mechanism proposed herein coordinates the access to a set of objects by a naming mechanism that gives names to _versions_ (virtual global states) of the system. There are two naming mechanisms described below. _Pseudo-times_ are a totally ordered set of names referring to successive virtual states of the system's data. Possibilities are a mechanism for referring to groups of updates to objects for the purpose of error recovery.

Atomic actions are implemented by giving the virtual processor executing the atomic action exclusive use of both a sequence of pseudo-times and a possibility. Access to a particular object in a particular state of the system requires that both a possibility and a pseudo-time for that state of the system be used as parameters to the access. There is a very close analogy between this approach to implementing atomic actions and the capability approach to protection of data [5,9]. In both approaches, having a name for something is a prerequisite for its use, so exclusive use can be granted by restricting the propagation of names.

The following discussion elucidates the properties of pseudo-times and possibilities, and illustrates their use in the construction of atomic actions.

Pseudo-time and Known Histories

Each object is represented as a sequence of versions.* Each version of an object represents a particular state that the object attains during its life. An update to an object is implemented by creating a version of the object and assigning that version to its proper place in the sequence of versions.

* Object versions were inspired by the treatment of synchronization in Stearns, et al. [18], though our mechanism using known histories and pseudo-time is quite different. Also closely related is the practice of using version numbering for modifications to files, as in TENEX [3], though such operating systems provide no mechanism for interfile consistency.

At any point in time, the object has a known history, which is an ordered sequence of versions of the object history that have been created by updates. As time proceeds, the known history is educed (led out) by reads and updates to the object. The eduction will converge toward a complete history of the object. The mechanism of eduction will be discussed shortly. Please note, however, that the time order of creation of an object's known history may be different than the order of versions in the known history.

Versions of different objects are correlated by a correspondence between the versions of an object and a totally ordered set called pseudo-times. For each object, pseudo-times serve as indices to the versions. That is, there is a mapping from pseudo-time to versions that is a function, so for a particular pseudo-time, its image under the mapping is at most one version. Each object has a creation pseudo-time and a deletion pseudo-time. The mapping from pseudo-time to versions has the property that for all pseudo-times between the creation and deletion pseudo-times there may exist corresponding versions, and for all other pseudo-times, there are no corresponding versions.

The pseudo-time ordering orders the versions in each object's known history. A pseudo-time stands for a particular (virtual) system state, where the version of each object that corresponds to the given pseudo-time is that object's contribution to the state. As we shall see, the pseudo-time ordering of states is somewhat decoupled from the real time ordering of events in the system. For example, suppose X and Y are pseudo-times, such that X<Y. Then, the version of object A corresponding to pseudo-time Y may be incorporated into A's known history at a real time when the version of object B corresponding to pseudo-time X is yet undetermined.

The desirable property of this definition of system states is that the definition does not require simultaneity in real time. Thus, requiring that the initial values of all objects accessed by an atomic action be from the same system state can be implemented by using the same pseudo-time to select each version accessed.

Figure A is a pictorial representation of a group of known histories. The circles represent object versions. Each version corresponds to a range of pseudo-times, indicated between the arrows connecting it to the pseudo-time axis. The versions making up the global system state A are X.1, Y.2, and Z.2. The versions making up the system state B are X.2 and Z.2. At the real time these known histories were observed,* there is no version yet decided for object Y in system state B. That version will be decided later.

Programs and Pseudo-time

To simplify the discussion here, we assume that each independently initiated computation is represented by a sequential process.** An individual process executes a program that consists of one or more atomic actions to be performed (where such atomic actionss are specified as programs making one or more data accesses). Grouping of program steps into atomic actions can be specified by some relatively simple syntax, as shown in the later example. We

---

* Presumably some omniscient observer, since in a distributed system it may not be possible to observe all objects simultaneously.

** The case where computations have internal concurrency, though too complex to handle here, can be treated by natural extensions to the mechanisms described here [17]. Note also that these sequential processes can model computations that need not be executed sequentially--for example, successive reads of two distinct data objects can be executed concurrently in real time.

discuss here the structure of an underlying implementation using pseudo-time and possibilities.

All programs access data with an implicit specification of the pseudo-time to be used in accessing the data. That is, when a program attempts to read a value from data object X, the underlying implementation specifies the particular pseudo-time to be used in conjunction with X to select the proper version of X. Similarly, when a program attempts to update X, the underlying implementation furnishes a pseudo-time that is used to select the place in the object history where the version resulting from the update should be put.

The mechanism for providing the proper pseudo-time for accesses and updates is the _pseudo-temporal environment_ (PTE). The pseudo-temporal environment is a part of the virtual processor executing the program, much the same as are the program counter and the object name resolution environment. Essentially, the pseudo-temporal environment is a source of monotonically increasing pseudo-times selected from a given subset of all pseudo-times. Monotonicity is required within a process so that the pseudo-time ordering of system starts reflects the causal ordering of steps within a process.

Each object access or update involves selecting a new pseudo-time from the pseudo-temporal environment of the executing program, to be used to specify the version accessed or updated. Since the pseudo-times selected for a given executing program increase from access to access, each program step sees a successively later state of the system.

Atomic Actions and Pseudo-temporal Environments

An atomic action is executed in a pseudo-temporal environment that provides exclusive access to a particular range of pseudo-time. No program executed outside the atomic action will have pseudo-times in its PTE that lie between two pseudo-times in the atomic action's PTE. As shown in figure B, each atomic action has exclusive access to a contiguous region of pseudo-time, while processes in general may access several regions of pseudo-time.

Creation of an atomic action requires the construction of a pseudo-temporal environment that grants the atomic action exclusive access to a region of pseudo-time that no other computation can access. In some sense, then, pseudo-times are used for synchronization in much the same way as capabilities are used for protection--if a program cannot name a system state, then it cannot access it. For this reason, the present mechanism can be described as a naming mechanism for achieving synchronization.

Implementing Known Histories

Objects are implemented as a set of versions that comprise the known history. One feasible implementation is as a singly-threaded list of versions, where each version is marked with the start and end pseudo-times of the range of pseudo-times that refer to the version. In figure C, we illustrate such a list. The following discussion assumes that once a version of an object is created, it will be stored forever. Practical implementation requires developing a strategy for throwing away old versions, a simple example of which is described in the author's dissertation [17].

Reading an object requires searching the known history for a version whose range of pseudo-times encompasses the pseudo-time selected from the PTE controlling the access. If such a version is found, then the access simply returns that version. If not, the proper version has not yet been chosen for that particular pseudo-time, so the completion of the access requires eduction of the known history.

One way in which an object's known history is educed is by updates to the object. An update is completed by installing a new version containing the required value into the proper point in the known history, with start and end pseudo-times equal to the pseudo-time selected for the update from the computation's pseudo-temporal environment. (See figure D.) Since it is required that the mapping from pseudo-time to versions of an object be functional, there is the possibility that an attempt to install a new version may be blocked by a version that already exists corresponding to the desired pseudo-time. In the case of such blockage, the update will fail, with no effect on the known history. Such failures will be discussed shortly.

The other way an object's known history is educed is by extending the range of pseudo-time covered by one version. That is, the end pseudo-time may be increased so that the version corresponds to a longer range of pseudo-times. This type of eduction corresponds to "inertial" behavior of objects--an object not updated remains the same. The cause of such an eduction is an access at pseudo-time P that is not satisfied by any version in the known history. In order to respond with a value, the version whose end pseudo-time is the greatest value still less than P is educed so that its end pseudo-time is equal to P. That version can then be used to satisfy the access. In figure E, an access at pseudo-time P causes the version whose end pseudo-time is Y to be educed.

Example

To illustrate what has gone on so far, consider the following simple example of an atomic action to subtract A from the balance of a bank account, and add A to the balance of another.

```
begin atomic action
     bal_1 := bal_1 + C
     bal_2 := bal_2 - C
end atomic action
```

The cells bal_1 and bal_2 are objects. The atomic action makes four uses of bal_1 and bal_2, two reads to obtain the values of bal_1 and bal_2, and two updates to set the new values. The value of bal_1 is to be obtained at pseudo-time P1, and updated at P2. The value of bal_2 is obtained at P3 and updated at P4. Assume that bal_1's initial value at P0 is X and bal_2's is Y. Because of the order of execution of the steps of the action, we know $P1 < P2 < P3 < P4$.

The steps are pictured in figure F. First, bal_1's known history is educed by extending the version at P0 to correspond to P1. Then bal_1 is updated by creating a new version valid in P2. Similar steps are then performed for bal_2.

We have insured that no other computation can use the objects at any pseudo-time between P1 and P4 inclusive. Thus, the values observed for bal_1 at P1 and for bal_2 at P3 must be unchanged from the initial values. Furthermore, with the exception of the changes to bal_1 and bal_2 that are explicitly made, no other object value changes will occur between the initial system state corresponding to P1 and the final system state corresponding to P4. We have not completely implemented atomic actions, however, since it is still possible by our eduction rules that a computation reading at later pseudo-times than P4 could read both the new version of bal_1 and the old version of bal_2. The token mechanism about to be described prevents this eduction.

Failures

The remaining knotty problem in implementing atomic actions is the handling of failure. If the atomic action example above failed to complete (was not able to perform the update at P4 for some reason), then other computations might be able to observe the intermediate state at P2 by extending the range of pseudo-times covered by the new version of bal_1 and the old version of bal_2. There are many possible re asons for such a failure, such as:

a) inability of the computer controlling the current process step to communicate with the site containing bal_1 after requesting the update at P2, but before it is known that the update completed,

b) a failure of some node or communications link preventing forward progress of the process,

c) a protection violation is incurred in trying to use bal_2 (perhaps resulting from revocation of access), etc.

Not all of these reasons can be detected in advance of the attempt to access bal_2.

In the mechanism for managing known histories, another possible occasion for failure was introduced. If a version of bal_2 already corresponds to P4, for example, the update cannot be performed. This failure is easily handled in the same way the other failures listed above must be handled--by _backward error recovery_ [5,6,16]. That is, all the updates made by the atomic action in the course of its execution must be undone.

Backward error recovery is potentially difficult in the presence of concurrency, since undoing an update may require undoing the effects of other computations that used the value created by the update as input. This can lead to a variety of problems, e.g., the _domino effect_ [16]. None of these problems will be encountered with respect to atomic actions if the implementation respects the defining rule that no computation outside the atomic action can observe the intermediate states the system attains during the atomic action.

Tokens and Possibilities

The mechanism used to implement backward error recovery is a close relation to the _two-phase commit_ mechanism described by Gray [10] and Lampson and Sturgis [13]. That is, all updates are made tentatively until the atomic atomic action finishes, whereupon a single primitive atomic action installs all updates permanently.

A tentatively created version is called a token. Tokens are grouped into sets called possibilities. Eventually either all tokens belonging to a possibility are committed, made actual versions, or all tokens of a possibility are aborted, completely erased. All updates made in an atomic action are in the same possibility, thus all updates are committed or all are aborted. Aborting a possibility allows backward error recovery in the case that a failure is discovered before the atomic action terminates.

Each computation executes in a possibility. That is, another part of the state of the virtual processor executing a computation is the name of the possibility in which it is executing. All updates performed by the computation result in the creation of tokens belonging to the possibility in which the computation executes. Tokens are like versions, in that they correspond to a range of pseudo-time in the object known history, and have a value, but there are restrictions on which computations can access the value of a token or extend its range of pseudo-time. When some computation attempts to read an object in a pseudo-time that either maps to a token in the known history or maps to a point where the immediately preceding item in the known history is a token (in figure G, where diamonds represent tokens, Q and R are such pseudo-times), the read must wait if the token is not a member of the possibility in which the reads is attempted. Such reads will wait until the possibility containing the token is either committed or aborted.

This waiting is the mechanism that guarantees that objects updated by a partly completed atomic action are not observed outside that action. We must take special care, however, that an updated object created by an atomic action can be read later in that action. For this reason, if a read that maps to a token as above, is executed in the possibility that contains the token, the token's value will be returned, and, if necessary, the end pseudo-time of the token will be increased to the pseudo-time of the read.

Consider, for example, an atomic action that updates an object, then reads it several times. The atomic action will create a token as the result of the update, but the possibility associated with the update cannot be committed until the atomic action completes, to guard against later failures, yet the reads attempted by the atomic action should be allowed. The rule for token accessibility gives the correct behavior.

Implementing Atomic Actions

An atomic action is executed in a pseudo-temporal environment that gives it exclusive access to a range of pseudo-time. In addition, it executes in a possibility that is used by no other computation, thereby guaranteeing that all the tokens it creates are visible within the atomic action, but not outside the atomic action. The final step of an atomic action is to commit its possibility. Since that is the final step, if the atomic action cannot complete due to any failure, its possiblity will never be committed.

To guard against a possibility remaining in a uncommitted, unaborted state forever, a timeout is associated with it. If the timeout expires while the possibility is not yet committed, the possibility is automatically aborted.

More Example

Let us consider the earlier bank account example further, given the new ideas of tokens and possibilities. At the beginning of its execution, the atomic action creates a pseudo-temporal environment and a possibility, Q, and loads them into its virtual processor. It then access bal_1 at pseudo-time P1, extending the range of the existing version. Then it updates bal_1 to its new value at P2, making the resulting token a member of Q. Should a failure be encountered past this point, the possibility Q will be aborted, either automatically by timeout, or explicitly by the atomic action. Consequently, the new token for bal_1 will not be observed outside of the atomic action should a failure occur. Assuming no failures, bal_2 will be accessed and updated, creating another token as part of Q. Once all has gone correctly, the process executes the primitive atomic action that commits all members of Q.

Now consider the interaction of the example atomic action with some other concurrent computation that reads or updates bal_2. Because the atomic action has exclusive access to its pseudo-temporal environment, the pseudo-time, P5, of such an access must be either less than P1 or greater than P4.

A read at P5 < P1, executed before the atomic action accesses bal_2 the first time, will extend the range of pseudo-time, for the initial value to P5, returning the initial value. Then the atomic action will extend the range of P3, also returning the initial value. If the read is attempted after the atomic action's access to bal_2 (but P5 < P1), then the atomic action will have already extended the range to P1. Thus it is irrelevant which order the two accesses actually are processed at the object's known history.

The order of processing does matter, however, if P4 < P5. If the update to bal_2 by the atomic action is processed first, then a token will be created. Then the read at P5 will discover the token, and since it is not executed in possibility Q, the read will wait until the Q is either committed or aborted. If Q is committed, then the token will become a version whereupon the access at P5 will extend it. If Q is aborted, then the token will be removed from the known history, and the initial version of bal_2 will be extended to P5.

The other order, where the access P5 occurs first, is more interesting. When the update to bal_2 is attempted, it will fail, since the initial version will have been extended to P5, covering the point of update at P4. Thus the atomic action will be aborted. In the following discussion, we show how to reduce the frequency of such aborted actions.*

Relation of Pseudo-time and Real Time

The problem of a read aborting an atomic action doing an update only results if the time at which the read is attempted precedes the time of the update, but the pseudo-time of the read is greater than that of the update. We can drastically reduce the likelihood of this occurrence by ensuring that the pseudo-temporal environment of an atomic action started at time t contains pseudo-times less than the pseudo-temporal environment of an atomic action started later than t.

Of course, in a distributed system with highly variable communications delays, an atomic action may be delayed so that its data accesses are attempted later than accesses to the same data by an atomic action starting later. In such an event, it is entirely possible that the earlier-starting atomic action will be aborted. This is a price that must be paid in order to

* There is a close analogy between this aborting described here and the aborting resulting from a detected deadlock in a scheme with deadlock detection.

support atomic actions where the set of data objects used and modified cannot be predicted in advance (an alternative scheme might use locking of data at the time an access is attempted, with deadlock detection, but it would be still necessary occasionally to abort atomic actions that have done a substantial amount of work).

Implementation of Pseudo-times and Pseudo-Temporal Environments

Pseudo-temporal environments for atomic actions are ranges of pseudo-time, guaranteed to be different from any other pseudo-times in the system. A pseudo-temporal environment should be easily constructed, since one is required for each new atomic action. The constraint just mentioned, that later-constructed pseudo-temporal environments have greater pseudo-times than earlier ones, should also be honored.

One implementation that satisfies these constraints uses approximately synchronized real-time clocks at each node of a distributed system. Clocks can be synchronized easily to within microseconds using the WWV time standard. Lamport's clock synchronization mechanism would also suffice [12]. These clocks are used to create timestamps 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 timestamps they generate are disjoint.

A pseudo-temporal environment for an atomic action is represented as a two-component structure consisting of the timestamp from its creation, and a timestamp read as part of the last selection of a pseudo-time from it. For simplicity, assume that each of these quantities is specified as an N-bit integer.

A new pseudo-time is selected from the pseudo-temporal environment by getting a timestamp and prefixing it with the timestamp of creation of the PTE. (See figure H.) The timestamp read must be greater than the timestamp 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 timestamp last selected in the PTE.

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

Thus far, we have discussed the PTE's and pseudo-times associated with atomic actions. Processes that make accesses to data outside of atomic actions nonetheless use pseudo-times. Such pseudo-times are derived from a simpler kind of PTE that consists only of a cell containing the time of last selection. A new pseudo-time is selected by just reading a timestamp and storing it in the PTE. Treating these N-bit timestamps as binary fractions orders them correctly with respect to timestamps belonging to atomic actions.

The use of timestamps for synchronization was originally developed by Johnson and Thomas [11]. Later work by Thomas [19] and Bernstein, _et al_. [2] have carried this approach further. We were inspired by these approaches, though they lacked the crucial insight of using a set of timestamps as a capability to gain exclusive use of a set of system states.

Implementing Tokens and Possibilities

To implement tokens, the representation must keep track of which tokens belong to each possibility, and provide a single primitive action by which all tokens in a possibility can be committed. The implementation shown here is a particularly simple two-phase commit protocol [10,13].

A possibility is implemented as a commit record, a piece of data that has three possible states--unknown, committed, and aborted. In addition the commit record has a time of expiration, after which it is automatically aborted if not yet committed. In a distributed system, the commit record is placed at a particular site.* When a commit record is created, it is in the "unknown" state, and its expiration time is set.

Tokens in a possibility contain a pointer to the commit record representing the possibility. An attempt to access a token first checks to see if the possibility in which the access is attempted matches the one containing the token. If so, the token is accessed as if it were already a version. If not, the current state of the commit record is checked--if still "unknown", the access waits, if aborted, the token is deleted and the access is reattempted, and if committed, the token is made a version and the access is completed.

Since the commit record permanently records whether the possibility is committed or aborted, and transmission of this information is triggered by

---

* It is possible to distribute the implementation of commit records so that the state of the commit record is not kept at a single site. Such a strategy enhances availability of the commit record's state at the cost of a more complex implementation. See the author's dissertation for details of this technique and other subtle issues of the implementation of possibilities.

attempts to access tokens, committing or aborting all of the tokens in a possibility is accomplished by simply setting the state of the commit record. Unlike the protocols of Lampson and Sturgis [13], Gray [10], or Alsberg [1], this protocol is extremely simple to understand and get right. The provision of a simple way to abort a whole possibility explicitly or by timeout is also novel.

Further Topics

The approach described here was first described in the author's doctoral dissertation [17]. Space in this paper does not permit discussion of a number of very important issues briefly outlined here. The interested reader is referred to the dissertation for details.

A major result is that atomic actions are 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 is 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 pseudo-temporal environments and possibilities. PTEs are extended so that a nested atomic action has exclusive access to a contiguous subset of the pseudo-times in its containing atomic action's PTE. Possibilities are extended so that the outermost atomic action is the union of <u>dependent possibilities</u>.

Because multiple versions of objects are maintained, atomic actions that read very large numbers of geographically distributed objects are easily executed concurrently with later updates. It is thus unnecessary for such large transactions to lock out all other atomic actions.

Using pseudo-times as names of system states allows backing up a set of objects to an earlier consistent state, by reading the value of these objects as of an earlier pseudo-time known to correspond to a consistent state, and performing updates in the present system state. Since all versions of objects are saved (but see below), taking a checkpoint for such backup simply requires remembering a pseudo-time. Pseudo-time thus also provides an indexing technique for backup, given that we add a state restoration mechanism that, given a pseudo-time saved at a checkpoint, will restore the states of a set of objects to that of the checkpoint.

Object known histories as described here have no provision for forgetting about old versions that will never be read again. At the very least, such versions should be moved to archival media for backup (given the checkpointing strategy above), and probably should be garbage-collected. We have a scheme for garbage-collecting such old versions that has the property that for objects that have not been recently updated, only the most recent (in pseudo-time) version need actually be saved.

We also have a garbage-collection mechanism for possibilities that removes a commit record when the last token referring to it is either aborted or committed.

In a distributed system, where communication is costly, it is often useful 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 several strategies for distributing new versions to encaching sites can use the fact that the (object name, pseudo-time) pair uniquely identifies the version.

If reads and updates 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 pseudo-time and possibility required for each read and update provides enough identification to order each read or update requests effect on the object, and to ensure that a request is idempotent (may be executed repeatedly, with the same effect as if executed once).

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 such sunchronization with the same ideas and mechanisms as other problems of synchronization, such as disk queue scheduling and interprocess control communication,* even though synchronization of access

---

* Interprocess control communication is a generic term for mechanisms that allow a process to block itself when it has nothing to do, allowing a physical processor to be multiplexed among many processes. IPCC may be used in implementing controls for access to shared data, though it is more generally useful.

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 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 updates) to the object are operations on that sequence is rather powerful. By defining a naming mechanism for selecting the point in the sequence of states to be operated on and implementing programs with that naming mechanism, programs accessing shared objects can be defined without need to consider their timing. 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.

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 more recent "side-effect free" machine semantics best illustrated by dataflow machine architecture [4]. 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 dataflow machine. The immutability of object versions leads to the same advantage that is accrued from immutability in a dataflow architecture, that 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 charging operations, a view that seems to be better for inter-user sharing. Thus, we may have gotten the "best of both worlds."

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. Both improper synchronization and improper recovery from failures may 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.

References

[1] Alsberg, P.A., Belford, G.G., Day, J.D., and Grapa, E. Multi-copy resiliency techniques. University of Illinois CAC Document 202, May, 1976.

[2] 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). Computer Corporation of America Technical Report CCA-77-09 (December 15, 1977).

[3] Bobrow, D., et al. TENEX--a paged time sharing system for the PDP-10. Comm. ACM 15, 3 (March 1972), 135-143.

[4] Dennis, J. First version of a data flow procedure language. M.I.T. Laboratory for Computer Science Technical Memo, TM-61 (May 1975).

[5] Dennis, J., and Van Horn, E. Programming semantics for multiprogrammed computations. Comm. ACM 9, 3 (March 1966), 143-153.

[6] Davies, C.T. Recovery semantics for a DB/DC system. Proc. 1973 ACM National Conf. New York (1973), 136-141.

[7] Davies, C.T. Data processing spheres of control. IBM systems Journal 17, 2 (1978), 179-198.

[8] Eswaran, K., Gray, J.N., Lorie, F., and Traiger, I. The notions of consistency and predicate locking in a database system. Comm. ACM 19, 11 (November 1976), 624-633.

[9] Fabry, R. Capability-based addressing. Comm. ACM 17, 7 (July 1974), 403-412.

[10] Gray, J.N. Notes on database operating systems. Operating Systems: An Advanced Course, Volume 60 of Lecture Notes in Computer Science, Springer-Verlag, (1978), 393-481.

[11] Johnson, P.R., and Thomas, R.H. The maintenance of duplicate databases. ARPANET NWG/RFC 677, January 1975.

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. Comm. ACM 21, 7 (July 1978), 558-565.

[13] Lampson, B., and Sturgis, H. Crash recovery in a distributed data storage system. Working paper, Xerox PARC ca. November 1976.

[14] Lomet, D.B. Process structuring, synchronization, and recovery using atomic actions. Proc. ACM Conf. Language Design for Reliable Software, Raleigh, N. C., (March 1977). Available as ACM SIGPLAN Notices 12, 3 (March 1977), 128-137.

[15] Parnas, D.L. On the criteria to be used in decomposing systems into modules. Comm. ACM 15, 12 (December 1972), 1053-1058.

[16] Randell, B. System structure for software fault tolerance. IEEE Trans. on Software Engineering SE-1, 2 (June 1975), 220-232.

[17] 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. Also available as M.I.T. Laboratory for Computer Science Technical Report, TR-205, September 1978.

[18] Stearns, R., et al. Concurrency control for database systems. IEEE Symposium on Foundations of Computer Science CH1133-8C, October 1976, 19-32.

[19] Thomas, R.H. A solution to the update problem for multiple copy databases which use distributed control. BBN Report 3340, July 1976.

Figure A

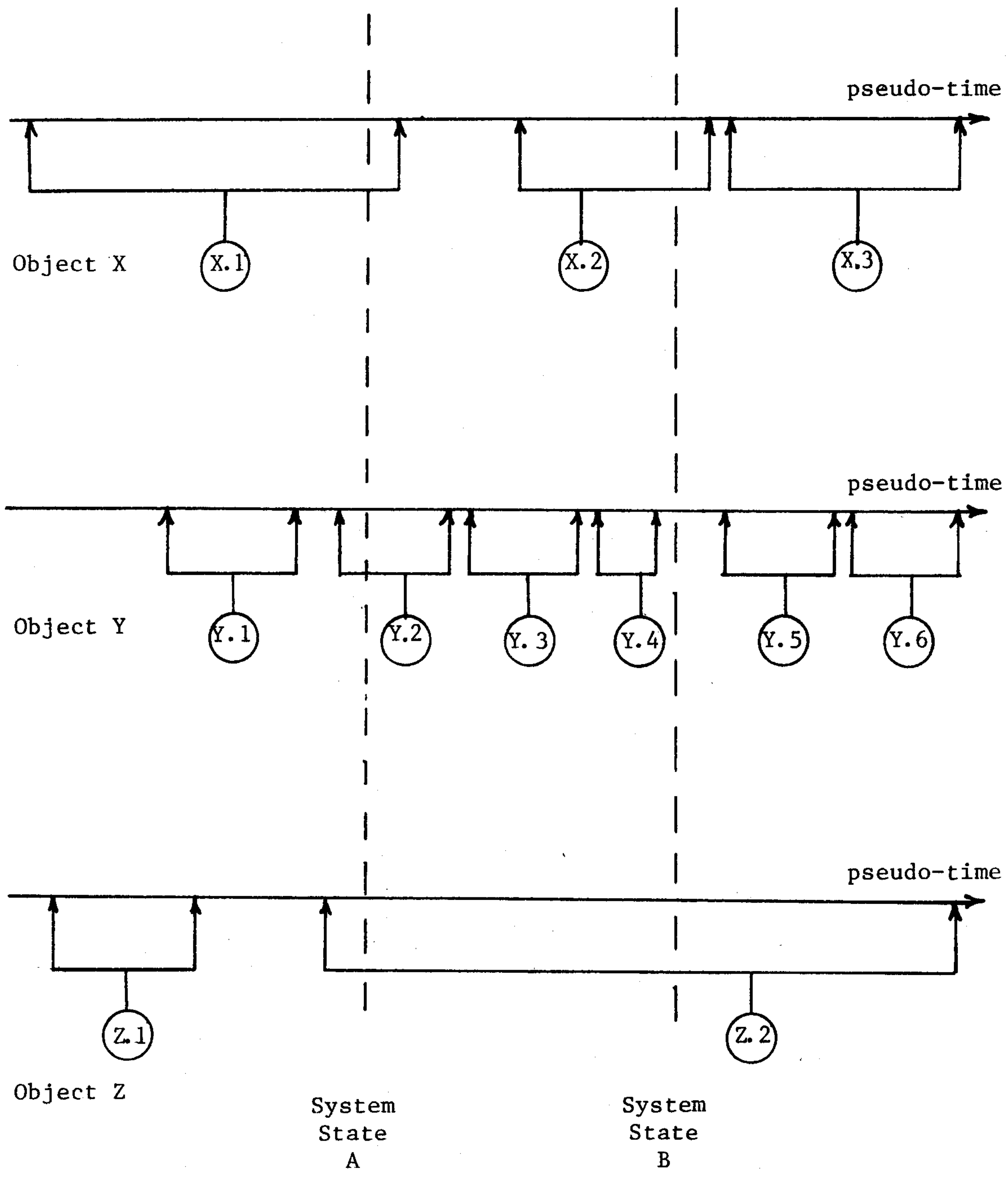

pseudo-time
Object X
X.1
X.2
X.3
pseudo-time
Object Y
Y.1
Y.2
Y.3
Y.4
Y.5
Y.6
pseudo-time
Z.1
Z.2
Object Z
System State A
System State B

Figure B

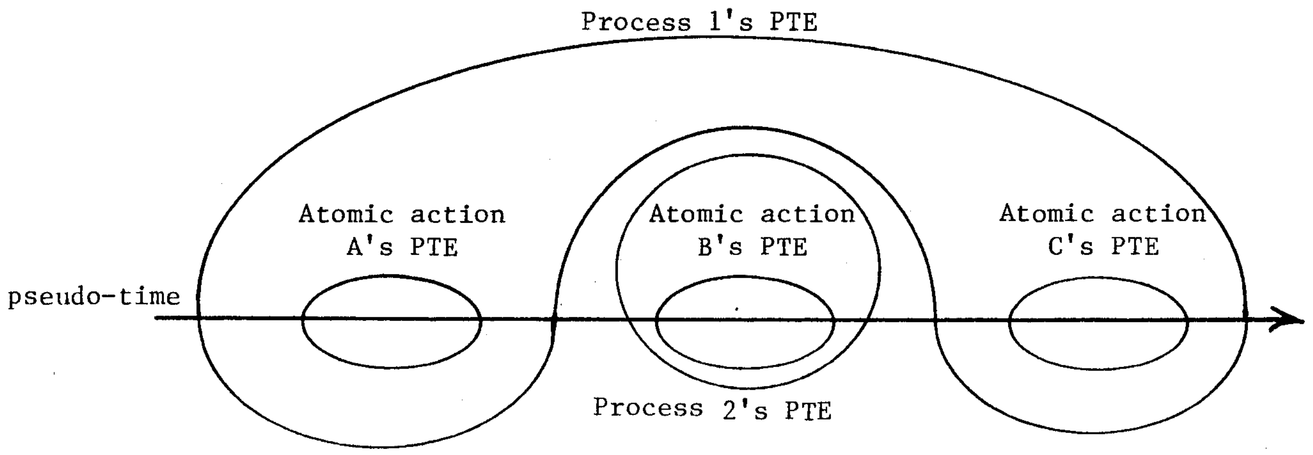

Process 1's PTE
Atomic action A's PTE
Atomic action B's PTE
Atomic action C's PTE
pseudo-time
Process 2's PTE

Figure C Known History of an Object

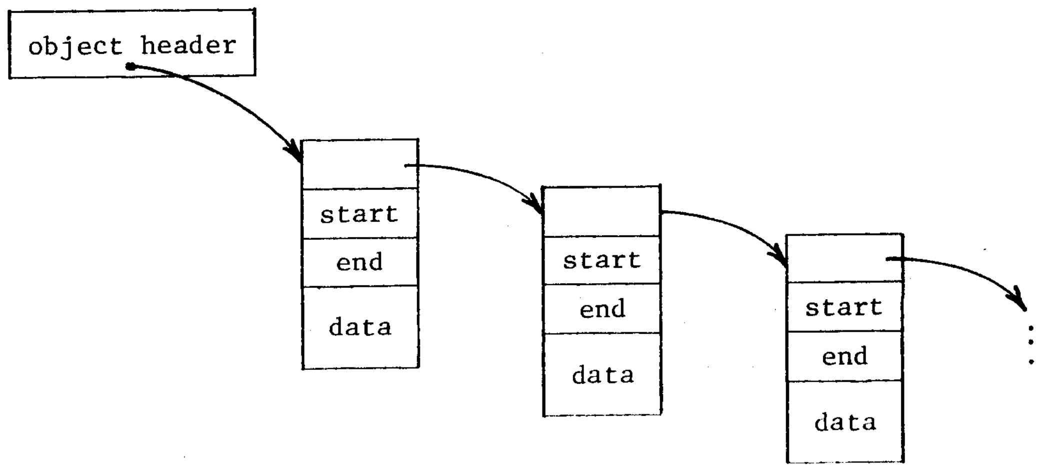


Figure D Updating an Object by Adding a Version

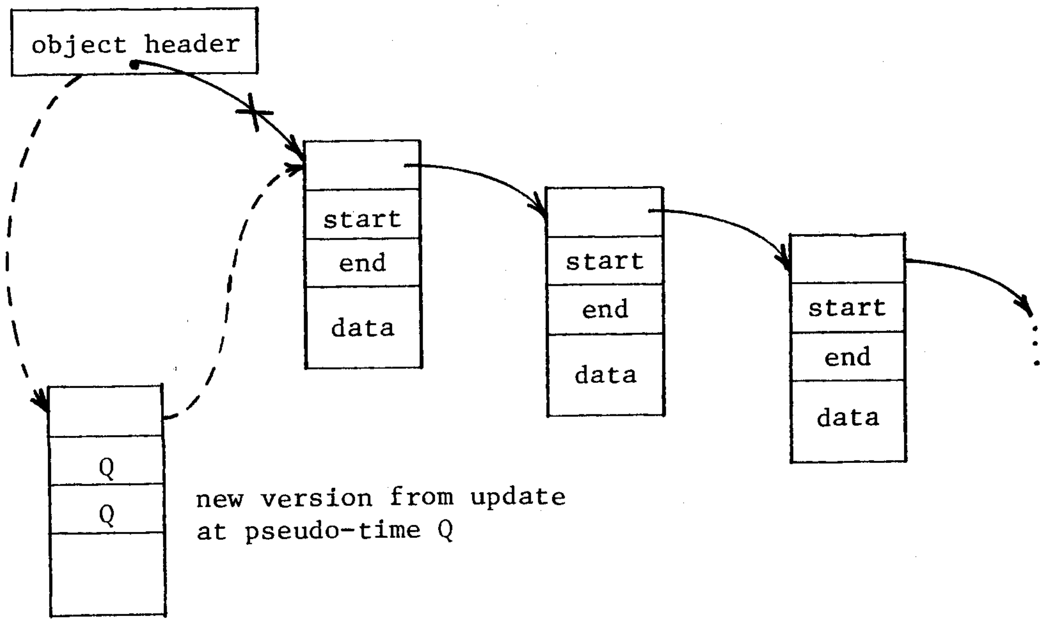


Figure E Extending the range of a version

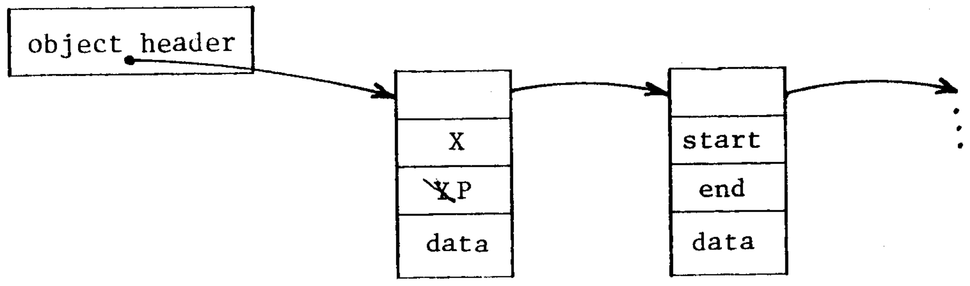


Figure F Example eduction of two known histories

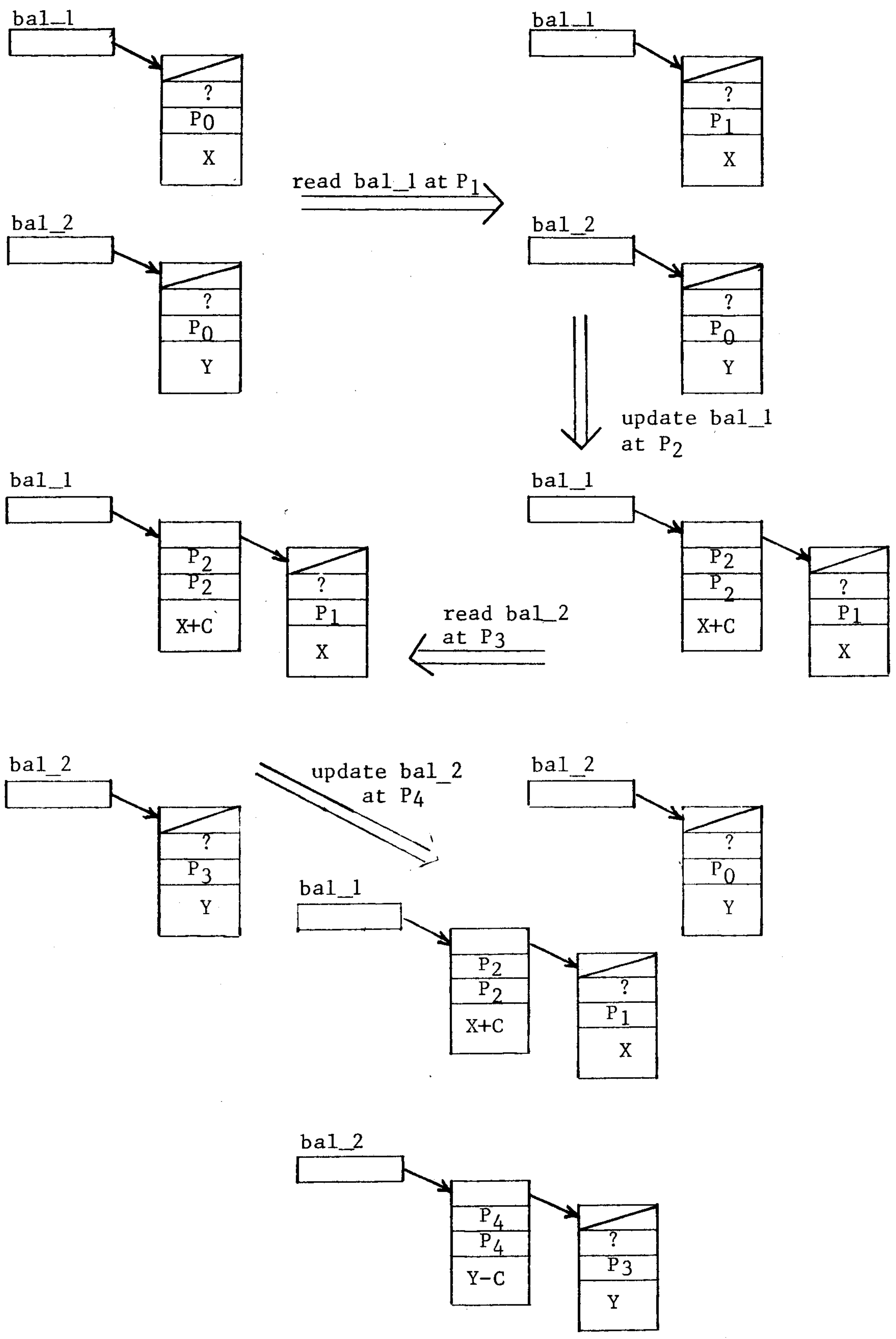
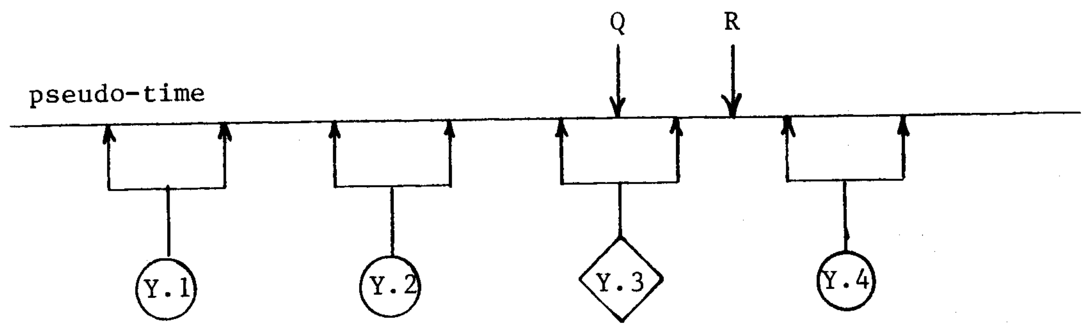
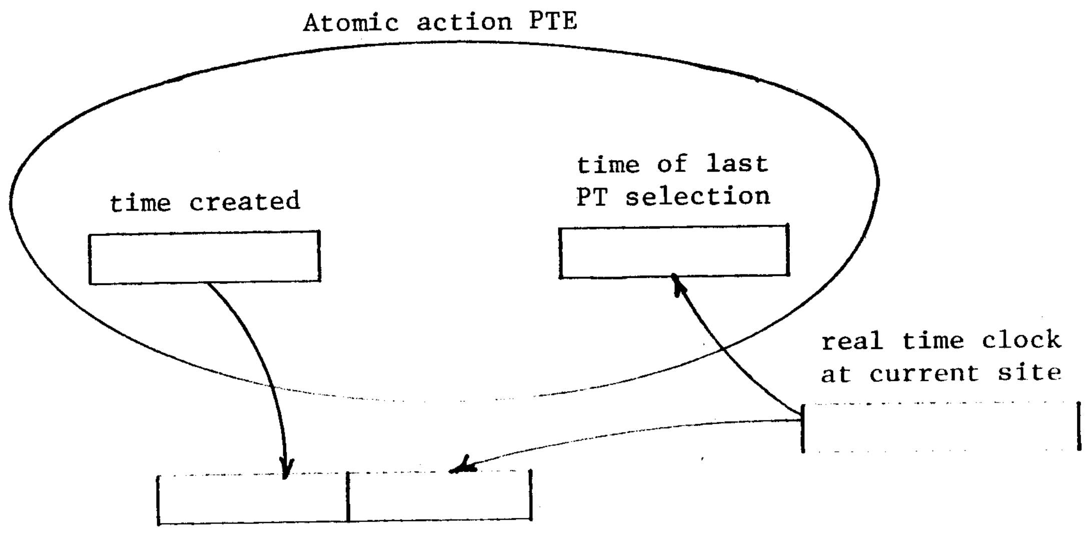


Figure G



Object history for Y

Figure H Selecting PT from PTE



pseudo-time (2N bits)